Typsysteme für die Dienstvermittlung in offenen verteilten Systemen

Dissertation zur Erlangung des
Doktorgrades der Naturwissenschaften

vorgelegt beim Fachbereich Informatik
der Johann Wolfgang Goethe–Universität
in Frankfurt am Main

von
Arno Puder
aus Toronto/Kanada

Frankfurt 1996
DF1

vom Fachbereich Informatik der Johann Wolfgang Goethe–Universität als Dissertation angenommen.

Dekan: _____

Gutachter: _____

Datum der Disputation: _____

„Where is the knowledge
we have lost in information?“
T.S. Eliot

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ANSI	American National Standards Institute
AI	Artificial Intelligence
CG	Conceptual Graph
CORBA	Common Object Request Broker Architecture
DCE	Distributed Computing Environment
EBNF	Extended Backus Naur Form
IDL	Interface Definition Language
IP	Internet Protocol
ISO	International Organization for Standardization
ITU–T	International Telecommunication Union, Telecommunication Standardization Sector
KI	Künstliche Intelligenz
KIF	Knowledge Interchange Format
KQML	Knowledge Query and Manipulation Language
ODP	Open Distributed Processing
OLE	Object Linking and Embedding
OMG	Object Management Group
ORB	Object Request Broker
OSF	Open Software Foundation
OSI	Open Systems Interconnection
RM	Reference Model
RPC	Remote Procedure Call
TCP	Transmission Control Protocol
UUID	Universal Unique Identifier
WWW	World Wide Web

Kapitel 1

Einleitung

1.1 Motivation

Die fortschreitende Vernetzung von Rechnersystemen hat aus einer Vielzahl unterschiedlicher, lokaler Systeme ein *offenes, verteiltes System* entstehen lassen, in welchem die verschiedensten Dienstleistungen frei angeboten und nachgefragt werden können. Immer neue Anbieter drängen auf diesen elektronischen Markt, genauso wie andere diesen wieder verlassen, was das System für den einzelnen Benutzer aufgrund seiner Komplexität und der permanenten Veränderungen, denen es unterworfen ist, unüberschaubar macht. Die Dynamik und Größe eines offenen verteilten Systems impliziert, daß der Wissensstand über den aktuellen Aufbau und Zustand in bezug auf das Dienstangebot proportional mit der Zeit abnimmt.

Verschiedene Gremien erarbeiten technologieunabhängige Standards, die Rahmenwerke für offene verteilte Systeme definieren. Beispielsweise wurde von den beiden Standardisierungsgremien *International Organization for Standardization* (ISO) und *International Telecommunication Union, Telecommunication Standardization Sector* (ITU–T) ein Referenzmodell für offene verteilte Systeme, das *Reference Model for Open Distributed Processing* (RM–ODP), entwickelt (siehe [42], [43], [44] und [45]). Das RM–ODP beschreibt eine Architektur, die die Unterstützung von Verteilung, Interoperabilität und Portabilität zum Ziel hat. Das RM–ODP definiert darüber hinaus die *Dienstvermittlung* in offenen verteilten Systemen, die eine systematische Suche nach Diensten ermöglicht (siehe [46]). Annahme dabei ist, daß Teilnehmer eines offenen verteilten Systems in den Rollen der *Dienstnutzer* und *Dienstanbieter* auftreten, die voneinander keine *a–priori*–Kenntnis besitzen und insbesondere nicht statisch aneinander gebunden sind.

Der Vorgang der Dienstvermittlung basiert nach dem RM–ODP auf einem Vermittlungsprotokoll (siehe Abbildung 1.1). Dazu wird neben dem Dienstnutzer und Dienstanbieter noch ein Vermittler eingeführt, der die Aufgabe der Zuordnung von Angebot und Nachfrage übernimmt. Ein Vermittlungsvorgang läuft in mehreren Schritten ab. Zunächst macht der Dienstanbieter sein Angebot bei dem Vermittler bekannt, der dieses Angebot in seiner Datenbank speichert (1). Anschließend steht der Dienstanbieter potentiellen Interessenten zur Verfügung. Dabei fragt ein Dienstnutzer den Vermittler nach einem Dienst, auf den der Dienstnutzer Zugriff wünscht (2). Ist ein passendes Angebot vorhanden, antwortet der Vermittler dem Dienstnutzer mit einem Verweis auf den entsprechenden

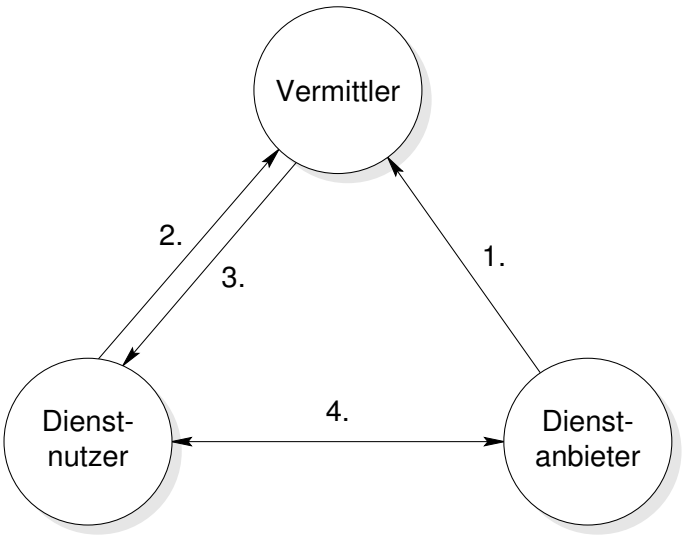


Abbildung 1.1: Interaktionen bei einem Vermittlungsvorgang.

Dienstanbieter (3). Nach der erfolgreichen Vermittlung wird der Dienstnutzer an den Dienstanbieter gebunden, der dann die von ihm erbrachte Funktionalität dem Dienstnutzer zugänglich macht (4).

Innerhalb des Internet übernehmen Suchmaschinen die Aufgabe der Dienstvermittlung (siehe [16]). Der Unterschied zu dem oben diskutierten Vermittlungsvorgang besteht darin, daß eine Suchmaschine automatisch das Angebot des Internet analysiert und daraus einen Index erstellt. Der oben erläuterte Schritt 1, bei dem ein Dienstanbieter sein Angebot explizit bekannt macht, entfällt hier. Die Suchmaschinen stellen *ad–hoc*–Lösungen zu dem allgemeinen Problem der Dienstvermittlung dar. Es zeigt sich, daß mit zunehmendem Dienstangebot die Qualität der Dienstvermittlung in bezug auf die Güte der vermittelten Dienste abnimmt. Dieser Sachverhalt war bereits Gegenstand von Diskussionen in der Tagespresse (siehe beispielsweise [96] oder [111]).

Eine zentrale Bedeutung für einen Vermittlungsvorgang kommt dem *Typsystem* zu. Das Typsystem legt zum einen eine Beschreibungssprache fest, mit der in einem offenen verteilten System Dienste spezifiziert werden können (siehe [85]). Weiterhin definiert es Regeln, nach denen Angebot und Nachfrage, dargestellt in Form von Dienstspezifikationen, zugeordnet werden. Im Rahmen des RM–ODP liegt der Schwerpunkt bei der Spezifikation eines Dienstes auf operationalen Aspekten, d.h. wie ein Dienst auf einer programmiertechnischen Ebene zugreifbar ist. Im globalen System mit seiner differenzierten Benutzerstruktur, wo neben Programmierern auch Anwender mit der Dienstvermittlung in Berührung kommen, gewinnt dagegen die Auswahl von Dienstspezifikationen eine herausragende Bedeutung (siehe [87]). Damit verbunden treten die semantischen Eigenschaften eines Dienstes als Auswahlkriterien in den Vordergrund, da hier dem Konsumenten eine Vielzahl von Diensten mit gleichem operationalen Verhalten, aber unterschiedlichen Eigenschaften angeboten werden.

1.2 Ziele der Arbeit

In dieser Arbeit wird die Dienstvermittlung in offenen verteilten Systemen untersucht. Dabei ist die Rolle, die ein Typsystem bei einem Vermittlungsvorgang spielt, von besonderem Interesse. Die Ziele dieser Arbeit gliedern sich in mehrere Teilziele auf. Das erste Ziel besteht in der Definition eines Typsystems. Auf Basis dieser Definition kann die Zuordnung von Angebot und Nachfrage während eines Vermittlungsvorgangs formal dargestellt werden. Als nächstes werden die Anforderungen an die Dienstvermittlung in offenen verteilten Systemen erarbeitet. Es wird sich dabei herausstellen, daß das von dem RM–ODP vorgeschriebene Vermittlungsmodell nicht ausreichend ist. Daraus leiten sich die Anforderungen an Typsysteme in offenen verteilten Systemen ab. Eine Diskussion existierender Typsysteme wird zeigen, daß diese den Anforderungen nicht gerecht werden.

Schwerpunkt der Arbeit liegt auf der Entwicklung von Typsystemen, die den postulierten Anforderungen für die Dienstvermittlung in offenen verteilten Systemen genüge leisten. Es werden ein deklaratives und ein wissensbasiertes Typsystem erarbeitet, welche Dienstbeschreibungen auf unterschiedlichen Abstraktionsebenen einer differenzierten Benutzerstruktur erlauben. Die dem wissensbasierten Typsystem zugrundeliegende Wissensrepräsentationstechnik ermöglicht dabei auch operationale Dienstspezifikationen.

Als weiteres Ziel dieser Arbeit soll nachgewiesen werden, daß ein wissensbasiertes Typsystem im Sinne einer *unifying theory* sowohl den Anforderungen des Rahmenwerks des RM–ODP entspricht, als auch den pragmatischen Ansatz der Dienstvermittlung im Internet auf eine fundierte Grundlage stellt. Neben den theoretischen Aspekten der in dieser Arbeit vorgestellten wissensbasierten Dienstvermittlung soll als letztes Ziel die praktische Umsetzbarkeit nachgewiesen werden. Im Zuge dessen wird die Architektur eines Prototyps beschrieben, der die Praktikabilität der wissensbasierten Dienstvermittlung demonstriert.

1.3 Aufbau der Arbeit

Die vorliegende Arbeit teilt sich in zehn Kapitel auf. Ausgehend von den Grundlagen der Dienstvermittlung in offenen verteilten Systemen werden zunächst Anforderungen an Typsysteme formuliert. Der Hauptteil dieser Arbeit stellt neue Ansätze bei der Definition von Typsystemen vor, die den zuvor formulierten Anforderungen genügen.

Kapitel 2 führt die Grundlagen ein, die für den Rest der Arbeit notwendig sind. Das Objektmodell wird beschrieben und der Begriff des Typsystems, so wie er in dieser Arbeit verwandt wird, eingeführt.

Kapitel 3 definiert zunächst den Begriff des offenen verteilten Systems, um anschließend die Notwendigkeit der Dienstvermittlung in derartigen Umgebungen aufzuzeigen. Dabei wird die besondere Rolle, die ein Typsystem während eines Vermittlungsvorgangs spielt, hervorgehoben. Als wichtigstes Ergebnis dieses Kapitels werden Anforderungen formuliert, welche Typsysteme in offenen verteilten Systemen erfüllen müssen.

Kapitel 4 stellt einige existierende Typsysteme exemplarisch vor, die in heutigen offenen verteilten Systemen zum Einsatz kommen. Neben ihrer Beschreibung werden

die vorgestellten Typsysteme bzgl. der im vorangegangenen Kapitel erarbeiteten Anforderungen bewertet.

Kapitel 5 präsentiert eine formale Spezifikation der Dienstvermittlung in offenen verteilten Systemen auf Basis der Spezifikationssprache Z. Die in dieser Arbeit vorgestellte Spezifikation verfeinert die im RM–ODP enthaltene Spezifikation eines Vermittlers, die ebenfalls in Z ausgeführt ist.

Kapitel 6 stellt ein neues Typsystem vor, welches die postulierten Anforderungen an Typsysteme in offenen verteilten Systemen erfüllt. Das resultierende deklarative Typsystem, das eine Erweiterung eines Typsystems nach den Definitionen des RM–ODP darstellt, erlaubt inhaltsorientierte Dienstspezifikationen auf Basis der deklarativen Semantik.

Kapitel 7 erweitert das im vorangegangenen Kapitel vorgestellte deklarative Typsystem dahingehend, daß es semantische Dienstspezifikationen auf unterschiedlichen Abstraktionsebenen erlaubt. Ausgangspunkt für diesen Übergang zu einem wissensbasierten Typsystem bildet die Theorie der Konzeptgraphen, einer Wissensrepräsentationstechnik aus der künstlichen Intelligenz.

Kapitel 8 beschreibt die Architektur eines wissensbasierten Dienstvermittlers. Seine Struktur wird durch das im vorangegangenen Kapitel vorgestellte wissensbasierte Typsystem geprägt. Im Anschluß werden einige Einsatzmöglichkeiten der wissensbasierten Dienstvermittlung aufgezeigt.

Kapitel 9 beschreibt einen Prototypen, der die Konzepte der wissensbasierten Dienstvermittlung umsetzt. Das Kapitel beschreibt die Struktur des Prototypen, die Ziele, die bei seiner Erstellung verfolgt wurden, sowie die Hilfswerkzeuge, mit denen die Implementierung geleistet wurde.

Kapitel 10 faßt die Ergebnisse dieser Arbeit zusammen und liefert eine abschließende Bewertung sowie einen Ausblick auf zukünftige Arbeiten.

Kapitel 2

Grundlagen

Die informationstechnischen Systeme, die Gegenstand dieser Arbeit sind, zeichnen sich durch eine inhärente Komplexität aus. Die Komplexität entsteht quantitativ wegen der zunehmenden Größe dieser Systeme und qualitativ wegen höherer Anforderungen an deren Funktionalität. Die methodische Strukturierung eines Systems ist Voraussetzung, der Komplexität zu begegnen. Ein Modell erlaubt die systematische Darstellung informationstechnischer Systeme. Die Prinzipien des Objektmodells sind in besonderem Maße geeignet, komplexe Problembereiche darzustellen. So beruhen beispielsweise die Arbeiten der OMG und der ISO auf den Konzepten des Objektmodells.

Wegen der Bedeutung des Objektmodells für diese Arbeit werden in Abschnitt 2.1 dessen Konzepte vorgestellt. Das Objektmodell induziert eine Terminologie, mit der ein Problembereich systematisch gemäß den Prinzipien des Objektmodells dargestellt werden kann. Für die in Abschnitt 2.2 eingeführte Terminologie steht der Begriff des Objekts im Mittelpunkt. Die Einführung von Typen erlaubt eine Abstraktion von gleichartigen Objekten. Die Klassifikation auf Basis von Typen faßt Mengen von Objekten nach bestimmten Kriterien zusammen. Die Bestandteile des in Abschnitt 2.3 eingeführten Begriffs eines Typsystems sind eine Typbeschreibungssprache und eine Typkonformität.

2.1 Das Objektmodell

Ein *Modell* unterstützt die systematische Darstellung eines Problembereichs, um dessen Untersuchung bzw. Erforschung zu erleichtern oder erst zu ermöglichen. Das Modell beschreibt, nach welchen Prinzipien diese Darstellung während der Modellierung erzeugt wird. In dem Modell sind Konzepte verankert, die eine Philosophie in der Betrachtungsweise eines Problembereichs widerspiegelt. Jedes Modell definiert dazu Konzepte, die bestimmte Aspekte des zu modellieren Problembereichs fokusiert. Ziel des *Objektmodells* ist die systematische Darstellung eines Problembereichs auf Basis der Konzepte *Abstraktion*, *Modularität*, *Kapselung* und *Hierarchie* (siehe [15] und [92]), die im folgenden erläutert werden.

Abstraktion: Eine Abstraktion vergröbert die Beschreibung eines Problembereichs, indem es zwischen relevanten und nicht relevanten Aspekten unterscheidet. Die Vereinfachung einer Darstellung ist notwendige Voraussetzung, der Komplexität eines

Problembereichs zu begegnen. Von einem Problembereich können unterschiedliche Abstraktionen erstellt werden. Die Entscheidung, welche Aspekte für eine Abstraktion relevant sind, ist von der subjektiven Sichtweise eines Betrachters abhängig. Darüber hinaus legt die Abstraktion die Terminologie fest, mit der die relevanten Eigenschaften eines Problembereichs beschrieben werden.

Modularität: Die Modularisierung partitioniert einen Problembereich. In Abhängigkeit einer subjektiven Sichtweise kann die Zerlegung auf unterschiedliche Art und Weise durchgeführt werden. Die von der Partition definierten Teilmengen, die Teilaspekte des Problembereichs widerspiegeln, zeichnen sich durch wohldefinierte Grenzen gegenüber den anderen Teilmengen aus. Eine Teilmenge des Problembereichs repräsentiert einen eigenen Problembereich.

Kapselung: Die Kapselung beschreibt den Vorgang, alle Details, die nicht als relevant für eine Darstellung klassifiziert werden, zu verbergen. Während die Abstraktion eine äußere Sicht beschreibt, verbirgt die Kapselung die innere Sicht auf die Implementation des von der Abstraktion festgelegten Verhaltens. Die Kapselung garantiert, daß keine Abhängigkeiten auf Grund interner Implementationsdetails entstehen. Die Konsequenz aus der Kapselung ist die Notwendigkeit einer Schnittstelle, die die äußere von der inneren Sicht trennt. Alle Informationen, die ein Betrachter von außen einer Darstellung entnehmen kann, sind ausschließlich in der Schnittstelle zu der Darstellung vereinigt.

Hierarchie: Eine Hierarchie setzt Abstraktionen unterschiedicher Teilprobleme zueinander in Beziehung. Zwei im Objektmodell gebräuchliche Hierarchien sind die *Spezialisierung* (englisch: *is–a*) und die *Aggregation* (englisch: *part–of*). Die Spezialisierungsbeziehung besteht zwischen zwei Abstraktionen, wenn eine der Abstraktionen alle Eigenschaften der anderen besitzt. Die Aggregationsbeziehung erlaubt einer Abstraktion die Bezugnahme auf eine andere Abstraktion für ihre eigene Definition.

Die Strukturierung eines Problembereichs gemäß dem Objektmodell orientiert sich an den vier Konzepten Abstraktion, Modularität, Kapselung und Hierarchie. Die resultierende Darstellung des Problembereichs als Ergebnis der Modellierung erlaubt Freiheitsgrade, die von dem subjektiven Standpunkt des Betrachters abhängen. Die kanonische Darstellung eines Problembereichs auf Basis der Konzepte des Objektmodells exisitiert nicht.

Das Objektmodell selbst schreibt keine konkreten Mechanismen für die Unterstützung der oben eingeführten Konzepte in informationsverarbeitenden Systemen vor. Die *Wiederverwendbarkeit* ist beispielsweise kein weiteres Konzept des Objektmodells, sondern stellt bereits einen Mechanismus dar. Sowohl die Modularität als auch die Hierarchie bilden die Grundlage für die Wiederverwendbarkeit. Hierdurch können Einheiten eines Problembereichs wiederverwendet werden. Bei der Spezialisierungsbeziehung ergibt sich die Wiederverwendbarkeit aus der Übernahme von Eigenschaften einer weniger spezialisierten Darstellung.

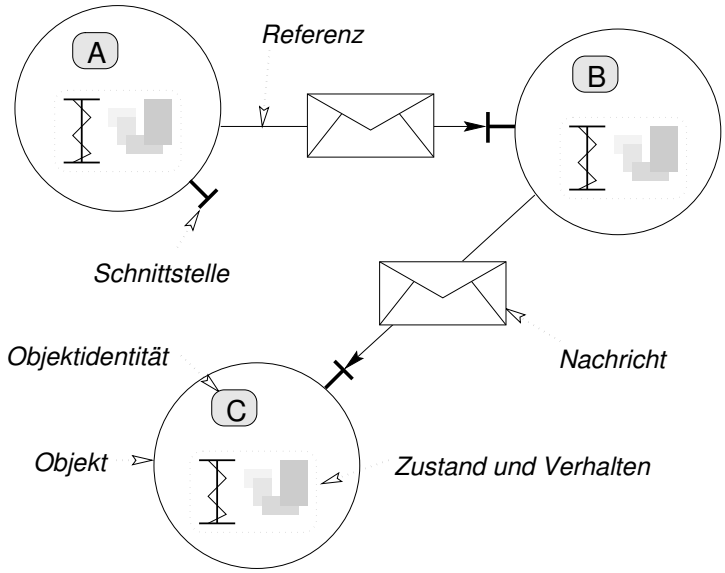


Abbildung 2.1: Dekomposition eines Problembereichs in eine Menge von Objekten.

2.2 Terminologie der Objektmodellierung

Die im letzten Abschnitt vorgestellten Konzepte des Objektmodells kommen bei der Strukturierung eines komplexen Problembereichs zur Anwendung. Das Ergebnis der Modellierung ist eine Darstellung, die als Ausgangspunkt für weitere Untersuchungen und Diskussionen dient. Um eine systematische Darstellung zu ermöglichen, ist die Einführung einer einheitlichen Terminologie notwendig. Die aus dem Objektmodell abgeleitete Terminologie führt zu allgemein akzeptierten Darstellungen von Problembereichen. Da nicht davon ausgegangen werden kann, daß ein Problembereich zeitlich unabhängig ist, muß bei der Wahl der Terminologie zwischen statischen und dynamischen Aspekten unterschieden werden. Eine Darstellung auf Basis der statischen Aspekte spiegelt einen stationären Zustand des modellierten Problembereichs wider, während durch die dynamischen Aspekte zeitabhängige Eigenschaften beschrieben werden.

2.2.1 Statische Aspekte

Der zentrale Begriff, dem das Objektmodell seinen Namen verdankt, ist das *Objekt* (siehe [20]). Die Dekomposition eines Problembereichs gemäß der im Objektmodell verankerten Konzepte führt zu einer Menge von Objekten. Ein Objekt ist ein Ding unserer Vorstellung, das sowohl abstrakter als auch konkreter Natur sein kann. Obwohl die Wahl der Granularität eines Objektes relativ zum Betrachter ist, grenzt sich ein Objekt ausschließlich auf Grund seiner Definition durch wohldefinierte Schnittstellen von anderen Objekten ab. Die Granularität in der Betrachtungsweise des Problembereiches legt dabei den Abstraktionsgrad fest. Die Abgrenzung zwischen Objekten bewirkt die im Objektmodell geforderte Modularisierung. Ein Objekt stellt somit einen für sich abgeschlossenen Problembereich dar.

Ein Objekt ist Träger einer *Identität*. Alle Objekte unterscheiden sich aufgrund ihrer eindeutigen Identität. Der Begriff der Identität ist dabei so zu verstehen, daß die Objekte durch ihre inhärente Existenz unterscheidbar sind und nicht aufgrund ihrer Eigenschaften[1]. Neben seiner Identität ist ein Objekt durch seinen *Zustand* und sein *Verhalten* charakterisiert. Diese beiden Eigenschaften sind nicht orthogonal zueinander, da das Verhalten vom aktuellen Zustand beeinflußt wird. Zustand und Verhalten definieren die *Funktionalität*, die ein Objekt erbringt und die von anderen Objekten genutzt werden kann. Die *Implementation* realisiert die Funktionalität eines Objekts, basierend auf einer Programmiersprache. Die Konzepte des Objektmodells müssen in der Programmiersprache durch Konventionen oder besondere Sprachkonstrukte unterstützt werden. Beipielsweise entsprechen in einer imperativen Programmiersprache die Begriffe Zustand und Verhalten den programmiersprachlichen Konstrukten Daten und Funktionen.

Eine Konsequenz der im Objektmodell geforderten Kapselung ist die Notwendigkeit einer wohldefinierten *Schnittstelle* eines Objekts, die anderen Objekten als Zugangspunkt dient und somit Zugriff auf dessen Funktionalität ermöglicht. Die Schnittstelle trennt eine äußere und eine innere Sicht auf ein Objekt. Während die äußere Sicht Aufschluß über die Funktionalität eines Objekts gibt, legt die innere Sicht Implementationsdetails offen. Eine Schnittstelle als fester Zugangspunkt reduziert die Abhängigkeiten zwischen Objekten. Implementationen können ausgetauscht werden, solange die nach außen hin sichtbare Funktionalität gewahrt bleibt.

Die Identität erlaubt die eindeutige Adressierung einer Schnittstelle (siehe Abbildung 2.1). An ein Objekt können Anfragen in Form von *Nachrichten* an seine Schnittstelle gerichtet werden, die es wiederum durch Nachrichten beantwortet. Der Empfang einer Nachricht stellt ein Ereignis dar, das den Objektzustand verändern kann. Ereignisse stehen in einer kausalen Abhängigkeit, da der Empfang einer Nachricht zu einer Abfolge weiterer Nachrichten führen kann.

Eine Nachricht besteht aus einem *Selektor* und einer endlichen Anzahl von *Parametern* (siehe [32]). Über den Selektor wird eine Auswahl bei der von einem Objekt erbrachten Funktionalität getroffen. Ein Objekt kann verschiedene Arten von Nachrichten, die sich in ihren Selektoren unterscheiden, akzeptieren. Die Menge der Selektoren aller Nachrichten, die ein Objekt an seiner Schnittstelle akzeptiert, partitionieren hierdurch die von dem Objekt angebotene Funktionalität. Die in einer Nachricht enthaltenen Parameter werden als *Eingabeparameter* (bzw. *Ausgabeparameter*) bezeichnet, wenn sie in einer Anfrage (bzw. Antwort) enthalten sind.

Das Objektmodell schreibt keine präzise Struktur von Selektoren und Parametern vor. Die Struktur basiert üblicherweise auf einem Programmiersprachenparadigma. Wird für die innere Struktur einer Nachricht beispielsweise eine imperative Programmiersprache zugrunde gelegt, entspricht dem Selektor ein Funktionsname und die geordnete Folge von Parametertypen[2]. Ein Selektor wird dann auch als die *Signatur* einer Nachricht be-

[1] Die aus der Mathematik bekannte Lambda–Funktion ist ein Beispiel für ein Konstrukt, welches zwar fest definierte *Eigenschaften* besitzt, jedoch keine *Identität*. Lambda–Funktionen werden deshalb auch als *unbenannte Funktionen* bezeichnet.

[2] Ein Beispiel für eine andere Ausprägung der Begriffe Selektor und Parameter entstammt den intelligenten Agentensystemen. Ein Selektor bei einem intelligenten Agentensystem besteht nach [30] aus einem Wort der *Knowledge Query and Manipulation Language* (KQML) und die Parameter aus Elementen des *Knowledge Interchange Format* (KIF) (siehe auch [29] und [31]).

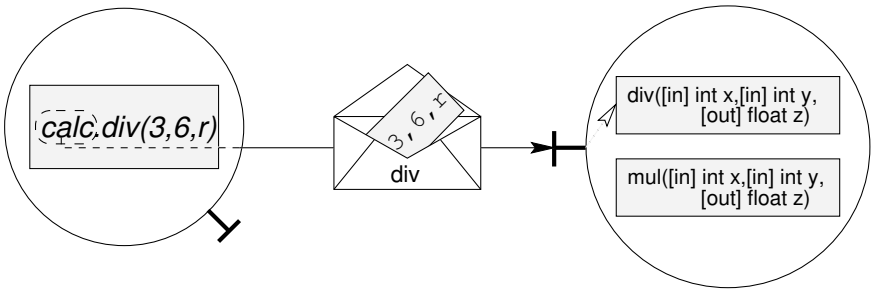


Abbildung 2.2: Methodenauswahl anhand der Signatur einer Nachricht.

zeichnet und die Schnittstelle als *operationale Schnittstelle*. Der Teil der Implementation, der die Funktionalität einer Nachricht in einem Objekt erbringt, wird als *Methode* oder *Operation* bezeichnet. Der Prozeß der Zuordnung einer eintreffenden Nachricht zu einer implementierenden Methode heißt *Methodenauswahl* (englisch: *Method Dispatching*).

2.2.2 Dynamische Aspekte

Das Objektmodell erlaubt die Dekomposition eines Problembereichs in eine Menge von Objekten. Im letzten Abschnitt wurde die Terminologie eingeführt, mit der die statischen Aspekte des Problembereichs beschrieben werden können. Die Menge von Objekten, die aus der Modellierung hervorgehen, sind nicht unabhängig voneinander, sondern erzielen durch ein Zusammenwirken die Funktionalität des modellierten Problembereichs. Eine systematische Darstellung der dynamischen Vorgänge basiert auch in diesem Fall auf einer Terminologie, die Gegenstand dieses Abschnitts ist.

Das Zusammenwirken einer Menge von Objekten basiert auf einer *Kooperation*. Die Kooperation definiert Ziele, die durch eine koordinierte Kommunikation von Objekten erreicht werden. Die *Kommunikation* beschreibt den Vorgang des Nachrichtenaustauschs zwischen Objekten. Die *Koordination* steuert den zeitlichen Ablauf der Kommunikation, um aus einzelnen Kommunikationsvorgängen das übergeordnete Kooperationsziel zu erbringen. Die Kooperation ist definiert durch das Zusammenwirken von Kommunikation und Koordination (siehe [91]):

$$Kooperation = Kommunikation \ \& \ Koordination$$

Die Kommunikation zwischen Objekten basiert auf einem Nachrichtenaustausch. Ein Objekt kann an die Schnittstelle eines anderen Objekts eine Nachricht schicken. Voraussetzung für einen Kommunikationsvorgang ist, daß der Absender einer Nachricht eine *Referenz* auf den Adressaten besitzt (siehe [32]). Eine Referenz verkörpert das Wissen um eine Objektidentität, das zur eindeutigen Adressierung notwendig ist. Eine Referenz manifestiert eine *Sicht*, die ein Objekt auf das referenzierte Objekt besitzt. Ein *Ausdruck* beschreibt die Anwendung einer Nachricht auf ein referenziertes Objekt. Beispielsweise bezeichnet der Ausdruck $calc.div(3, 6, r)$ die Anwendung einer Nachricht auf ein Objekt mit einer operationalen Schnittstelle, welches über den Bezeichner *calc* referenziert wird (siehe Abbildung 2.2).

Die Nachricht in diesem Beispiel besitzt die Signatur *div*(`[in] int`, `[in] int`, `[out] float`). Die Schlüsselwörter `[in]` und `[out]` bezeichnen Ein– und Ausgabeparameter der Operation. Eingabeparameter sind an konkrete Werte gebunden (wie die Werte 3 und 6 in Abbildung 2.2). Der Bezeichner r wird nach Ausführung der Operation an das Resultat gebunden und in einer Antwortnachricht an den Aufrufer übermittelt. Die informelle Semantik der `div`–Operation ist, daß der Bezeichner r als Ergebnis den Quotienten der beiden Eingabeparameter enthält. Der Ausdruck ist Teil einer Implementierung eines Objekts (dem linken Objekt in Abbildung 2.2). Der Bezeichner *calc* referenziert die operationale Schnittstelle eines anderen Objekts. Anhand der Signatur der eintreffenden Nachricht wird die Methode *div*(`[in] int` x, `[in] int` y, `[out] float` z) ausgewählt. Die aktuellen Parameter 3, 6 und r werden an die formale Parameter x, y und z der Methode gebunden und die Implementierung ausgeführt.

Die Kooperation ist ein dynamischer Vorgang, für den eine Menge von Objekten über einen Zeitraum hinweg untereinander in *Beziehung* stehen. Den Objekten, die an einer Kooperation teilnehmen, können aufgrund ihrer Beziehungen zu anderen Objekten Rollen zugeordnet werden. Eine *Rolle* beschreibt die Aufgabe, die das Objekt innerhalb einer Beziehung übernimmt. Beispielsweise setzt eine *Client/Server*–Kooperation zwei Objekte in eine Beziehung, die den Objekten die Rolle des Nutzers (Client) und des Anbieters (Servers) zuweist (siehe [26]). Die Kooperation zweier Objekte in diesen Rollen hat zum Ziel, dem Nutzer die Funktionalität des Anbieters zugänglich zu machen. Erreicht wird dies durch einen koordinierten Nachrichtenaustausch. Der Client schickt dem Server zuerst eine Nachricht, die der Server nach einer Bearbeitung mit einer weiteren Nachricht an den Client beantwortet.

2.3 Definition eines Typsystems

Ein *Typsystem* setzt sich aus einer Definition eines Typs und einer Typkonformität zusammen. Im folgenden werden die Begriffe Typ und Typkonformität repektive ihrer Eigenschaften eingeführt und nach bestimmten Kriterien klassifiziert. In den Abschnitten 2.3.1 und 2.3.2 wird detailliert auf die Eigenschaften von Typen und Typkonformität eingegangen. Die Diskussion geeigneter Mechanismen, die ein Typsystem realisieren, erfolgt später in Kapitel 4.

2.3.1 Typen

Typen erlauben die Klassifikation von Objekten nach vorgegebenen Kriterien. Danach können Teilmengen von Objekten nach bestimmten Eigenschaften kategorisiert werden. Ein Typ ist eine Spezifikation von Bedingungen, die, wenn sie auf ein Objekt zutreffen, das Objekt dem Typ zuordnen. Treffen die im Typ formulierten Bedingungen auf ein Objekt zu, so ist dieses eine *Instanz* des Typs.

Ein Typ kann unterschiedlichen Zwecken dienen (siehe [81]):

- *Sicherheit*: erlaubt die automatische Erkennung bestimmter Programmierfehler.

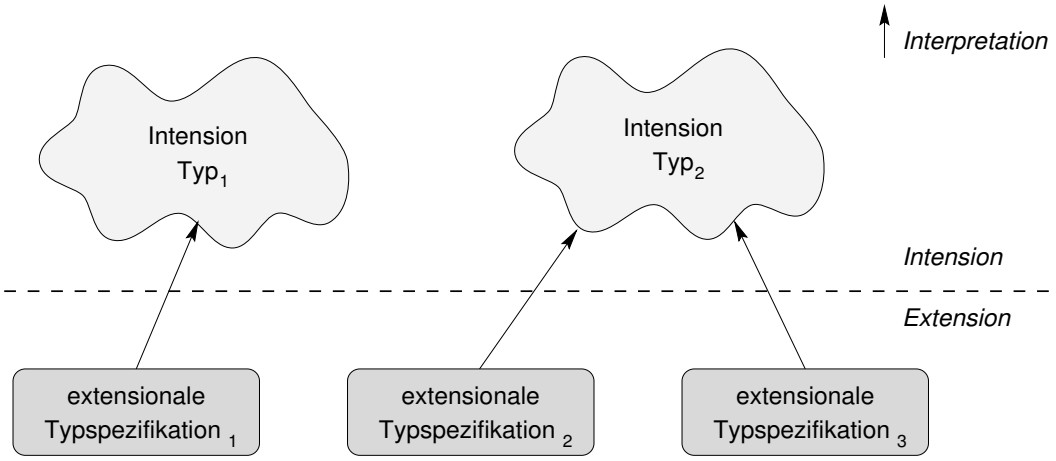


Abbildung 2.3: Extension und Intension eines Typs.

- *Abstraktion*: Kategorisierung abstrahiert von unwesentlichen Details.
- *Dokumentation*: unterstützt die Lesbarkeit von Programmen.
- *Optimierung*: ermöglicht eine effiziente Codegenerierung.

Typen dienen traditionell in erster Linie der Sicherheit von Programmsystemen. Zunächst ist festzustellen, daß der Übergang von einer untypisierten zu einer typisierten Programmiersprache sich nicht auf die Berechenbarkeit auswirkt (d.h. durch die Einführung von Typen können keine neuen Problemklassen, die vorher nicht berechenbar waren, gelöst werden). Redundanzen bei den Typangaben in einem Ausdruck und entsprechende Inferenzregeln erlauben die Erkennung einer Vielzahl von Fehlern bereits während der Entwicklungsphase eines Programms. Dadurch, daß Typen wie ein Schutzmantel um ein Objekt wirken, schützen sie vor unzulässigen Zugriffen.

Neben der Sicherheit unterstützen Typen Abstraktionen oberhalb der Objektebene, da ein Typ mehrere Objektinstanzen logisch zusammenfaßt. Die Einführung von Typen hilft hierdurch bei der Strukturierung komplexer Problembereiche. Zudem erhöhen typisierte Programme die Lesbarkeit, da die Typannotationen Hinweise auf die Verwendung der typisierten Objekte geben. Desweiteren ist eine effizientere Codegenerierung bei typisierten Programmen möglich, da beispielsweise die Typisierung eines Objekts eine verbesserte Speicherverwaltung erlaubt (beispielsweise bei der Unterscheidung zwischen dem Speicherbedarf von Instanzen der Typen `long int` und `short int`).

Ein Typ besitzt eine *Intension* und eine *Extension* (siehe Abbildung 2.3). Die Intension eines Typs ist abstrakt in dem Sinne, daß sie eine Vorstellung über die Eigenschaften des Typs verkörpert. Eine extensionale Beschreibung vergegenständlicht dahingegen die Intension eines Typs auf einer Beschreibungsebene. Während die Intension einer Vorstellung entspringt, ist jede Darstellung der Intension eine ihrer möglichen Extensionen. Die Intension eines Typs ist feststehend, obwohl deren Extensionen äußerlich unterschiedlich sein können. Es existiert beispielsweise auf einer intensionalen Ebene eine klare Vorstellung über den Begriff der „natürlichen Zahlen“. Um diesen Begriff zu beschreiben,

existieren unterschiedliche Extensionen. In der Mathematik sind die natürlichen Zahlen durch die Peano–Axiome formal definiert, wohingegen in der Programmiersprache C++ sie durch den Bezeichner `int` charakterisiert sind.

Eine Abbildung zwischen den Extensionen und den Intensionen bildet eine *Interpretation* der Typen. Die Interpretation ordnet jeder extensionalen Beschreibung eine Intension zu. Die Interpretation kann hierdurch als mathematische Funktion verstanden werden, die als Wertebereich die extensionalen Beschreibungen und als Bildbereich die Intensionen besitzt. Ist die Interpretationsfunktion injektiv, besitzten je zwei Extensionen unterschiedliche Intensionen. Die Extensionen sollen in diesem Fall als *eindeutig* bezüglich der Interpretation genannt werden. Ist die Interpretationsfunktion nicht injektiv, können zwei extensionale Beschreibungen die gleiche Intension besitzen. Die Extensionen sollen entsprechend als *nicht eindeutig* bezüglich der Interpretation bezeichnet werden.

Unter einer Typspezifikation soll im folgenden eine extensionale Beschreibung eines Typs verstanden werden. Eine Typspezifikation bedient sich einer Notation, der *Typbeschreibungssprache*. Mit einer Typbeschreibungssprache ist eine formale Sprache gemeint, deren Wörter gerade den syntaktisch korrekten Typspezifikationen entsprechen. Ein Typ wird hierdurch als ein Wort, gebildet aus einer Grammatik, repräsentiert. Der *Typraum* ist die Menge aller aus einer Typbeschreibungssprache ableitbaren Typspezifikationen. Eine Typbeschreibungssprache einer Programmiersprache setzt sich typischerweise aus einer Menge von elementaren Typen zusammen, wie beispielsweise `int`, `real` oder `bool`, sowie aus einer Konstruktionsvorschrift für zusammengesetzte Typen, wie `struct`, `union` oder `array` in der Programmiersprache C++. Eine herkömmliche Typbeschreibungssprache für offene verteilte Systeme erlaubt darüber hinaus die Beschreibung einer operationalen Schnittstelle, wie beispielsweise die in Kapitel 4 vorgestellten Typbeschreibungssprachen von DCE und CORBA.

Eine Typbeschreibungssprache wird unterschiedlichen Abstraktionsniveaus zugeordnet. Die Arten der Typbeschreibungssprachen werden nach [112] in *syntaktisch*, *semantisch* und *pragmatisch* unterschieden (siehe Abbildung 2.4). Der Detaillierungsgrad einer Typbeschreibungssprache nimmt von syntaktischen über semantischen bis pragmatischen Beschreibungen zu, so daß eine pragmatische Typspezifikation mehr Details als eine syntaktische Typspezifikation aufweist. Letztere besteht aus einer Menge von Selektoren, die die an einer Schnittstelle eines Objekts akzeptierten Nachrichten festlegen. Das Verhalten eines Objekts ist hier nicht expliziter Teil der Beschreibung, weswegen diese Spezifikationsform die Intension nicht ausreichend darstellt. Syntaktische Typbeschreibungssprachen besitzen jedoch den Vorteil, daß sie einfach handhabbar sind.

Semantische Typbeschreibungssprachen definieren neben der Schnittstelle eines Objekts noch das Verhalten der einzelnen Methoden. Die Bedeutung einzelner Methoden ist hier expliziter Bestandteil der Spezifikation. Die Verhaltensbeschreibung in einer semantischen Typbeschreibungssprache geht jedoch nicht auf implementationsspezifische Details ein. Objekte mit unterschiedlichen Implementierungen können deshalb dem gleichen Typ angehören. Semantische Spezifikationen basieren auf einer Abstraktion des Objektzustands. Alle Zustände, die ein Objekt einnehmen kann, werden zu einem abstrakten Zustandsraum zusammengefaßt. Die Struktur des Zustandsraums einer konkreten Objektimplementierung wird durch die Abstraktion verdeckt. Das Verhalten einer Methode läßt sich als Transition in diesem Zustandsraum beschreiben.

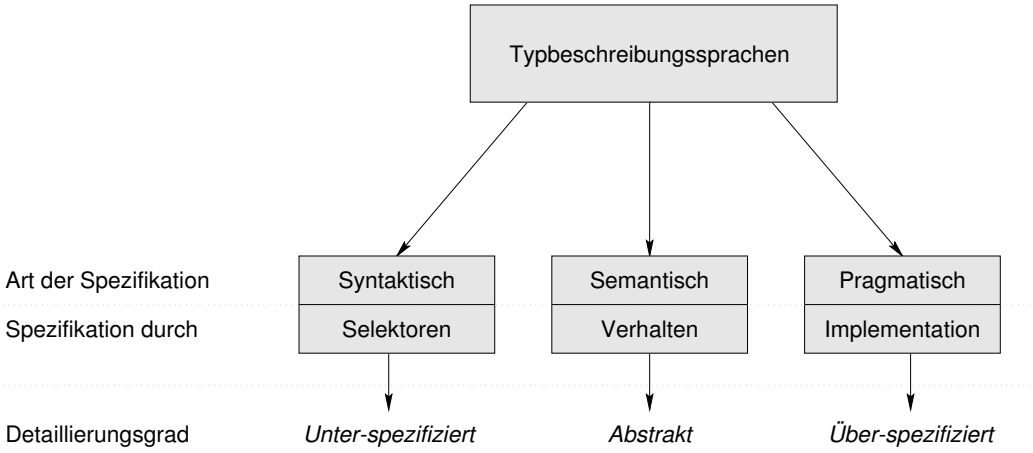


Abbildung 2.4: Abstraktionsniveaus von Typbeschreibungssprachen.

Pragmatische Typbeschreibungssprachen zeichnen sich ebenfalls durch eine explizite Verhaltensbeschreibung aus. Im Unterschied zu den semantischen Typspezifikationen handelt es sich jedoch um ausführbare Beschreibungen mit einem hohen Detaillierungsgrad. Da eine ausführbare Beschreibung an eine Programmiersprache (und damit auch an ein Programmierparadigma) gebunden ist, wird die pragmatische Typbeschreibungssprache als überspezifiziert eingestuft. Diese Form von Typsystemen findet häufig in objektorientierten Programmiersprachen Verwendung, wo anstatt von Typen auch von *Klassen* gesprochen wird. Tabelle 2.1 gibt einen Überblick über die Vor– und Nachteile der verschiedenen Abstraktionsebenen von Typbeschreibungssprachen.

2.3.2 Typkonformität

Typen erlauben die Klassifikation von Objekten. Eine Typspezifikation ist eine extensionale Beschreibung eines Objekts in einem Typsystem. Typisierte Objekte kooperieren durch eine koordinierte Kommunikation, die auf einem Nachrichtenaustausch zwischen Objekten basiert. Für den Nachrichtenaustausch ist eine Referenz auf die Schnittstelle eines Objekts notwendig. Eine Referenz beschreibt eine Sicht auf das adressierte Objekt und kann hierdurch ebenfalls mit einem Typ assoziiert werden. Der Referenztyp muß nicht notwendigerweise mit dem Objekttyp übereinstimmen, da die durch eine Referenz definierte Sicht ihrerseits eine Abstraktion darstellt, die von gewissen Details des Objekttyps abstrahiert. Um eine *Zugriffssicherheit* auf die von einem Objekt angebotene Funktionalität zu gewährleisten, muß der Referenztyp in einem bestimmten Verhältnis zu dem Objekttyp stehen. Die *Typkonformität* definiert Kriterien, nach denen entschieden werden kann, wann zwei Typen konform zueinander sind, so daß die Zugriffssicherheit gewahrt bleibt.

Die von einer Typkonformität festgelegten Kriterien können nur anhand der extensionalen Beschreibungen von Typen überprüft werden. Die Typkonformität ist daher eingeschränkt auf die Ausdruckskraft einer Typbeschreibungssprache. Idealerweise sollte

Spezifikationsart	*Vorteile*	*Nachteile*
Syntaktisch	Einfache Spezifikationsart	Typkonformität auf einer semantischen Ebene kann nicht gewährleistet werden.
	Selektor–basierte Typkonformität ist entscheidbar	Verhaltensbeschreibung muß als externe Dokumentation beigefügt werden.
Semantisch	Hohe Wiederverwendbarkeit durch Abstraktion von Implementationsdetails.	Typkonformität i.a. unentscheidbar
	Abstrakte Spezifikation unterstützt Analyse des Problembereichs	Es ist unentscheidbar, ob ein Objekt Instanz eines Typs ist.
Pragmatisch	Ermöglicht in lokalen Systemen die Wiederverwendbarkeit der Implementation.	Typkonformität i.a. unentscheidbar
	Selektor–basierte Typkonformität einfach entscheidbar	Implementation eignet sich nicht als Dokumentation des Verhaltens.

Tabelle 2.1: Vor– und Nachteile verschiedener Abstraktionsebenen von Typbeschreibungssprachen.

die Typkonformität entscheidbar sein, so daß sie vom System automatisch überprüfbar ist. Eine entscheidbare Variante der Typkonformität liegt beispielsweise vor, wenn der Typ einer Referenz mit dem referenzierten Objekttyp übereinstimmen soll. Danach sind zwei Typen konform zueinander genau dann, wenn deren Repräsentationen in einer Typbeschreibungssprache syntaktisch äquivalent sind.

Ein Ausdruck beschreibt die Anwendung einer Nachricht auf ein Objekt mit einer operationalen Schnittstelle. Die Zugriffssicherheit bezieht sich in diesem Fall auf die Anwendbarkeit der Nachricht auf das adressierte Objekt. Die Zugriffssicherheit ist gewahrt, wenn eine Methodenauswahl möglich ist, d.h. es existiert eine eindeutige Zuordnung zwischen der Nachricht und einer Methode. Frühere Programmiersprachen zeichneten sich durch eine enge Kopplung zwischen Signaturen einer Nachricht und der ausgewählten Methode aus. Die daraus resultierende *monomorphe Typkonformität* erzwingt eine Übereinstimmung von aktuellen und formalen Parametertypen. „Konform“ ist in diesem Fall gleichzusetzen mit „syntaktisch übereinstimmend“.

Diese strenge Definition kann durch eine *polymorphe Typkonformität* flexibilisiert werden. Sie zeichnet sich dadurch aus, daß Ausdrücke, die bei einer monomorphen Typkonformität nicht korrekt im Sinne der Zugriffssicherheit sind, bei einer polymorphen korrekt sein können. Dies kann nur erreicht werden, wenn die Laufzeitumgebung Mechanismen anbietet, die beispielsweise durch Typkonvertierungen die Typsicherheit herstellen und dadurch den Polymorphismus auflösen.

Je nach Einsatzzeitpunkt dieser Mechanismen wird zwischen einer *statischen* und einer *dynamischen* Typkonformität unterschieden. Ein Typsystem besitzt eine statische Typkonformität, falls die Korrektheit von Ausdrücken zur Übersetzungszeit einer Anwendung festgestellt werden kann. Bei der dynamischen Typkonformität kann die Überprüfung

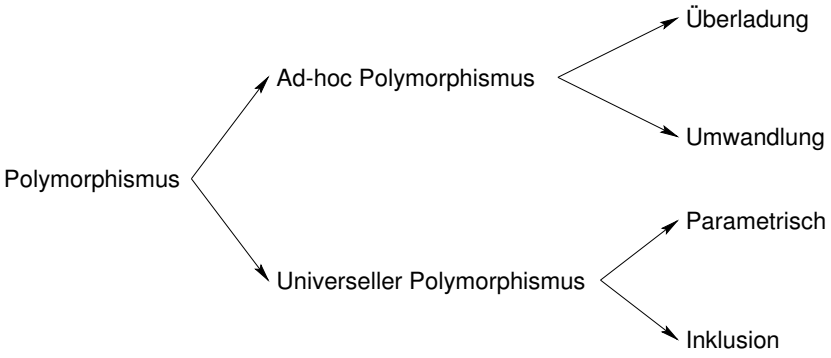


Abbildung 2.5: Kategorisierung des Polymorphismus.

erst zur Laufzeit geschehen. Jedes Typsystem mit einer monomorphen Typkonformität ist demnach auch immer statisch; die Umkehrung gilt i.a. jedoch nicht (z.B. besitzt das Typsystem der Programmiersprache C++ eine polymorphe, aber statische Typkonformität).

Nach [18] kann der Polymorphismus in *ad–hoc Polymorphismus* und *universeller Polymorphismus* unterteilt werden (siehe Abbildung 2.5). Der ad–hoc Polymorphismus definiert nur für eine endliche Menge von Ausdrücken eine polymorphe Beziehung zwischen Objekttyp und Referenztyp, während beim universellen Polymorphismus diese Menge unendlich sein kann. Der ad–hoc Polymorphismus unterteilt sich weiterhin in *Überladung* (englisch: *Overloading*) und *Umwandlung* (englisch: *Coercion*). Bei der Überladung ergibt sich der Polymorphismus daraus, daß für unterschiedliche Kombinationen von Parametertypen unterschiedliche Signaturen erzeugt werden, jedoch bei gleichen Funktionsnamen. Bei der Umwandlung werden die Parameter zunächst auf einen einheitlichen Typ konvertiert. Zur Verdeutlichung seien folgende vier Ausdrücke gegeben:

$$
\begin{aligned}
&calc.div(3, 6, r)\\
&calc.div(3.0, 6, r)\\
&calc.div(3, 6.0, r)\\
&calc.div(3.0, 6.0, r)
\end{aligned}
$$

Der Polymorphismus ergibt sich aus der homogenen Anwendung des Funktionsnamens *div* auf eine unterschiedliche (jedoch endliche) Menge an Parametertypen. Referenziert der Bezeichner *calc* in allen vier Ausdrücken das gleiche Objekt, so läßt dies zwei Interpretationen zu. Entweder besitzt das referenzierte Objekt vier Methoden mit dem Funktionsnamen *div* mit allen möglichen Permutationen von `int` und `real` als formalen Eingabeparametertypen (Überladung) oder es existiert nur eine Version der Methode mit der Signatur $div(\texttt{real}, \texttt{real}, \texttt{real})$. Im letzten Fall werden die Eingabeparameter vor Anwendung der Nachricht immer auf den allgemeineren Typ[3] `real` konvertiert (Umwandlung). Der ad–hoc Polymorphismus ist in Programmiersprachen häufig zu finden (C++ bietet sie durch die Konzepte des *Function Overloadings* bzw. *Type Castings* an, siehe [101]).

[3]Mit „allgemeiner“ ist der Typ mit dem größeren Zustandsraum gemeint.

Universeller Polymorphismus		**Ad–hoc Polymorphismus**	
Parametrisch	*Inklusion*	*Überladung*	*Umwandlung*
Uniforme Syntax von Ausdrücken	Uniforme Syntax von Ausdrücken	Polyforme Syntax von Ausdrücken	Polyforme Syntax von Ausdrücken
Uniforme Semantik	Uni–/polyforme Semantik	Polyforme Semantik	Polyforme Semantik
Generische Spezifikation durch parametrisierten Typ.	Abgeleitete Spezifikationen durch Vererbung.	Individuelle Spezifikation für jeden Typ.	Für verschiedene Typen genügt eine Spezifikation.
Objekte arbeiten homogen auf mehrere Typen.	Ein Objekt besitzt mehrere Typen.	Für jeden Kontext wird eine eigene Methode definiert.	Nachrichten werden an den Kontext angepaßt.

Tabelle 2.2: Eigenschaften der Polymorphismusarten.

Beide Unterformen des ad–hoc Polymorphismus setzen eine Systemunterstützung voraus (bei der Überladung durch eine korrekte Zuordnung einer Nachricht zu einer Methode (d.h. Methodenauswahl) und bei der Umwandlung durch die Anpassung der aktuellen Parameter).

Eine Typkonformität auf Basis des universellen Polymorphismus bietet Konzepte, die auf eine *a priori* unbekannte Anzahl von Ausdrücken anwendbar sind. Der *parametrische Polymorphismus* basiert auf generischen Typspezifikationen, die mit unterschiedlichen konkreten Parametertypen instantiiert werden können. Die Funktionalität ist unabhängig vom Parametertyp. Ein Beispiel für diese Form des Polymorphismus ist eine verkettete Liste, die mit einem Elementtyp parametrisiert werden muß. Ohne spezielle Kenntnis dieses Parametertyps können dennoch Listenoperationen wie Einfügen und Löschen von Listenelementen implementiert werden.

Der *Inklusions–Polymorphismus* basiert auf dem Austauschbarkeitsprinzip. Ein Typ A ist konform zu einem Typ B, wenn in jedem Ausdruck, in dem ein Objekt vom Typ B referenziert wird, dieses durch ein Objekt vom Typ A ausgetauscht werden kann. Die Typkonformität auf Basis des Inklusions–Polymorphismus muß nicht symmetrisch sein. Ist ein Typ A konform zu einem Typ B, so gilt die Umkehrung nicht notwendigerweise. Die Definition des Inklusions–Polymorphismus läßt jedoch den Schluß zu, daß die Konformität transitiv ist. Der ad–hoc Polymorphismus läßt nur zwischen einer endlichen Anzahl von Typen eine polymorphe Beziehung zu. Der Inklusions–Polymorphismus erlaubt dagegen wegen der Transitivität der Typkonformität eine polymorphe Beziehung zwischen einer nicht *a priori* festgelegten Menge von Typen. Tabelle 2.2 faßt die Eigenschaften der vorgestellten Polymorphismusarten zusammen.

2.4 Zusammenfassung

Das Objektmodell beruht auf den Konzepten der Abstraktion, der Modularität, der Kapselung und der Hierarchie. Diese Konzepte erlauben die Strukturierung eines Problembereichs in Form einer Menge kooperierender Objekte. Die Kapselung von Zustand und Verhalten macht Objekte zu Einheiten der Verteilung, die nur über wohldefinierte Schnittstellen von außen beeinflußbar sind. Ein Typ spezifiziert Eigenschaften, die, wenn sie auf ein Objekt zutreffen, dieses zu einer Instanz des Typs macht. Zu einem Typ ist hierdurch indirekt eine Menge von Objekten assoziiert, die über die gleichen Eigenschaften verfügen. Ein Typ besitzt eine Intension und eine Extension. Während die Intension eine abstrakte, nicht gegenständliche Charakterisierung darstellt, bildet jede konkrete Spezifikation basierend auf einer Notation eine Extension des Typs. Die Typkonformität, die über einer Typbeschreibungssprache definiert ist, garantiert eine Zugriffssicherheit auf die Schnittstelle eines Objekts. Je nach Flexibilität bei der Konformität zweier Typen wird zwischen monomorphen und polymorphen Typsystemen unterschieden. Ein polymorphes Typsystem stellt eine Erweiterung eines monomorphen Typsystems dar. Es existieren vier Polymorphismusarten, die unterschiedliche Unterstützung von einer Entwicklungs– und Laufzeitumgebung verlangen.

Kapitel 3

Dienstvermittlung in offenen verteilten Systemen

Während bis jetzt die Klassifikation von Typsystemen im Vordergrund stand, soll in diesem Kapitel auf deren Eigenschaften und Anforderungen im Kontext offener verteilter Systeme eingegangen werden. Der in Abschnitt 3.1 eingeführte Begriff eines offenen verteilten Systems charakterisiert Umgebungen, in denen Rechner über ein weitverzweigtes Kommunikationsnetzwerk miteinander verbunden sind. In Abschnitt 3.2 werden die Konzepte des im letzten Kapitel vorgestellten Objektmodells auf ein offenes verteiltes System übertragen. Die in einem offenen verteilten System existierenden Objekte treten in den Rollen von Dienstnutzern und Dienstanbietern auf. Die räumliche und zeitliche Entkopplung zwischen diesen erfordert anwendungsunabhängige Mechanismen, die die Kooperation zwischen Objekten unterstützen, wie die in Abschnitt 3.3 vorgestellte Dienstvermittlung.

Die Definition eines Typsystems ist eng verknüpft mit den Eigenschaften und Anforderungen der Dienstvermittlung in offenen verteilten Systemen. Daher werden in Abschnitt 3.4 zunächst Eigenschaften von Typsystemen, die für diese Umgebungen geeignet sind, vorgestellt. Da für ein offenes System kein spezielles Typsystem vorgeschrieben werden kann, entstehen Typdomänen, in denen unterschiedliche Typsysteme vorherrschen. Abschnitt 3.5 geht auf die Konsequenzen bei der transparenten Überbrückung verschiedener Typdomänen ein und in welchem Verhältnis diese zur Dienstvermittlung stehen. Aus den Eigenschaften offener verteilter Systeme leiten sich eine Reihe von Anforderungen ab, die an Typsysteme gestellt werden. Die in Abschnitt 3.6 hergeleiteten Anforderungen resultieren in Kriterien, nach denen ein Typsystem beurteilt wird. Die Anforderungen an Typsysteme besitzen Parallelen in anderen Forschungsgebieten, auf die exemplarisch in Abschnitt 3.7 eingegangen wird.

3.1 Offene verteilte Systeme

Ziel dieses Abschnitts ist die Definition eines *offenen verteilten Systems*. Ausgangspunkt bildet ein verteiltes System, das unter zusätzlichen Bedingungen zu einem offenen System wird. Ein verteiltes System charakterisiert eine Art von informationstechnischen Systemen, deren Abgrenzung zu den im Gegensatz stehenden zentralisierten Systemen fließend

sind und sich einer allgemeingültigen Definition entziehen (siehe [34]). Die in der Literatur zu findenden Definitionen beruhen auf subjektiven Kriterien, die eine eindeutige Kategorisierung eines informationstechnischen Systems nicht zulassen (siehe [55]). In [103] wird ein verteiltes System beispielsweise als eine Menge von Prozessoren aufgefaßt, die miteinander verbunden sind und zusammenarbeiten. Diese Definition ist zu allgemein, da ein Mehrprozessorsystem die Definition bereits erfüllt, wobei dieses eher den zentralisierten Systemen zugeordnet wird (siehe [73]).

Aus der Literatur sind mehrere Definitionen eines verteilten Systems bekannt (siehe [49], [51], [34] oder [25]). In Anlehnung an [10] soll ein verteiltes System wie folgt charakterisiert werden:

> Ein *verteiltes System* ist ein informationsverarbeitendes System, das eine Vielzahl von eigenständigen Rechnern enthält, die über ein Kommunikationsnetzwerk miteinander kooperieren, um ein angestrebtes Ziel zu erreichen.

Diese Definition unterscheidet sich zu jener aus [103] in drei Aspekten. Die elementare Einheit eines verteilten Systems ist ein Rechner und nicht mehr ein einzelner Prozessor und weist somit eine gröbere Granularität auf. Als Konsequenz daraus sind die einzelnen Rechner mit einem Netzwerk verbunden und nicht mehr über eine beliebige Verbindung. Zuletzt unterstreicht die Definition die Autonomie der Rechner, die zu einem gewissen Grad eigenständig sein müssen.

Verteilte Systeme zeichnen sich durch eine Vielzahl von Eigenschaften aus, die im Vergleich zu zentralisierten Systemen besonderer Unterstützung bedürfen. Für den Rahmen dieser Arbeit spielen einige dieser Eigenschaften eine besondere Rolle, da sie in engem Zusammenhang zu den im letzten Kapitel eingeführten Typsystemen stehen. Dazu gehören die *Entkopplung*, die *Heterogenität* und die Bildung von *Domänen*.

Entkopplung: Die zu einem verteilten System zusammengeschlossenen Rechner sind räumlich und zeitlich entkoppelt. Die räumliche Entkopplung ergibt sich aus der geographischen Ausdehnung des Systems. Die zeitliche Entkopplung entsteht aus der Autonomie der Rechner, die unabhängig von äußeren Einflüssen bestimmen, wann sie an dem verteilten System partizipieren wollen.

Heterogenität: Verschiedene Anwendungen, die in einem verteilten System realisiert werden, haben unterschiedliche Anforderungen hinsichtlich der Eigenschaften der zugrundeliegenden Hardware und der eingesetzten Programmiersprachen. Beispielsweise eignen sich einige Programmiersprachen nur für bestimmte Problembereiche. Um den unterschiedlichen Anforderungen gerecht zu werden, kann ein verteiltes System nicht homogen in seiner technischen Umsetzung sein.

Domänen: In einem verteilten System bilden sich Domänen aufgrund unterschiedlicher Interessen der Teilnehmer. Die Interessen beziehen sich nicht nur auf technologische Aspekte, sondern können auch abstrakte Interessen — wie beispielsweise unterschiedliche marktorientierte Ziele — umfassen. Weitere Beispiele sind administrative Domänen mit abweichenden Managementstrategien oder Typdomänen, in denen unterschiedliche Typsysteme zum Einsatz kommen (siehe Tabelle 3.1).

Domänenart	Erläuterung
Typdomäne	Die Domäne ist definiert durch eine Menge von Objekten, die einem gleichen Typsystem angehören.
Technische Domäne	Die entsprechenden Domänen grenzen Bereiche voneinander ab, in denen bestimmte Technologien vorherrschen.
Administrative Domäne	Innerhalb einer administrativen Domäne gilt eine festgelegte Managementstrategie.
Marktorientierte Domäne	Eine Menge von Objekten einer marktorientierten Domäne optimieren ihre Kooperation nach marktwirtschaftlichen Kriterien.

Tabelle 3.1: Definition unterschiedlicher Domänen.

Das geordnete Zusammenwirken der Anwendungen in einem verteilten System setzt standardisierte Schnittstellen und Protokolle zwischen diesen voraus. Die *International Organization for Standardization* (ISO) hat mit ihrem Standard für *Open System Interconnection* (OSI) ein 7–Schichten–Modell geschaffen, welches die Ende–zu–Ende–Kommunikation zwischen Anwendungen standardisiert (siehe [62]). Die ISO betont mit diesem Standard die operationalen Aspekte, die für ein *offenes verteiltes System* notwendig ist, dieses jedoch nicht ausreichend charakterisiert. Ein offenes verteiltes System zeichnet sich durch weitere Eigenschaften wie *Vielfalt*, *Dynamik*, *Austauschbarkeit* und *Skalierbarkeit* aus. Die Vielfalt in einem offenen verteilten System ergibt sich nicht nur aus der Quantität der dort vorzufindenen Anwendungen, sondern auch aus deren Qualität. Die Vielfalt ist hierdurch Ausdruck der Innovationsfähigkeit in einem offenen verteilten System.

Die Dynamik als weiteres Charakteristikum eines offenen verteilten Systems spiegelt die Autonomie seiner Teilnehmer wider. Nicht das System, sondern die Anwendungen entscheiden unabhängig, wann sie Teil des Systems sein wollen. Die sich daraus ergebende Fluktuation bedingt die Notwendigkeit der Austauschbarkeit von Anwendungen. Verschiedene Anwendungen sind nicht statisch aneinander gebunden, da sich deren Kopplung erst zur Laufzeit entscheidet. Damit werden die Anwendungen einem potentiellen Wettbewerb ausgesetzt, was Grundlage für marktwirtschaftliche Prinzipien ist. Weiterhin zeichnet sich ein offenes verteiltes System durch eine Skalierbarkeit seiner Komponenten aus, die weder den quantitativen noch den qualitativen Zuwachs eingeschränkt.

Die von einer Anwendung erbrachte Funktionalität wird als *Dienst* (englisch: *Service*) bezeichnet (siehe [43]). Eine Anwendung kann in der Rolle eines *Dienstnutzers* (englisch: *Service Requester*) bzw. eines *Dienstanbieters* (englisch: *Service Provider*) auftreten, je nachdem ob sie einen Dienst anbietet oder in Anspruch nimmt. Diese Rollenverteilung ist an das Client/Server–Modell angelehnt (siehe [26]). Im Client/Server–Modell stehen operationale Aspekte wie Replikation, Ausfallsicherheit oder Migration im Vordergrund. Die hier gewählte Terminologie soll eine Abstraktion dieser operationalen Aspekte andeuten, die das Management von Diensten hervorhebt.

Der Begriff der *elektronischen Dienstlandschaft* ist eine subjektive Interpretation der

Abläufe in einem offenen verteilten System. Wegen der geographischen Ausdehnung eines offenen verteilten Systems, das mehrere administrative und marktorientierte Domänen umfaßt, erhöht sich das Dienstangebot nicht nur quantitativ, sondern wegen einer zunehmenden Spezialisierung auch qualitativ. Die Bedürfnisse einer Anwendung in der Rolle eines Dienstnutzers nehmen in dem gleichen Maße zu, so daß sie sich durch das lokale Angebot in der eigenen Domäne nicht befriedigen lassen. Die Konsequenz daraus ist, daß sich in einem offenen verteilten System die Interaktion zwischen Dienstnutzer und –anbieter typischerweise über die Grenzen einer Domäne ausstrecken.

3.2 Das Objektmodell und offene verteilte Systeme

Die Konzepte des Objektmodells spiegeln in geeigneter Weise die Eigenschaften offener verteilter Systeme wider. Ein Objekt stellt eine Abstraktion einer Anwendung dar. Den Dienst, den eine Anwendung erbringt, wird an der Schnittstelle des Objekts, welches die Anwendung modelliert, angeboten. Ein Objekt kann in der Rolle eines Dienstnutzers bzw. eines Dienstanbieters auftreten. Objekte sind Einheiten der Verteilung, die über einen Nachrichtenaustausch die räumliche Entkopplung überwinden. Die dynamische Zuordnung von Objekten in Form von Referenzen unterstützt zudem die zeitliche Entkopplung. Erst zum Zeitpunkt der Zuordnung wird entschieden, welche Objekte im System für die Zuordnung in Frage kommen. Ein Objekt kapselt Zustand und Verhalten und ist nur über eine wohldefinierte Schnittstelle zugänglich. Die Schnittstelle verbirgt implementationsspezifische Details und trägt zur Überbrückung der Heterogenität bei.

Die Abläufe in einem offenen verteilten System lassen sich als Folge von gerichteten Graphen, den *Objektgraphen*, darstellen (siehe [27]). Ein Graph dieser Folge repräsentiert einen Schnappschuß einer objektbasierten Berechnung. Die Knoten des Graphen stellen die zu diesem Zeitpunkt existierenden Objekte dar, und die Kanten spiegeln Referenzen zwischen den Objekten wider. Wenn zwischen Objekt O_1 und Objekt O_2 eine Referenz existiert, kann O_1 Nachrichten an die Schnittstelle von O_2 schicken. Eine Folge von Objektgraphen beschreibt somit den zeitlichen Strukturwandel der elektronischen Dienstlandschaft. Ein Objekt nimmt die Rolle eines Dienstnutzers ein, wenn von ihm eine Referenz ausgeht. Umgekehrt ist ein Objekt durch eine einlaufende Referenz als Dienstanbieter charakterisiert. Delegiert ein Dienstanbieter einen Auftrag, so tritt es zugleich in der Rolle eines Dienstnutzers auf.

In einem offenen verteilten System ist der Vernetzungsgrad der Objekte typischerweise gering. Insbesondere können sich Partitionen ausbilden, d.h. es existieren Objekte, die nicht über eine Folge von Referenzen verbunden sind. Die Ausbildung von Partitionen eines Objektgraphen deutet auf einen fundamentalen Unterschied zwischen lokalen und verteilten Systemen hin. Während in einer lokalen Umgebung alle bis auf eine ausgezeichnete Partition gelöscht werden (*Garbage Collection*), bedarf die Unterscheidung zwischen zu löschenden und nicht zu löschenden Objekten in einer verteilten Umgebung anderer Kriterien.

Die Autonomie der Rechner in einem verteilten System überträgt sich zu einer Autonomie der Dienstanbieter in offenen verteilten Systemen. Diese entscheiden unabhängig von äußeren Einflüssen, wann sie ihre Dienstleistungen potentiellen Nutzern zur Verfügung stellen. Die daraus resultierende hohe Fluktuation im Angebot von Diensten muß durch

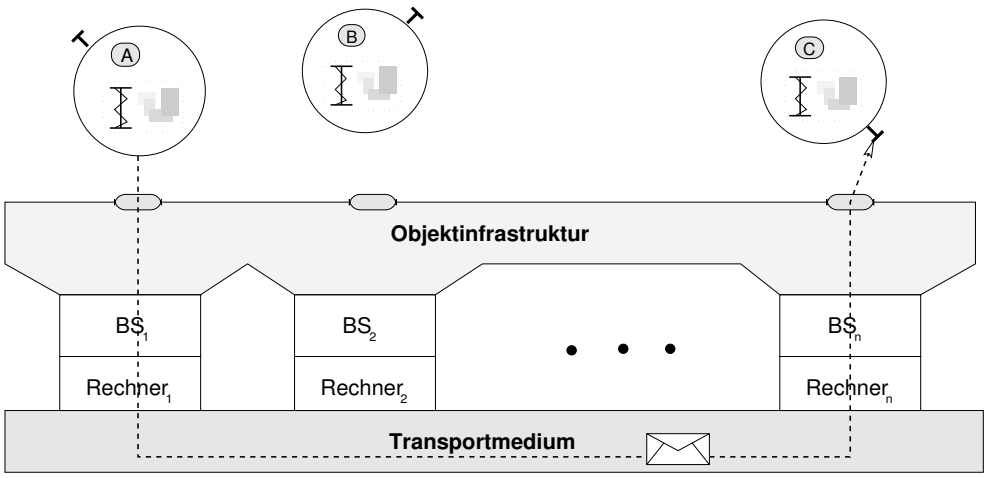


Abbildung 3.1: Infrastruktur für die Unterstützung des Objektmodells.

geeignete Mechanismen unterstützt werden. Um der Fluktuation und der räumlichen Entkopplung zu begegnen, darf die Bindung eines Dienstnutzers an einen gewünschten Anbieter nicht schon zur Übersetzungszeit geschehen, sondern erst zur Laufzeit.

Das Objektmodell wird in einem offenen verteilten System durch eine *Objektinfrastruktur* unterstützt. Eine Objektinfrastruktur dient als Plattform, die die Heterogenität eines Rechnernetzes verdeckt, sowie dessen physikalische Topologie. Konzeptionell bedient sich eine Objektinfrastruktur des Betriebssystems eines Rechners, um Objekten auf allen Rechnern eine identische Schnittstelle mit identischer Funktionalität anzubieten (siehe Abbildung 3.1). Auf jedem Rechner existiert ein wohldefinierter Zugangspunkt zu der Objektinfrastruktur, mit der anwendungsunabhängige Operationen (beispielsweise Methodenaufrufe oder Erzeugung von Objekten) den auf diesem Rechner befindlichen Objekten zugängig gemacht werden.

Anwendungsunabhängige Mechanismen, die eine Objektinfrastruktur anbietet, können auf zwei unterschiedlichen Ebenen angesiedelt sein. Bei einer Integration auf *Systemebene* werden die Mechanismen innerhalb der Objektinfrastruktur implementiert. Anstatt einer Ansiedelung der Mechanismen auf Systemebene können diese als dedizierte Objekte ausgelagert werden. In diesem Fall bieten ausgezeichnete Objekte die Mechanismen als Dienste an ihren Schnittstellen an. Diese Form der Integration findet somit auf der *Anwendungsebene* statt. Während die Einbettung der Mechanismen auf Systemebene effiziente Implementierungen dieser Mechanismen erlaubt, vermindert deren Auslagerung auf Anwendungsebene Systemabhängigkeiten und garantiert mehr Flexibilität sowie Austauschbarkeit der Mechanismen.

3.3 Dienstvermittlung

Die späte Bindung von Anbieter und Nachfrager zur Laufzeit bedingt eine dynamische Vergabe von Referenzen. Dienstnutzer und –anbieter werden nicht bereits zur Übersetzungszeit aneinander gebunden, da ein Dienstanbieter selbst über den Zeitpunkt entscheidet,

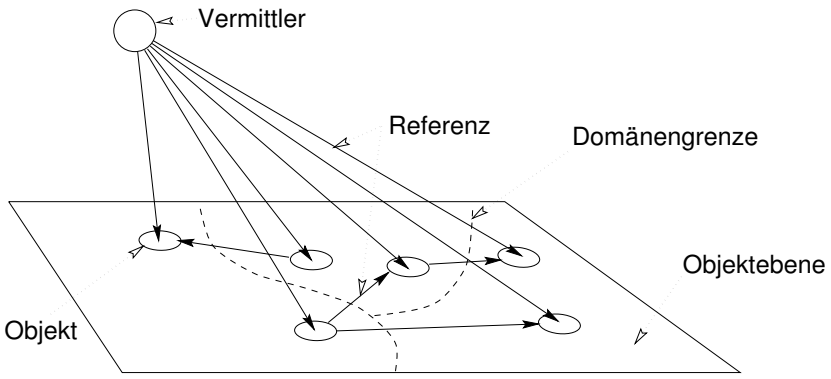


Abbildung 3.2: Vermittler als Verwalter von Objektreferenzen.

wann er an dem Dienstmarkt partizipieren möchte. Ein Dienstnutzer muß zum Zeitpunkt, da er das Bedürfnis nach einem bestimmten Dienst entwickelt, an einen passenden Anbieter vermittelt werden. Erst nach einer erfolgreichen Anbindung eines Dienstnutzers an einen Anbieter steht dem Dienstnutzer die Funktionalität des Anbieters zur Verfügung. Für die Zuordnung von Angebot und Nachfrage muß die Objektinfrastruktur eine Vermittlungsfunktionalität anbieten.

Die Einbettung der Vermittlungsfunktionalität auf Anwendungsebene weist einem Objekt die Rolle eines *Vermittlers* zu (im Englischen als *Trader* oder *Broker* bezeichnet). Dieses Objekt besitzt Kenntnis über das Dienstangebot eines offenen verteilten Systems (siehe Abbildung 3.2). Andere Objekte können die Funktionalität des Vermittlers nur dann in Anspruch nehmen, wenn sie eine Referenz auf dessen Schnittstelle besitzen. Da der Vermittler nicht seine eigenen Dienst vermitteln kann, müssen die Klienten des Vermittlers bereits bei ihrer Erzeugung eine Referenz auf diesen implizit zugewiesen bekommen. Der Vermittler wird hierdurch zu einem global *a priori* bekannten Objekt (englisch: *well–known object*) dessen Schnittstelle allen im System existierenden Objekten implizit bekannt ist.

3.3.1 Architektur und Funktionalität eines Dienstvermittlers

Die hier verwandte Terminologie stützt sich auf die von der ISO im Rahmen der ODP–Standardisierungen geprägten Begriffe (siehe [46]). Für einen Vermittlungsvorgang kooperieren drei Objekte in den Rollen des Vermittlers, des Dienstanbieters und des Dienstnutzers mit dem Ziel der Vermittlung eines Dienstes. Zunächst wendet sich der Dienstanbieter an den Vermittler mit der Aufforderung, seine Funktionalität innerhalb des offenen verteilten Systems potentiellen Dienstnutzern bekannt zu machen. Der Vermittler vermerkt das Angebot und berücksichtigt es fortan bei der Vermittlung. Im nächsten Schritt wendet sich ein Dienstnutzer an den Vermittler. In seiner Anfrage an den Vermittler teilt der Nutzer den konkreten Dienst mit, den er sucht. Findet der Vermittler einen passenden Dienst, den ein Anbieter zuvor registriert hat, beantwortet er die Anfrage des Nutzers mit einer Referenz auf die Schnittstelle des Anbieters. Der Vorgang der Bekanntmachung eines Dienstes durch einen Anbieter wird als *Export* bezeichnet und die Nachfrage eines Nutzers als *Import*.

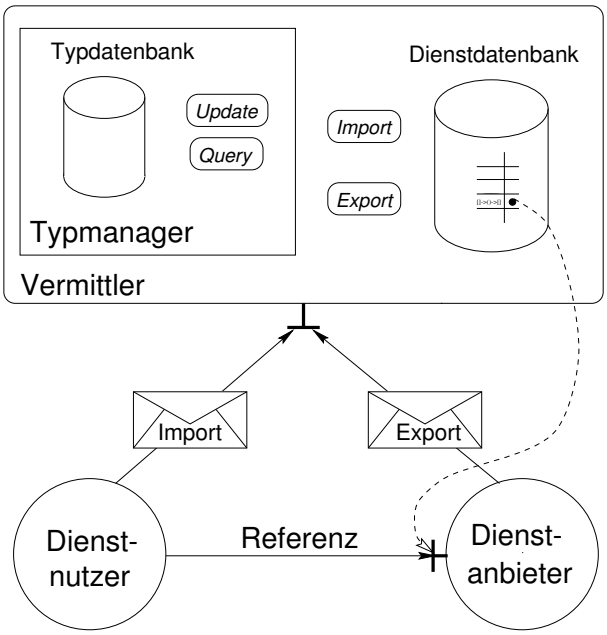


Abbildung 3.3: Allgemeine Architektur eines Vermittlers.

Die Bedeutung eines Typsystems für den Vermittler ist durch das Verhältnis der Begriffe Typ und Dienst gegeben. Ein Dienst charakterisiert eine Funktionalität, die an einer Schnittstelle eines Objekts erbracht wird. Die Beschreibung eines Dienstes nimmt keinen Bezug auf die Art der Implementierung des Dienstes, da diese für einen Dienstnutzer nicht relevant ist. Zu einem Dienst ist eine Menge von Objekten assoziiert, die an ihrer Schnittstelle diesen Dienst erbringen. Dadurch korrespondieren die Definition von Dienst und Typ, so daß im folgenden die Begriffe Dienst und Typ, bzw. Diensttypspezifikation und Typspezifikation, synonym verwendet werden.

Konzeptionell verwaltet der Vermittler eine Datenbank mit Dienstangeboten, die zuvor exportiert wurden. Der Vermittler speichert die Angebote in einer *Dienstdatenbank* (siehe Abbildung 3.3). Ein Eintrag in der Dienstdatenbank besteht aus einer Diensttypspezifikation und einer Menge von Referenzen auf Objektschnittstellen, an denen ein Dienst entsprechend der Diensttypspezifikation angeboten wird. Die Dienstdatenbank entspricht hierdurch einer 1–zu–n–Abbildung, die einen Diensttyp auf eine Menge von Referenzen abbildet.

Um einen *Single Point of Failure* zu vermeiden und um den Vermittlungsvorgang zu dezentralisieren, existieren typischerweise mehrere Vermittler, die jeweils einer Typdomäne zugeordnet sind. Eine Kooperation zwischen mehreren Vermittlern, mit der das vollständige Dienstangebot eines offenen verteilten Systems erreichbar ist, wird als *Federation of Traders* bezeichnet (siehe [11, 47, 107]). Die Menge der Vermittler treten auf einer übergeordneten Ebene („Meta–Ebene“) wiederum in den Rollen der Dienstnutzer und –anbieter untereinander auf, und es ist vorstellbar, daß die Kooperation zwischen den Vermittlern von einem „Meta–Vermittler“ unterstützt wird.

Die Zuordnung von Angebot und Nachfrage während der Dienstvermittlung ist von der Definition des zugrundeliegenden Typsystems abhängig. Sowohl die Export– als auch

die Import–Nachrichten enthalten als Parameter eine Typspezifikation auf Basis der Typbeschreibungssprache des Typsystems. Die Menge aller Dienste, die ein Vermittler verarbeiten kann, entspricht hierdurch dem Typraum des Typsystems. Die Export–Operation überprüft zunächst, ob die in der Nachricht enthaltene Typspezifikation gültig ist, d.h., ob diese Element des Typraums ist. Danach wird die Dienstdatenbank um das neue Dienstangebot erweitert. Existiert bereits ein Eintrag zu dem exportierten Typ, wird die Referenz in die Menge der zugehörigen Anbieter aufgenommen. Ansonsten wird ein neuer Eintrag in der Dienstdatenbank erzeugt.

Die Import–Operation vergleicht die Dienstnachfrage, die als Parameter in Form einer Typspezifikation in der Import–Nachricht enthalten ist, mit allen in der Dienstdatenbank registrierten Angeboten. Der Vergleich zweier Typen basiert auf der Typkonformität des zugrundeliegenden Typsystems. Ein Dienstanbieter erfüllt die Anforderungen eines Dienstnutzers genau dann, wenn der Typ des Anbieters konform zum gewünschten Typ des Nutzers ist. Die Treffermenge der möglichen Dienstanbieter ist definiert als die Menge aller Referenzen aus der Dienstdatenbank, deren Typ konform zu dem Typ der Anfrage ist. Die Art der Typkonformität steuert hierdurch indirekt die Größe der Treffermenge. Für monomorphe Typsysteme ist zu erwarten, daß die Treffermenge kleiner ist als bei einem polymorphen Typsystem. Eine quantitativ größere Treffermenge ist in offenen verteilten Systemen eine wünschenswerte Eigenschaft, da aus Sicht des Dienstnutzers die Menge der potentiellen Dienstanbieter und hiermit seiner Auswahlmöglichkeit größer ist.

Die Überprüfung der Typkonformität ist Aufgabe einer weiteren Komponente innerhalb des Vermittlers, dem *Typmanager* (siehe [39] und [84]). Der Typmanager ermittelt auf Anfrage, ob zwei Typspezifikationen konform zueinander sind. Dabei geht der Typmanager in zwei Schritten vor. Zunächst überprüft er, ob die beiden zu vergleichenden Typspezifikationen in seiner *Typdatenbank* als konform zueinander vermerkt sind. Liegt eine entsprechende Information nicht vor, versucht der Typmanager in einem zweiten Schritt, daß Ergebnis zu berechnen. Für eine automatische Herleitung der Typkonformität muß die Typbeschreibungssprache eine ausreichende Ausdruckskraft besitzen. Dies ist beispielsweise nicht der Fall, wenn die Extension eines Typs lediglich aus einem Schlüsselwort besteht. Anhand zweier unterschiedlicher Typspezifikationen in Form von Schlüsselwörtern kann nicht festgestellt werden, ob diese Typen konform zueinander sind. Kann der Typmanager die Typkonformität berechnen, so speichert er das Ergebnis in seiner Typdatenbank für zukünftige Anfragen ab. Über die Schnittstelle des Typmanagers kann darüberhinaus der Inhalt der Typdatenbank manuell gepflegt werden.

3.3.2 Vermittlung von Instanzen und Typen

Aus einer makroskopischen Perspektive stellt sich ein Vermittlungsvorgang im Objektgraphen als ein Hinzufügen einer Referenz dar. Der Vermittler, repräsentiert durch ein Objekt, besitzt Referenzen auf alle Dienstanbieter, die ihren Dienst zuvor exportiert haben. Ein Dienstnutzer, der sich durch eine Import–Operation an den Vermittler wendet, bekommt als Ergebnis seiner Anfrage eine Kopie einer Referenz vom Vermittler mitgeteilt. Nach einem erfolgreichen Vermittlungsvorgang besteht eine Referenz zwischen Dienstnutzer und –anbieter.

Die Referenzen können als eine Form von Beziehung zwischen Objekten angesehen

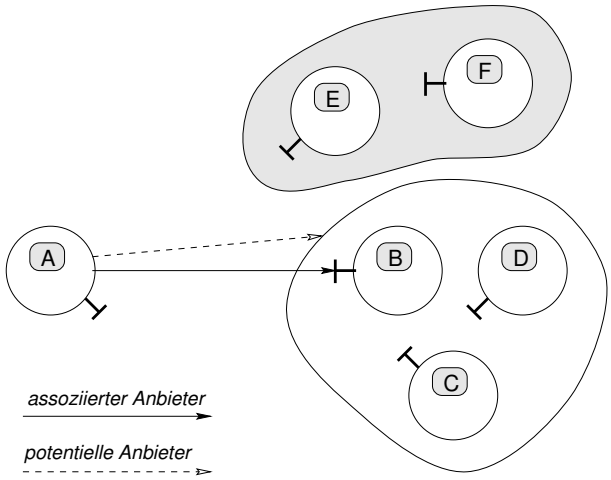


Abbildung 3.4: Assoziierte und potentielle Dienstanbieter.

werden, die im folgenden als *Erreichbarkeit* bezeichnet wird. Diese Beziehung, wenn sie zwischen zwei Objekten besteht, weist den Objekten die Rolle eines *Dienstnutzers* und eines *assoziierten Dienstanbieters* zu. Aus Sicht des Dienstnutzers ist das Ergebnis eines erfolgreichen Vermittlungsvorgangs die Erreichbarkeit eines ihm assoziierten Dienstanbieters. Voraussetzung für einen Import ist die Kenntnis einer Typspezifikation, da dieser als Parameter der Import–Nachricht enthalten sein muß. Der Typ, der das Bedürfnis einer bestimmten Funktionalität des Dienstnutzers verkörpert, muß bezüglich einer Typkonformität mit dem Typ des assoziierten Dienstanbieters übereinstimmten. Einem Typ kommt hierdurch die Rolle eines *Standards* zu, über den *a priori* in einem offenen verteilten System Konsens bestehen muß. Ein Dienstnutzer kann nur solche Dienste importieren, deren Typen ihm bekannt sind.

Die Bezeichnung *Closed World Assumption* (CWA) soll im folgenden die Annahme zum Ausdruck bringen, daß einem Dienstnutzer die Intension aller existierender Typen bekannt ist. Der Dienstnutzer besitzt hierdurch Kenntnis aller in einem Typraum existierenden Typspezifikationen und deren Interpretation. In einem offenen verteilten System ist es hingegen wahrscheinlicher, daß aufgrund der hohen Dynamik des Dienstmarkts einem Nutzer *nicht* alle ihn potentiell interessierenden Typspezifikationen bekannt sind. Diese Annahme soll entsprechend als *Open World Assumption* (OWA) bezeichnet werden[1].

Die Unterscheidung zwischen der CWA und der OWA soll durch eine weitere Beziehung zwischen Objekten modelliert werden. Damit ein Dienstnutzer sein Dienstbedürfnis in Form eines Typs formulieren kann, muß ihm der gewünschte Typ bekannt sein. Anders formuliert, sind ihm mit Kenntnis eines Typs eine Menge von *potentiellen Dienstanbietern* bekannt. Aufgabe der Vermittlung, so wie sie bis zu diesem Zeitpunkt vorgestellt wurde, ist es, aus der Menge der potentiellen Anbieter einen auszuwählen und diesen zu einem assoziierten Anbieter zu machen. Die Beziehung, die einen Dienstnutzer mit potentiellen

[1]Die Begriffe der *Closed World Assumption* und der *Open World Assumption* entstammen der Logik (siehe [59]). Dort bezeichnen die Begriffe unterschiedliche Interpretationen bzgl. der Negation.

Dienstanbietern verbindet, soll im folgenden *Sichtbarkeit* genannt werden. Abbildung 3.4 verdeutlicht die Unterscheidung zwischen der Erreichbarkeit und der Sichtbarkeit. Für Objekt A ist Objekt B ein assoziierter Dienstanbieter (Erreichbarkeit) und die Objekte B, C und D potentielle Dienstanbieter (Sichtbarkeit). Die Objekte E und F können nicht an Objekt A vermittelt werden, da Objekt A deren Typ nicht kennt und sie hierdurch nicht zu den potentiellen Anbieter gehören. Zusammenfassend sind die Erreichbarkeit und die Sichtbarkeit wie folgt charakterisiert:

Erreichbarkeit: Die Erreichbarkeit repräsentiert die Möglichkeit des Informationsflusses zwischen Objekten. Kann ein Objekt O_1 ein Objekt O_2 erreichen, so ist O_1 in der Lage, Nachrichten an die Schnittstelle von O_2 zu adressieren. Die Erreichbarkeit kann in Form eines Parameters einer Nachricht propagiert werden. Im mathematischen Sinne ist die Erreichbarkeit als Beziehung zwischen Objekten reflexiv (erlaubt die Kommunikation mit sich selbst) und symmetrisch. Besteht zwischen zwei Objekten eine Referenz, so stehen sie demnach in einer Erreichbarkeitsbeziehung.

Sichtbarkeit: Um mit einem Objekt kommunizieren zu können, muß neben seiner Schnittstelle auch sein Typ bekannt sein. Der Typ repräsentiert die äußere Sicht auf ein Objekt, und seine Kenntnis ist Voraussetzung für den korrekten Zugriff auf die wohldefinierte Schnittstelle eines Objekts. Ändert sich die Implementierung eines Objekts nicht, ist die Menge der ihm bekannten Typen fest. Die Sichtbarkeit von Typen kann somit nur in reflexiven Systemen (d.h. solche, die eine Änderung der Implementation zur Laufzeit unterstützen) propagiert werden. Die Sichtbarkeit ist wie die Erreichbarkeit reflexiv und gerichtet.

Die beiden Beziehungen sind nicht unabhängig voneinander. Damit ein Objekt mit einem anderen kommunizieren (d.h. es erreichen) kann, muß ihm der Typ des Zielobjekts bekannt (d.h. sichtbar) sein. Besitzt ein Objekt Kenntnis über einen Typ, so muß es jedoch nicht zwangsläufig auch eine Instanz dieses Typs erreichen können. Die Sichtbarkeit ist demnach eine Verallgemeinerung der Erreichbarkeit. Obwohl beide Beziehungsarten zwischen Objekten propagiert werden können, ist die Voraussetzung für diesen Vorgang die Erreichbarkeit.

Die Sichtbarkeit aller existierenden Typen, die in offenen verteilten Systemen nicht Voraussetzung ist, resultiert in einer Verfeinerung des Vermittlungsbegriffs. Der im letzten Abschnitt beschriebene Vermittlungsvorgang entspricht einer Vermittlung von Instanzen des Systems. In einem offenen verteilten System muß darüber hinaus noch eine Vermittlung von Typen gefordert werden, um der hohen Dynamik des Dienstmarkts gerecht zu werden. Der Vermittlungsvorgang bezieht sich in diesem Fall auf die Typen und nicht deren Instanzen. Um zwischen den beiden Arten der Vermittlung zu unterscheiden, werden diese im folgenden *Instanzenvermittlung* (oder einfach *Vermittlung*) und *Typvermittlung* (oder *Meta–Vermittlung*) genannt. Während die Instanzenvermittlung die Erreichbarkeit zwischen Objekten propagiert, leistet die Typvermittlung dies für die Sichtbarkeit.

3.4 Typsysteme für offene verteilte Systeme

Ein Typsystem nimmt eine zentrale Rolle bei der Vermittlung von Diensten in offenen verteilten Systemen ein. Dienstanbieter und –nutzer beschreiben Dienste in Form von Typspezifikationen, die auf einer Typbeschreibungssprache basieren. Der Dienstvermittler ordnet Angebot und Nachfrage gemäß der Definition einer Typkonformität zu. Während im letzten Abschnitt die Dienstvermittlung im Vordergrund stand, soll im folgenden auf die Eigenschaften von Typsystemen in offenen verteilten Systemen eingegangen werden. Abschnitt 3.4.1 diskutiert die Bedeutung syntaktischer Typbeschreibungssprachen für die Spezifikation operationaler Schnittstellen. Neben dem programmiersprachlichen Zugang zu Objektschnittstellen gewinnen benutzerorientierte Typsysteme zunehmend an Bedeutung. In Abschnitt 3.4.2 werden deren Eigenschaften vorgestellt.

3.4.1 Syntaktische Typsysteme

In offenen verteilten Systemen haben bestimmte Formen von syntaktischen Typsystemen weite Verbreitung gefunden. Deren Typbeschreibungssprache erlaubt die Spezifikation einer operationalen Schnittstelle eines Objekts. Eine operationale Definition einer Schnittstelle ist an das prozedurale Programmierparadigma angelehnt, welches den Nachrichtenaustausch zwischen Objekten weitestgehend transparent als Prozeduraufruf innerhalb einer Programmiersprache erscheinen läßt. Die Eingabeparameter eines entfernten Prozeduraufrufs werden in einer Anfragenachricht kodiert und an die Schnittstelle des bearbeitenden Objekts geschickt. Anschließend werden die Ergebnisse, nachdem sie über eine entsprechende Antwortnachricht dem Aufrufer mitgeteilt wurden, in die Ausgabeparameter übertragen. Dem Programmierer präsentiert sich die Interaktion mit einem anderen Objekt wie ein lokaler Prozeduraufruf.

Eine entsprechende Typbeschreibungssprache, auch *Schnittstellenbeschreibungssprache* (englisch: *Interface Definition Language*, IDL) genannt, erlaubt eine Spezifikation einer operationalen Schnittstelle in Form einer Menge von Prozeduren, mit jeweils einer endlichen Folge von Ein– und Ausgabeparametern. Die für eine Spezifikation verwandte Notation ist unabhängig von einer konkreten Programmiersprache. Es existieren zahlreiche Varianten unterschiedlicher Schnittstellenbeschreibungssprachen, von denen einige im nächsten Kapitel vorgestellt werden. Es ist zu bemerken, daß durch eine IDL lediglich die Menge von Signaturen definiert wird, jedoch die präzise Semantik der sie implementierenden Operationen verborgen bleibt.

Die nachfolgende operationale Schnittstellenbeschreibung spezifiziert einen hypothetischen Druckerdienst, der das Einfügen und Löschen von Druckaufträgen erlaubt (die Notation basiert auf der CORBA–IDL, siehe [78]):

```
interface PrintServiceInterface  {
    // Druckauftrag absetzen
    void SubmitJob (
      in  string Data,
      out long   JobNumber
    );
```

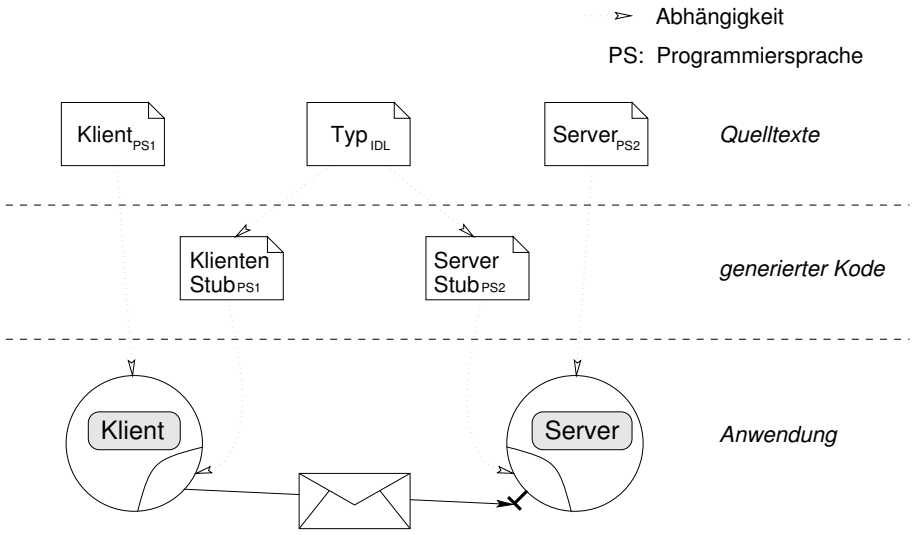


Abbildung 3.5: Stub–Generierung aus einer Schnittstellenbeschreibungssprache.

```
    // Druckauftrag entfernen
    void KillJob (
      in long JobNumber
    );
}
```

Ein Objekt, welches an seiner Schnittstelle einen solchen Dienst anbietet, akzeptiert zwei Nachrichten, die jeweils durch den Selektor `SubmitJob` und `KillJob` gekennzeichnet sind. Die erste Nachricht besitzt zwei typisierte Parameter: `Data` (Eingabeparameter vom Typ `string`) und `JobNumber` (Ausgabeparameter vom Typ `long`). Die zweite Nachricht besitzt lediglich den typisierten Eingabeparameter `JobNumber` vom Typ `long`. Ein– und Ausgabeparameter sind durch die CORBA–Schlüsselwörter `in` und `out` gekennzeichnet. Die informale Semantik des Diensttyps ist das Drucken von beliebigen Zeichenketten. Jedem Auftrag wird dabei eine eindeutige Nummer vergeben, mit Hilfe derer ein Auftrag zu einem späteren Zeitpunkt wieder annuliert werden kann.

Die Typbeschreibungssprache ist unabhängig von einer bestimmten Programmiersprache. Eine *Sprachanbindung* definiert eine Abbildung einer Typspezifikation auf Basis einer IDL an die Konventionen einer spezifischen Programmiersprache. Entsprechende Hilfswerkzeuge generieren aus einer Schnittstellenbeschreibung einen sogenannten *Stub* für eine Programmiersprache (siehe Abbildung 3.5). Ein Stub ist ein automatisch generierter Quelltext, der die operationale Schnittstelle eines Objekts gemäß den Konventionen einer Programmiersprache in Form einer Bibliothek anbietet. Aus der Typspezifikation werden zwei verschiedene Stubs erzeugt. Der Klienten–Stub ist dafür zuständig, Methodenaufrufe des aufrufenden Objekts in Nachrichten umzuwandeln, während der Server–Stub auf der Seite des aufgerufenen Objekts eine eintreffende Nachricht in einen Methodenaufruf umwandelt.

Typischerweise existieren für eine Schnittstellenbeschreibungssprache unterschiedliche Sprachanbindungen. Ein Stub–Generator kann parametergesteuert Schnittstellen für ver-

schiedene Programmiersprachen aus einer Schnittstellenbeschreibung erzeugen. Wie in Abbildung 3.5 dargestellt, können die Klienten– und Server–Stubs bzgl. einer operationalen Schnittstelle für unterschiedliche Programmiersprachen erzeugt werden. Die Typbeschreibungssprache eines syntaktischen Typsystems überwindet hierdurch die Sprachheterogenität. Eine operationale Schnittstelle verbirgt nicht nur die Implementierung eines Objekts, sondern auch die Programmiersprache, in der die Methoden des Objekts implementiert sind.

Eine Referenz, die ein Dienstnutzer auf einen Anbieter besitzt, repräsentiert eine Sicht auf den Anbieter. Sowohl Referenzen als auch Objekte sind über ein Typsystem typisiert. Die Typkonformität, die eine Zugriffsicherheit auf den Dienstanbieter gewährt, impliziert die Invariante, daß der Typ des Anbieters konform zu dem Referenztyp sein muß. Dies gewährleistet, daß der Dienstnutzer nur im Rahmen des Erlaubten auf den Dienstanbieter zugreift. Im Kontext der syntaktischen Typsysteme ist Zugriffsicherheit gleichzustellen mit der Garantie, bestimmte Methoden aufrufen zu können. Strengere Zusicherungen, die beispielsweise Bezug auf die Semantik der Methoden nehmen[2], sind bei einem syntaktischen Typsystem aufgrund der fehlenden Ausdruckskraft der Typbeschreibungssprache nicht möglich.

Die Definition der Typkonformität als Zusicherung der Verfügbarkeit bestimmter Methoden des Dienstanbieters ist ein Beispiel des im letzten Kapitel vorgestellten Inklusions–Polymorphismus. Beispielsweise ist die oben angegebene Schnittstellenbeschreibung `PrintServiceInterface` konform zu der folgenden Schnittstellenbeschreibung:

```
interface SimplePrintServiceInterface  {
    // Druckauftrag absetzen
    void SubmitJob (
      in  string Data,
      out long   JobNumber
    );
}
```

Besitzt ein Dienstnutzer eine Referenz vom Typ `SimplePrintServiceInterface` auf ein Objekt, dessen Typ `PrintServiceInterface` ist, ist die Zugriffsicherheit gewahrt, da alle Methoden, die der Dienstnutzer erwartet, vom Dienstanbieter implementiert werden. Der Zusammenhang zwischen Typkonformität und Zugriffsicherheit ist durch eine Teilmengenbeziehung der Methodenmengen der beiden Schnittstellenbeschreibungen gegeben. In einem syntaktischen Typsystem ist ein Typ T_1 konform zu einem Typ T_2, wenn die Menge der in Typ T_2 definierten Methoden eine Teilmenge der in Typ T_1 definierten Methoden bildet. Im nächsten Kapitel werden Erweiterungen dieser Definition vorgestellt, wie sie in bestehenden syntaktischen Typsystemen anzutreffen sind. Diese Definition erfüllt das Austauschbarkeitsprinzip des Inklusions–Polymorphismus und der Zugriffssicherheit.

Eine syntaktische Typbeschreibungssprache ermöglicht keine direkte Spezifikation der Semantik der Methoden. Die Bedeutung eines Diensttyps ist nur indirekt über die Wahl der Schlüsselwörter gegeben. Obwohl eine Person die Intension eines Diensttyps anhand

[2]Beispielsweise könnte eine Invariante eines *Stack*–Objekts lauten, daß vor dem Aufruf seiner `Pop`–Methode erst mindestens einmal die `Push`–Methode aufgerufen werden muß.

der Schlüsselwörter teilweise ableiten kann, kann ein Dienstvermittler nur anhand der in den Typen definierten Methoden die Typkonformität entscheiden. Die Konsequenz ist, daß bei einem syntaktischen Typsystem die Sichtbarkeit nicht propagiert werden kann. Eine Vermittlung von Typen ist wegen der fehlenden semantischen Spezifikation nicht möglich.

3.4.2 Benutzerorientierte Typsysteme

Die im letzten Abschnitt vorgestellten syntaktischen Typsysteme spielen in offenen verteilten Systemen eine wichtige Rolle. Sie ermöglichen eine Interoperabilität zwischen Objekten auf Basis des prozeduralen Programmierparadigmas. Die Vermittlung von operationalen Schnittstellenspezifikationen ist durch einen einfachen Vergleich der Methodenmengen möglich. Ein Dienstnutzer kann auf unterschiedlichen Abstraktionsebenen angesiedelt sein. Bisher wurde mit einem Dienstnutzer ein Objekt assoziiert, welches in einer bestimmten Programmiersprache implementiert ist. Unter einem Dienstnutzer kann aber auch auf einer abstrakteren Ebene ein *Anwender* gemeint sein. In diesem Abschnitt wird untersucht, inwiefern die Einbeziehung eines Anwenders in die Vermittlung von Diensten Auswirkungen auf die Anforderungen an Typsysteme für offene verteilte Systeme hat.

Anwender erlangen Zugang zu den in einem offenen verteilten System angebotenen Dienste über *generische Klienten*. Ein generischer Klient ist im Sinne des Objektmodells ein Objekt, welches in der Rolle eines Dienstnutzers auftritt. Die Implementierung eines generischen Klienten nimmt keinen Bezug auf eine spezielle Funktionalität eines Anbieters. Zur Übersetzungszeit eines generischen Klienten steht noch nicht fest, auf welche Dienste der Anwender, der den generischen Klienten steuert, zur Laufzeit zugreifen möchte. Ein generischer Klient greift ausschließlich über eine wohldefinierte, generische, operationale Schnittstelle auf die Funktionalität eines Dienstanbieters zu. Zur Laufzeit baut der generische Klient eine graphische Benutzeroberfläche auf, über die ein Anwender Aktionen auslösen kann (beispielsweise durch das Selektieren von Auswahllisten oder Eingabefelder). Die einzelnen Elemente der graphischen Benutzeroberfläche sind mit Methoden auf Seite des Dienstanbieters verknüpft, die bei Selektierung durch den Anwender aufgerufen werden. Beispiele für generische Klienten lassen sich im World–Wide Web, OLE2, OpenDoc oder COSM finden (siehe [14], [104], [19], [68]).

Die flexiblen Interaktionsmöglichkeiten von Anwendern mit *a priori* unbekannten Dienstanbietern über generische Klienten impliziert besondere Anforderungen an die Eigenschaften eines Typsystems. Ein Anwender entwickelt ein Bedürfnis nach einem bestimmten Diensttyp und muß hierdurch aktiv an der Dienstvermittlung partizipieren. Ein Typsystem, insbesondere dessen Typbeschreibungssprache, muß die Vorstellungswelt eines Anwenders unterstützen. Einem Anwender sind operationale Schnittstellenspezifikationen verborgen. Er beschreibt den von ihm gewünschten Dienst mit umgangssprachlichen Mitteln. Die Diensttypen zerfallen in zwei Gruppen, die *Systemtypen* und die *Benutzertypen*. Diese Unterscheidung spiegelt eine Klassifizierung einer differenzierten Dienstnutzerstruktur wider. Die Systemtypen basieren auf einem syntaktischen Typsystem und erlauben die Spezifikation operationaler Schnittstellen. Benutzertypen dahingegen repräsentieren abstraktere Spezifikationen, die im Unterschied zu Systemtypen im allgemeinen erst zur Laufzeit einer Anwendung entstehen (siehe Tabelle 3.2).

	Systemtypen	**Benutzertypen**
Zeitpunkt	Zur Übersetzungszeit.	Zur Laufzeit.
Abstraktionsniveau	Gering. Unterstützung fein–granularer Objekte aus *Programmierersicht*.	Hoch. Unterstützung von *Anwendern* auf einer kognitiven Ebene.
Detaillierungsgrad	Hoch. Strenge Definition von Objektschnittstellen auf Basis von Signaturen.	Gering. Basiert auf vagen und unvollständigen Informationen der Umwelt.
Typkonformität	Strenge deterministische Regeln.	Typbeschreibung müssen u.U. aufgrund unvollständiger Informationen verfeinert werden.
Unterstützte Objektarten	Feingranulare Objekte mit einer *a priori* bekannten operationalen Schnittstelle.	Generische Dienstnutzer, die mit *a priori* unbekannten Diensten arbeiten (WWW, OLE2, OpenDoc).
Typbeschreibungssprachen	CORBA–IDL, DCE–IDL.	Menge von Schlüsselwörtern.

Tabelle 3.2: Differenzierung zwischen Systemtypen und Benutzertypen.

Eine weitere Unterscheidung zwischen Systemtypen und Benutzertypen betrifft deren Kopplung zwischen Intension und Extension. In einer syntaktischen Typspezifikation ist die Semantik eines Diensttyps nur indirekt enthalten. Es muß globaler Konsens über die Bedeutung des Typs herrschen. Daraus resultiert eine strenge Eins–zu–Eins–Abbildung zwischen Extensionen und Intensionen. Bei einer Spezifikation eines Benutzertyps ist dagegen damit zu rechnen, daß zwei unterschiedliche Anwender abweichende Sichtweisen auf die Intension eines Typs haben. Das resultiert in unterschiedlichen Extensionen, die die gleiche Intension besitzen. Die Typkonformität zweier Typen wird anhand zweier Typspezifikationen entschieden. Besitzt eine Intension unterschiedliche Extensionen, so können diese nur als konform erkannt werden, wenn die zugrundeliegende Typbeschreibungssprache semantische Typspezifikationen erlaubt. Eine semantische Typbeschreibungssprache ermöglicht eine Erweiterung der Typkonformität, so daß die Konformität nicht nur die Struktur operationaler Schnittstellen berücksichtigt, sondern auch die mit der Intension eines Typs assoziierte Semantik.

Gerade ein Anwender benötigt einen Mechanismus, der ihn bei der in Abschnitt 3.3.2 eingeführten Vermittlung von Typen unterstützt. Da ein Anwender nicht über alle in einem offenen verteilten System vorkommenden Typen Kenntnis besitzen kann, muß ein Dienstvermittler Mechanismen anbieten, die ausgehend von vagen Vorstellungen des Anwenders, einen spezifischen Diensttyp ermitteln. Bei einer nicht–injektiven Interpretation, die mehreren Anwendern ihre eigene Sicht auf einen Diensttyp erlaubt, muß ein Vermittler die Konformität anhand semantischer Typspezifikationen überprüfen.

3.5 Überqueren von Typdomänen

In Abschnitt 3.1 wurde die Ausbildung von Domänen als ein wesentliches Charakteristikum offener verteilter Systeme vorgestellt. Eine Domäne definiert einen abgeschlossenen Bereich, in dem beispielsweise eine bestimmte Technologie oder Managementstrategie vorherrscht. Syntaktische Typsysteme erlauben die Überwindung technischer Domänen, indem verschiedene Sprachanbindungen Unterschiede in Datendarstellungen und physikalischen Gegebenheiten verbergen. Der Einflußbereich eines Typsystems bildet eine Domäne, genannt *Typdomäne*. Ein Objektgraph kann bezüglich einer Domänenart aufgeteilt werden (siehe Abbildung 3.2). Die Überwindung einer Domänengrenze verlangt eine besondere Unterstützung der zugrunde liegenden Objektinfrastruktur. Im folgenden wird eine Technik vorgestellt, mit der transparent die Grenze zwischen zwei Typdomänen überwunden werden kann. Daraus lassen sich weitere Anforderungen an Typsysteme für offene verteilte Systeme ableiten.

Wie bei der Vermittlungsfunktionalität kann die Unterstützung für die Überbrückung von Domänen auf System– oder auf Anwendungsebene angesiedelt sein. Aus Gründen der methodischen Darstellung auf Basis des Objektmodells soll die Überwindung von Typdomänen auf Anwendungsebene durch besondere Objekte geleistet werden. Die Objekte, die die Überwindung von zwei Typdomänen ermöglichen, heißen *Binder* (englisch: *Interceptor* oder *Bridge*). Binder sind spezielle Objekte, die durch ihre Rolle zwischen Dienstnutzer und –anbieter liegen. Sie leiten Methodenaufrufe eines Dienstnutzers nach einer Konvertierung gemäß den Konventionen der anderen Typdomäne transparent an den Dienstanbieter weiter. Konzeptionell befindet sich der Binder in einem Bereich, der Laufzeitunterstützung für beide Typsysteme bietet (siehe Abbildung 3.6). Der Bereich kann selbst verteilt sein und sich über Rechnergrenzen ausdehnen. Der Binder führt folgende Operationen aus (siehe [36]):

1. Er empfängt an seiner Schnittstelle in der einen Domäne eine Nachricht,
2. wandelt die in der Nachricht enthaltenen Parameter um und
3. leitet dann die Nachricht an das Zielobjekt in der anderen Domäne weiter.

Besondere Vorkehrungen müssen bei der Übergabe von Referenzen getroffen werden. Eine Referenz auf eine Objektschnittstelle kann als Parameter in einer Nachricht enthalten sein. Überquert die Nachricht die Grenzen einer Typdomäne, liegen Anfangs– und Endpunkt der Referenz in unterschiedlichen Domänen. Die Umwandlung des Referenztyps alleine reicht nicht aus. Da die Referenz die Grenzen einer Typdomäne überquert, muß für sie ein eigener Binder instantiiert werden, damit der Empfänger seinerseits über die Referenz Nachrichten verschicken kann. Je nach der Komplexität eines Binders kann zwischen folgenden Klassen von Bindern unterschieden werden (siehe [116]):

Statisch: Die Typen der Binderobjekte sind zur Laufzeit fest vorgegeben. Es können nur solche Referenzen eine Typdomäne überqueren, für deren Typ ein geeignetes Binderobjekt existiert.

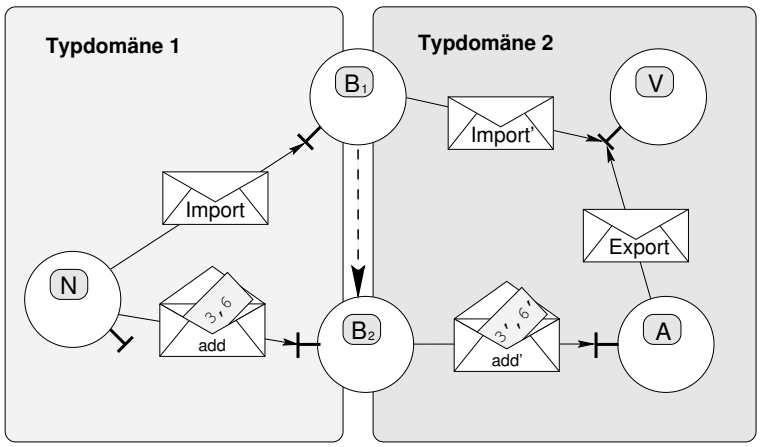


Abbildung 3.6: Überwindung von Typdomänen durch Binder.

Automatisch: Die Generierung eines Binders geschieht automatisch zur Laufzeit. Anhand von Abbildungsregeln für Parametertypen wird automatisch ein neuer Binder generiert, kompiliert und instantiiert.

Dynamisch: Ein dynamischer Binder benötig keine *a–priori*–Kenntnis eines Referenztyps. Die Parameterisierung des Binders erfolgt zur Laufzeit, ohne daß der Binder erst konstruiert werden muß.

Ein Vermittlungsvorgang, der sich über die Grenzen einer Typdomäne erstreckt, ist in Abbildung 3.6 skizziert. Zunächst exportiert der Dienstanbieter A in Typdomäne 2 seinen Dienst bei dem Vermittler V, der in seiner eigenen Typdomäne liegt. Da der Vermittler nach Definition ein *well–known object* ist, wird davon ausgegangen, daß der Dienstnutzer N eine Referenz auf diesen besitzt. Da diese Referenz jedoch die Grenzen der Typdomäne 1 überschreitet, muß ein Binder existieren, der die angrenzenden Typdomänen überbrückt. Diese Aufgabe übernimmt Binder B_1 in Abbildung 3.6. Für den Dienstnutzer N ist es transparent, daß er mit dem Binder B_1 kommuniziert. Ihm gegenüber verhält sich der Binder wie der Vermittler selbst.

Als Antwort auf seine Import–Operation erhält der Dienstnutzer N eine Referenz auf den Anbieter A. Während die Referenz über den Binder B_1 an den Dienstnutzer weitergegeben wird, instantiiert Binder B_1 einen neuen Binder B_2 (dargestellt durch den gestrichelten Pfeil in Abbildung 3.6) und reicht dem Dienstnutzer eine Referenz auf den neuen Binder zurück. Die Instantiierung des Binders B_2 geschieht entweder statisch, automatisch oder dynamisch. Der Binder B_2 präsentiert sich dem Dienstnutzer N wie der tatsächliche Anbieter. Gegenüber dem Anbieter A verhält sich der Binder B_2 als Dienstnutzer, indem er Nachrichten, die er von Objekt N erhält, nach einer Umwandlung an Objekt A weiterreicht.

Für den Kontext eines offenen verteilten Systems können somit drei Phasen bei der Kooperation zwischen einem Dienstnutzer und Diensterbringer identifiziert werden:

1. *Vermittlung*: Ein Dienstnutzer erfragt durch eine Import–Operation einen geeigneten Dienstanbieter, der sich zuvor mit einer Export–Operation bei einem Vermittler

registriert hat.

2. *Bindung*: Der Dienstnutzer erhält eine Referenz, über die er mit dem Dienstanbieter kommunizieren kann. Liegen Anbieter und Nutzer in unterschiedlichen Typdomänen, muß ein Binder zwischen diesen instantiiert werden.

3. *Interaktion*: Nach erfolgreicher Anbindung an einen Dienstanbieter kann der Dienstnutzer dessen Funktionalität durch Methodenaufrufe in Anspruch nehmen.

Obwohl die Vermittlung zeitlich vor der Bindung abläuft, hat die Art und Weise der Bindung zwischen Dienstnutzer und –anbieter Auswirkungen auf die Typkonformität. Ein Binder muß Methodenaufrufe eines Dienstnutzers, bevor er sie an einen Anbieter weiterreicht, an die Konventionen der anderen Typdomäne anpassen. Dies kann beispielsweise die Typkonvertierung der Parameter gemäß des Umwandlungs–Polymorphismus beinhalten. Die Umwandlung der Parameter muß bei der Vermittlung berücksichtigt werden. Entweder modifiziert der Binder, über den die Import–Operation weitergeleitet wird, die Typspezifikation entsprechend, oder der Vermittler hat Kenntnis von den Fähigkeiten des Binders. Zugunsten einer homogenen Darstellung der Dienstvermittlung soll davon ausgegangen werden, daß der Vermittler die von einem Binder vorgenommenen Umwandlungen kennt, und sie bei der Überprüfung der Typkonformität berücksichtigen kann.

3.6 Anforderungen an Typsysteme

Typsysteme und offene verteilte Systeme wurden bis zu diesem Zeitpunkt weitgehend getrennt betrachtet. Ein Typsystem spielt eine zentrale Rolle bei der Dienstvermittlung. Die Typbeschreibungssprache eines Typsystems erlaubt die Diensttypspezifikation, mit der Mengen von Objekten zu Typen zusammengefaßt werden. Die Typbeschreibungssprache ist das verbindende Glied zwischen Dienstnutzer und –anbieter. Gemeinsamer Konsens bei der Vermittlung von Diensten kann nur über eine gemeinsame Typbeschreibungssprache erreicht werden. Die Zuordnung von Angebot und Nachfrage innerhalb des Vermittlers basiert auf der Definition einer Typkonformität. Die Typkonformität garantiert eine Zugriffssicherheit, die notwendige Voraussetzung für die Kooperation zwischen Objekten ist.

Im folgenden sollen Anforderungen an Typsysteme in offenen verteilten Systemen formuliert werden. Wie bereits erläutert, bietet das Objektmodell eine geeignete Abstraktion für die Programmierung in offenen verteilten Systemen. Es unterstützt heterogene Umgebungen und bildet einen entfernten Zugriff auf ein Objekt transparent als Methodenaufruf ab. In einem offenen verteilten System muß auf einer operationalen Ebene die Kommunikation zwischen Objekten möglich sein. Die syntaktischen Typsysteme unterstützen durch entsprechende Hilfswerkzeuge den Umgang mit operationalen Schnittstellen. Die Möglichkeit der Spezifikation operationaler Schnittstellen stellt die erste Anforderung an ein Typsystem dar.

Bezüglich der Abstraktionsebenen von Typbeschreibungssprachen ist zu bemerken, daß pragmatische Typbeschreibungen für offene verteilte Systeme nicht geeignet sind. Bei pragmatischen Typbeschreibungen handelt es sich um ausführbare Beschreibungen,

die eher einer Implementation als einer Spezifikation gleichen. Damit eine pragmatische Typbeschreibung in einer heterogenen Umgebung ausführbar ist, muß auf jeder Plattform ein Interpreter verfügbar sein. Als Typbeschreibungssprache für offene verteilte Systeme besitzen pragmatische Typbeschreibungen deshalb mehrere Nachteile:

- Sie enthalten zu viele implementationstechnische Details und sind daher nicht geeignet als Abstraktion für Anwender.
- Ausführbare Beschreibungen, die auf beliebigen Plattformen ausführbar sind, stellen ein Sicherheitsproblem dar, da Programme sich potentiell unkontrolliert in einem Rechnernetz ausbreiten können.
- Dienstanbieter werden aus kommerziellen Gründen ihre Investitionen in Form der ausführbaren Typspezifikationen nicht offenlegen wollen. Pragmatische Typspezifikationen stellen Produkte dar, die erst mit Einführung eines Abrechnungssystems die Investitionen schützen.

Nach der Klassifikation aus Abschnitt 2.3.1 existieren neben syntaktischen und pragmatischen noch semantische Typbeschreibungssprachen. Diese sind für benutzerorientierte Typsysteme notwendig. Mit der Technik der generischen Klienten erlangen Anwender Zugang zum Dienstangebot eines offenen verteilten Systems. Das hat zur Konsequenz, daß sie zur Laufzeit ihre Bedürfnisse in Form von Typspezifikationen beschreiben müssen. Operationale Schnittstellen auf Basis syntaktischer Spezifikationen sind weder als Abstraktion für Anwender geeignet, noch können durch diese die in generischen Umgebungen vorzufindenen Dienste beschrieben werden. Eine Typspezifikation in einem offenen verteilten System sollte deshalb sowohl die Spezifikation operationaler Schnittstellen als auch inhaltsorientierte Dienstbeschreibungen erlauben, die die Domäne eines Anwenders unterstützen.

Unabhängig vom Abstraktionsniveau sollte eine Typbeschreibungssprache, die eine Spezifikation eines Dienstes erlaubt, entscheidbar sein (d.h. zu jeder Typspezifikation kann entschieden werden, ob sie Element des Typraums ist), um eine maschinelle Verarbeitung zu ermöglichen. Die Fluktuation in der Qualität und Quantität des Dienstangebots eines offenen verteilten Systems resultiert in der Forderung nach einer skalierbaren Typbeschreibungssprache. Eine skalierbare Typbeschreibungssprache besitzt eine ausreichende Ausdruckskraft, mit der alle existierenden Dienste hinreichend spezifizierbar sind. Die schwächste Bedingung, nach der eine Typbeschreibungssprache skalierbar ist, ist die Unbeschränkheit des Typraums. Ein beschränkter Typraum würde in einer endlichen Menge von spezifizierbaren Diensttypen resultieren.

Die Definition einer Typkonformität gehört neben einer Typbeschreibungssprache zu einem Typsystem. Sie garantiert eine Zugriffssicherheit, die eine fehlerfreie Kommunikation zwischen Dienstnutzer und –anbieter garantiert. Der Vermittler führt Dienstangebot und –nachfrage aufgrund der Typkonformität zusammen. Der Grad der Flexibilität bei der Überprüfung der Typkonformität steuert das quantitative Angebot aus Sicht eines Dienstnutzers. Ein polymorphes Typsystem flexibilisiert die Typkonformität im Vergleich zu der konservativen Definition bei den monomorphen Typsystemen. Der Polymorphismus gliedert sich in einen ad–hoc und einen universellen Polymorphismus auf (siehe Abschnitt 2.3.2). Der ad–hoc Polymorphismus definiert eine polymorphe Beziehung für eine

endliche Menge von Typen eines Typsystems. Aufgrund der nur marginal flexibleren Typkonformität im Vergleich zu der von monomorphen Typsystemen ist diese Form des Polymorphismus für offene verteilte Systeme ungeeignet. Der universelle Polymorphismus dehnt das polymorphe Verhältnis auf eine unbeschränkte Anzahl von Typen aus, was zu einer flexiblen Definition der Typkonformität führt. Aus Sicht eines Dienstnutzers ist bei dieser Form des Polymorphismus deshalb das quantitative Angebot von konformen Diensten in der Regel umfangreicher.

Der universelle Polymorphismus gliedert sich in den parametrischen und den Inklusions–Polymorphismus auf. Der parametrische Polymorphismus setzt eine pragmatische Typbeschreibungssprache voraus, was ihn für den Kontext offener verteilter Systeme ungeeignet macht. Das auf dem Inklusions–Polymorphismus basierende Austauschbarkeitsprinzip ist dagegen unabhängig von einer Typbeschreibungssprache allgemein einsetzbar. Die einzige in offenen verteilten Systeme praktikable Polymorphismusart ist hierdurch der Inklusions–Polymorphismus. Zur Flexibilisierung der Typkonformität eines Typsystems wird ein Inklusions–Polymorphismus gefordert.

Der Vermittler entscheidet aufgrund einer Typkonformität, ob ein Dienstanbieter die Forderungen bzgl. einer Funktionalität erfüllt oder nicht. Ein Typ übernimmt somit die Rolle eines Standards. Alle Parteien, d.h. Dienstnutzer und –anbieter, müssen sich *a priori* auf die Extensionen der Typen einigen. Es herrscht globaler Konsens über alle Typspezifikationen, und unter der *Closed World Assumption* ist jedem Teilnehmer des offenen verteilten Systems der Typraum vollständig bekannt. Gerade die Absprachen von Typspezifikationen laufen den Eigenschaften eines offenen verteilten Systems zuwider. Die hohe Fluktuation der Dienstangebote impliziert ebenfalls eine hohe Fluktuation der im Dienstmarkt bekannten Typen. Die Notwendigkeit der *a–priori*–Kenntnis der Typen hemmt die Dynamik des Systems. Jeder Dienstanbieter wäre hierdurch gezwungen, seinen Typ potentiellen Anbietern bekannt zu machen.

Die sich daraus ergebende Eigenschaft der *Open World Assumption*, die den Dienstanbieter von dieser Forderung befreit, resultiert in einen Dualismus in den Anforderungen an ein Typsystem für offene verteilte Systeme. Einerseits setzt die Dienstvermittlung Typen in Form von Standards voraus, um zweifelsfrei Angebot und Nachfrage zuzuordnen. Andererseits sollten *a–priori*–Absprachen zwischen den Teilnehmern über konkrete Typspezifikationen vermieden werden, um die Dynamik des Systems nicht zu beeinträchtigen. Grund für diesen Dualismus liegt in dem Verhältnis zwischen der Intension und der Extension eines Typs (siehe Abbildung 3.7). Die Typkonformität kann nur anhand der Extensionen zweier Typen entschieden werden. Ein Typ als Standard resultiert in einer eindeutigen Abbildung zwischen Extension und Intension, die mehrere Sichten (d.h. Extensionen) auf einen Typ unmöglich macht.

Unter der *Open World Assumption* hingegen kann eine Intension unterschiedliche Extensionen besitzen, die nicht notwendigerweise standardisiert sein müssen. Die verschiedenen Extensionen, die alle auf eine Intension abgebildet werden, spiegeln unterschiedliche Sichten auf einen Typ wider. Zwischen Dienstanbieter und –nutzer besteht nicht mehr Konsens über eine Extension, sondern jeder der beiden Beteiligten hat „seine“ Vorstellung darüber, wie eine Typspezifikation die Intension repräsentiert. Die Interpretation von Typen ist wegen der Freiheitsgrade bei der Wahl der Extension nicht–injektiv. Unterschiedliche Extensionen, die durch ihre Interpretation die gleiche Intension besitzten,

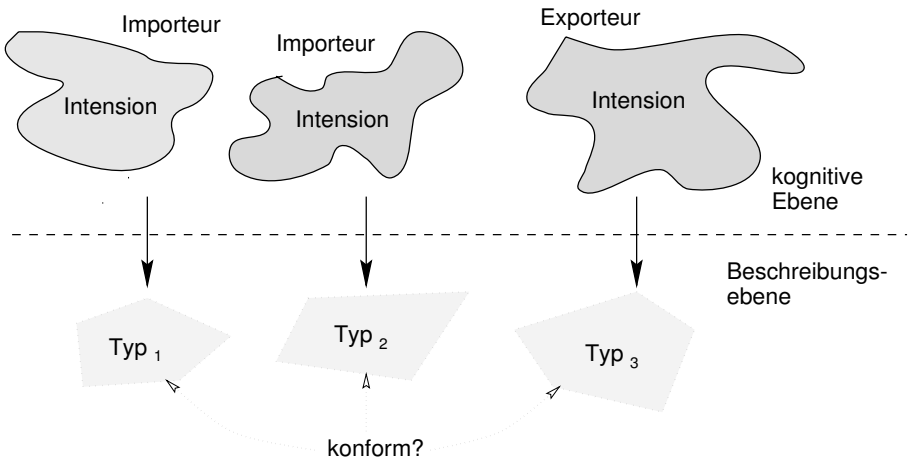


Abbildung 3.7: Problem der Typkonformität.

sind hierdurch typkonform. Um die Typkonformität anwenden zu können, muß eine Extension ausreichende Semantik beinhalten, so daß zwei Typspezifikationen als konform erkannt werden, wenn sie die gleiche Intension besitzen.

Zusammenfassend werden folgende Anforderungen an ein Typsystem für offene verteilte Systeme gestellt:

(TS_1) *Skalierbarkeit*: die Typbeschreibungssprache muß eine ausreichende Ausdruckskraft besitzen.

(TS_2) *Open World Assumption*: keine notwendige *a–priori*–Kenntnis des Typraums.

(TS_3) *Inklusions–Polymorphismus*: Flexibilisierung der Typkonformität.

(TS_4) *Entscheidbarkeit*: Entscheidbare Typbeschreibungssprache und Typkonformität.

(TS_5) *Abstraktion*: Unterstützung für unterschiedliche Benutzergruppen.

Die Bedingungen TS_1 bis TS_5 entsprechen Idealvorstellungen, die sich teilweise wechselseitig ausschließen. Beispielsweise bedingt die *Open World Assumption* eine semantische Typbeschreibungssprache, die nicht–injektive Interpretationen zwischen Extensionen und Intensionen zuläßt. Eine semantische Typbeschreibungssprache besitzt zwar eine höhere Ausdruckskraft, läuft aber Gefahr, unentscheidbar zu sein. Entscheidbarkeit ist jedoch Voraussetzung für eine Automatisierung der Dienstvermittlung. Bei der Konstruktion eines neuen Typsystems, wie beispielsweise das in einem späteren Kapitel vorgestellte deklarative oder wissensbasierte Typsystem, muß zugunsten von einer dieser Anforderungen ein Schwerpunkt gelegt werden.

3.7 *Open World Assumption* in anderen Gebieten

Der Zwiespalt in den Anforderungen an ein Typsystem für offene verteilte Systeme findet sich auch in anderen Forschungszweigen wieder. Die in diesem Kapitel eingeführten

Begriffe der Intension und der Extension, mit denen die Beschreibungsebene von der Vorstellungsebene getrennt wird, entstammen der Philosophie (siehe [8]). Grundlegende Arbeiten zu diesem Bereich finden sich in den Arbeiten des amerikanischen Philosophen Charles Sanders Peirce, der mit der *Semiotik* die Unterscheidung zwischen einem Zeichen und dem Ding, das es repräsentiert, eingeführt hat (siehe [94]). Im Bereich der Informatik sind die Begriffe Extension und Intension bei deduktiven Datenbanken zu finden, wo zwischen Fakten und hergeleitetem Wissen unterschieden wird (siehe [9]).

Eine Parallele zu dem Problem der Notwendigkeit der *a–priori*–Kenntnis existiert in der Kryptographie. Um eine geheime Nachricht zu verschlüsseln, müssen Sender und Empfänger der Nachricht sich auf einen geheimen Schlüssel einigen. Nur mit Kenntnis des Schlüssels kann auf den Inhalt einer Nachricht zugegriffen werden. Das Problem der *a–priori*–Kenntnis stellt sich hier in der Frage, wie sich Sender und Empfänger auf einen Schlüssel einigen. Der Austausch eines Schlüssels bedingt bereits eine gesicherte Übertragung, da kein Außenstehender den Schlüssel erfahren darf. Die gesicherte Übertragung kann aber erst garantiert werden, wenn beide Parteien sich auf einen Schlüssel geeinigt haben. Lösung zu diesem Problem bieten die *Public–Key*–Verschlüsselungsverfahren (wie beispielsweise der in [90] beschriebene RSA–Algorithmus) oder das 3–Wege–Protokoll (siehe [66]).

Das Schlüsselverteilungsproblem ist eine Spezialisierung der Probleme, die allgemein bei einem Kommunikationsvorgang auftreten. Ein Kommunikationsvorgang, der auf Basis einer Sprache Informationen vermittelt, setzt eine *a–priori*–Absprache der Bedeutung voraus. Diese Voraussetzung verleitete Bertrand Russel zu einer nach ihm benannten Paradoxie. Jede Kommunikation, die, um verstanden zu werden, ihre eigene Erklärung kommunizieren muß, erzeugt eine Selbstbezüglichkeit, die in einem Paradoxon mündet. Im Bezug auf die Logik findet die Russelsche Paradoxie eine Parallele in Gödels Unvollständigkeitssatz. Gödel bewies, daß kein System sich selbst voll erklären oder beweisen kann, ohne auf Begriffe zurückzugreifen, die es selbst nicht abzuleiten imstande ist.

Die Linguistik unterscheidet ebenfalls zwischen Intension und Extension. Ein Wort eines Vokabulars einer Sprache (Extension) besitzt eine Bedeutung (Intension), mit der es verknüpft ist (siehe [70]). Zwei Wörter, die die gleiche Bedeutung besitzen (d.h. auf die gleiche Intension abgebildet werden), heißen *Synonyme*. Umgekehrt kann ein Wort unterschiedliche Bedeutungen haben. Ein solches Wort heißt in diesem Fall *polysemisch*. Um einem Wort eine korrekte Bedeutung zuordnen zu können, muß dessen Kontext betrachtet werden, in das es eingebettet ist. Die Eigenschaft der Polysemie wird in einem späteren Kapitel Auswirkungen für benutzerorientierte Typsysteme haben, welche die *Open World Assumption* unterstützen.

3.8 Zusammenfassung

Innerhalb eines offenen verteilten Systems besteht eine Symbiose zwischen der Vermittlung von Diensten und Typsystemen. Die Typbeschreibungssprache erlaubt Typspezifikationen, die als Parameter von Import und Export–Operationen Dienste charakterisieren. Für den Kontext offener verteilter Systeme erlaubt die Spezifikation operationaler Schnittstellen eine Abstraktion, die den Zugriff auf ein entferntes Objekt als einen lokalen Methodenaufruf erscheinen lassen. Die Zuordnung zwischen Angebot und Nachfrage,

die Aufgabe eines Vermittlers ist, basiert auf der Definition einer Typkonformität. Aus Sicht eines Dienstnutzers erhöht ein polymorphes Typsystem das Dienstangebot, da es die Menge der potentiellen Dienstanbieter vergrößert.

Die Technik der generischen Klienten ermöglicht Anwendern den unmittelbaren Zugang zu dem Dienstangebot einer offenen Umgebung. Die Dienstvermittlung bindet die Anwender aktiv in die Dienstsuche mit ein. Zu diesem Zweck muß eine Typbeschreibungssprache die Abstraktionsebene von Anwendern unterstützen. Ferner muß jedem Anwender seine eigene Sicht auf die Intension eines Typs erlaubt werden. Die nicht–injektive Interpretation zwischen Extensionen und Intensionen verringert die Notwendigkeit von *a–priori*–Absprachen und fördert somit die Dynamik des Systems. Neben der Vermittlung von Instanzen muß ein Dienstvermittler Mechanismen anbieten, die die Vermittlung von Typen unterstützen.

Kapitel 4

Überblick über bestehende Typsysteme

Die späte Zuordnung zwischen Dienstnutzern und –anbietern bedingt die Notwendigkeit einer Vermittlungsfunktionalität in offenen verteilten Systemen. Zentraler Kern des Vermittlungsvorgangs ist ein Typsystem, welches die Spezifikation von Diensttypen erlaubt und eine Typkonformität definiert, die bestimmt, wann Angebot und Nachfrage zusammenpassen. Unterschiedliche Objektinfrastrukturen, die für offene verteilte Systeme entstanden sind, definieren ihre eigenen Typsysteme. Gegenstand dieses Kapitels ist es, einige ausgewählte, weit verbreitete Typsysteme vorzustellen und zu bewerten. Die Typsysteme werden hinsichtlich ihrer Typbeschreibungssprachen und der Definition ihrer Typkonformität vorgestellt. Auf eine ausführliche Beschreibung der zugrundeliegenden Objektinfrastrukturen wird verzichtet. In Abschnitt 4.1 werden zunächst drei unterschiedliche syntaktische Typsysteme und anschließend in Abschnitt 4.2 zwei semantische Typsysteme diskutiert.

4.1 Syntaktische Typsysteme

Syntaktische Typsysteme haben sich in Objektinfrastrukturen für offene verteilte Systeme etabliert. Eine syntaktishe Typspezifikation beschreibt das operationale Verhalten einer Objektschnittstelle. Eine operationale Schnittstelle ist charakterisiert als eine Menge von Nachrichten, die an der Schnittstelle akzeptiert werden. Werkzeuge, wie die in Abschnitt 3.4.1 erwähnten Stub–Generatoren, erlauben eine transparente Integration eines entfernten Objektzugriffs in eine Programmiersprache. Mit einer syntaktischen Typbeschreibungssprache kann die Heterogenität bzgl. unterschiedlicher Programmiersprachen und Datendarstellungen überwunden werden. Die Definition der Typkonformität von syntaktischen Typsystemen legt eine Zugriffssicherheit fest, die garantiert, daß ein Dienstanbieter alle Nachrichten akzeptiert, die ein zugeordneter Dienstnutzer an dessen Schnittstelle adressiert. Diese Form der Typkonformität ist entscheidbar und kann von einem Vermittler zur Laufzeit überprüft werden.

Aufgrund ihrer einfachen Handhabbarkeit haben syntaktische Typsysteme eine weite Verbreitung erfahren. Im folgenden werden drei verschiedene syntaktische Typsysteme vorgestellt: das der *Open Software Foundation* (OSF) (Abschnitt 4.1.1), der *Object Ma-*

nagement Group (OMG) (Abschnitt 4.1.2) und das aus dem *Reference Model for Open Distributed Processing* (RM–ODP) (Abschnitt 4.1.3). Die OSF und die OMG bezeichnen ihre Typbeschreibungssprache als *Schnittstellenbeschreibungssprache* (englisch: *Interface Definition Language*, IDL). Die ISO beschreibt lediglich Eigenschaften eines syntaktischen Typsystems, ohne jedoch eine konkrete Notation für eine Typbeschreibungssprache vorzuschreiben. Die Eigenschaften der drei Typsysteme werden in den folgenden Abschnitten diskutiert. Auf die Architektur der zugrundeliegenden Objektinfrastrukturen wird nur soweit eingegangen, wie es für das Verständnis der vorliegenden Arbeit notwendig ist. Vertiefende Informationen sind der Literatur zu entnehmen (siehe beispielsweise [26], [60] oder [95]).

4.1.1 Distributed Computing Environment

Auf Basis der von der OSF vorgegebenen Anforderungen an eine Infrastruktur für verteilte Systeme wurden nach einer offenen Ausschreibung aus den von Herstellern eingereichten Vorschlägen die Struktur und Funktion des *Distributed Computing Environment* (DCE) festgelegt (siehe [79]). DCE bietet Unterstützung für anwendungsunabhängige Problemstellungen, wie beispielsweise Kommunikation zwischen Objekten, Authentisierung und Autorisierung von Objekten, entfernter Dateizugriff oder Uhrensynchronisation. DCE ist inzwischen für eine Vielzahl von Hardware–Plattformen und Betriebssystemen verfügbar.

DCE unterstützt die Kommunikation zwischen Objekten in einem verteilten, heterogenen System nach dem Client/Server–Modell. Objekte nehmen die Rollen von *Klienten* oder *Servern* ein, je nachdem, ob sie Funktionalität anbieten oder in Anspruch nehmen. Ein Objekt kann sowohl eine Klienten– als auch eine Server–Rolle zu einem Zeitpunkt einnehmen. Die Kommunikation zwischen einem Klienten und einem Server erfolgt in DCE durch einen *Remote Procedure Call* (RPC), der gemäß dem prozeduralen Programmierparadigma einen verteilten Funktionsaufruf realisiert.

DCE besitzt eine eigene Schnittstellenbeschreibungssprache. Sie erlaubt die Spezifikation einer geordneten Menge von Funktionen, die durch einen Funktionsnamen und eine Folge von typisierten Ein– und Ausgabeparametern beschrieben sind. Die DCE–IDL definiert für die Typisierung der Parameter eine Menge von Basistypen (beispielsweise `long` oder `char`) sowie Konstruktionsvorschriften für zusammengesetzte Benutzertypen (beispielsweise `struct` oder `union`). Die dafür eingeführten Schlüsselwörter und Schreibweisen sind an das Typsystem der Programmiersprache C angelehnt. DCE bietet Werkzeuge an, die aus einer Schnittstellenspezifikation die Klienten– und Server–Stubs erzeugen. Die Schnittstellenspezifikation aus Abschnitt 3.4.1 hat unter der DCE–IDL folgendes Aussehen:

```
[
    uuid(5526050c-fb2e-11cf-8059-08005ac71af9),
    version(1.0)
]

interface PrintServiceInterface {
    // Druckauftrag absetzen
    void SubmitJob(
```

```
        [in, string, ptr] char *Data,
        [out]             long  JobNumber
      );
      // Druckauftrag entfernen
      void KillJob(
        [in] long JobNumber
      );
}
```

Eine Schnittstellenspezifikation wird unter Angabe von Schnittstellenattributen eingeleitet, die durch eckige Klammern eingegrenzt sind. Auf sie wird weiter unten eingegangen. Anschließend folgt die Spezifikation der operationalen Schnittstelle. Die Syntax unterscheidet sich nur geringfügig von der Schnittstellenspezifikation aus Abschnitt 3.4.1. Den Parametern einer Funktion sind Attribute zugeordnet, die die Art des Parameters charakterisieren. Neben den Schlüsselwörtern `in` und `out` für Ein– und Ausgabeparameter kennzeichnen die Schlüsselwörter `ptr` bzw. `string` die zugehörigen Parameter als Zeiger bzw. Zeichenketten.

Eine Schnittstellenspezifikation auf Basis der DCE–IDL kann nicht als Parametertyp dienen. DCE macht einen Unterschied zwischen einer Schnittstellenspezifikation in der DCE–IDL und den benutzerdefinierten Typen innerhalb einer Schnittstellenspezifikation. Diese Eigenschaft wird in der Literatur als *First–Classness* bezeichnet (siehe [1]). Eine Schnittstellenbeschreibungssprache unterstützt *First–Class*–Schnittstellenspezifikationen, wenn eine Schnittstellenspezifikation als Parametertyp einer Funktion erlaubt ist (siehe [28]). Der Bezeichner `PrintServiceInterface` aus dem obigen Beispiel kann somit nicht als Typ verwendet werden, eine Referenz auf eine operationale Schnittstelle kann in DCE nicht weitergereicht werden. DCE sieht nur die Möglichkeit vor, einen *Funktionsverweis* (englisch: *Function Pointer*) als Parameter einer Funktion mitzugeben. Das gerufene Objekt kann mit dem Funktionsverweis eine entfernte Funktion aufrufen. Für die Spezifikation der Funktionssignatur steht jedoch nicht die vollständige DCE–IDL zur Verfügung.

Die Typkonformität zweier Typen basiert in DCE auf dem Inklusions–Polymorphismus. Ein Typ T_1 ist konform zu einem Typ T_2 genau dann, wenn:

1. alle Funktionen, die in T_2 definiert sind, ebenfalls in T_1 definiert sind und
2. die Reihenfolge und die Signatur der in T_2 definierten Funktionen mit denen in T_1 übereinstimmen.

D.h. der Typ T_1 darf den Typ T_2 um zusätzliche Funktionen erweitern, ohne die Reihenfolge der in T_2 aufgeführten Funktionen zu ändern. Das Prinzip der Austauschbarkeit zweier Typen, die sich aus dem Inklusions–Polymorphismus ableitet, wird durch diese Definition nicht verletzt, da ein Typ durch einen anderen ausgetauscht werden darf, der mindestens die gleiche Anzahl identischer Methoden besitzt. Die Austauschbarkeit der Typkonformität bei DCE nimmt nur Bezug auf die Spezifikation der operationalen Schnittstellen. Davon unberührt ist deren Semantik. Zwei Funktionen mit gleichen Signaturen aus zwei unterschiedlichen Schnittstellenspezifikationen müssen nicht notwendigerweise die gleiche Implementation besitzten, wodurch sich die Funktionen trotz gleicher Signaturen durch eine andere Semantik auszeichnen. Die Typkonformität bezieht sich bei DCE

auf die Menge von Nachrichten, die an einer operationalen Schnittstelle akzeptiert werden. Die mit einer Funktion assoziierte Semantik kann während der Dienstvermittlung aufgrund der syntaktischen Typbeschreibungssprache nicht auf Konformität überprüft werden.

Um Schnittstellenspezifikationen nicht zur Laufzeit handhaben zu müssen, werden in DCE jeder Schnittstellenspezifikation eine global eindeutige Identifikation (englisch: *Universal Unique Identifier*, UUID) und eine Versionsnummer zugewiesen. Diese eindeutige Identifikation ist Teil der bereits oben erwähnten Schnittstellenattribute der DCE–IDL. Eine Schnittstellenspezifikation wird in DCE zur Laufzeit ausschließlich durch das Tupel bestehend aus UUID und Versionsnummer repräsentiert. Die Austauschbarkeit zweier Typen wird in DCE zur Laufzeit anhand eines Vergleichs zweier Tupel der Form $(UUID_i, V_i)$ entschieden. Sind $UUID_i$ und V_i die UUID und Versionsnummer von T_i mit $i = 1, 2$, dann ist T_1 konform zu T_2 genau dann, wenn $UUID_1 = UUID_2$ und $V_1 \geq V_2$. Ein Dienst ist somit austauschbar mit einem anderen Dienst, der eine gleiche oder höhere Versionsnummer bei gleicher UUID besitzt. Höhere Versionsnummern können als Weiterentwicklung eines Dienstes interpretiert werden, weswegen der Inklusions–Polymorphismus bei DCE auch als *Versionierung* bezeichnet wird (siehe [60]).

Die Zuweisung einer UUID und einer Versionsnummer an eine Schnittstellenspezifikation resultiert in zwei unterschiedlichen Definitionen der Typkonformität in DCE. Einerseits definiert DCE den Inklusions–Polymorphismus als Teilmengenbeziehung der Methodenmengen zweier Typspezifikationen. Das kann aber zur Laufzeit nicht überprüft werden, da zu diesem Zeitpunkt außer UUID und Versionsnummer keine Typinformationen verfügbar sind. Andererseits existiert eine Definition des Inklusions–Polymorphismus auf Basis eines Vergleichs von UUIDs und Versionsnummern, anhand derer die Typkonformität zur Laufzeit überprüfbar ist. Es liegt in der Verantwortung des Programmierers, die Konsistenz der beiden Definitionen der Typkonformität einzuhalten. Er muß UUID und Versionsnummer dergestalt den Typspezifikationen zuordnen, daß die Typkonformität für beide Varianten übereinstimmt. DCE kann zur Laufzeit eine Inkonsistenz nicht feststellen, so daß eine Mißachtung der Regeln seitens des Programmierers zu einem Laufzeitfehler führt.

4.1.2 Common Object Request Broker Architecture

Die *Object Management Group* (OMG), ein Konsortium bestehend aus über 500 Firmen, veröffentlichte unter der Bezeichnung *Common Object Request Broker Architecture* (CORBA) eine Spezifikation für eine offene verteilte Infrastruktur (siehe [78]). Das Objektmodell dient als Grundlage dieser Infrastruktur.

Die Objekte werden, analog zu DCE, durch eine eigene Schnittstellenbeschreibungssprache typisiert. Die CORBA–IDL erlaubt die Spezifikation einer operationalen Schnittstelle in Form einer Menge von Funktionen. Jede Funktion besitzt eine Folge von typsierten Ein– und Ausgabeparametern. Basistypen und zusammengesetzte Typen sind im Sprachumfang der Schnittstellenbeschreibungssprache enthalten und an die Notation der Programmiersprache C++ angelehnt. CORBA ermöglicht die Wiederverwendung von Typspezifikationen. Ist ein Typ T_1 von einem Typ T_2 abgeleitet, dann erbt T_1 sämtliche Eigenschaften (d.h. Funktionen, benutzerdefinierte Typen, usw.) von T_2.

Weiterhin erlaubt die CORBA–IDL die Spezifikation von *Ausnahmen* (englisch: *Excep-*

tion) und *Kontexten* (englisch: *Context*). Eine Ausnahme bezeichnet eine benutzerdefinierte Fehlermeldung, die in einer Fehlersituation während der Ausführung einer Methode an den Aufrufer weitergeleitet wird. Eine Ausnahme ist vergleichbar mit einer besonderen Form eines Ausgabeparameters. Die Spezifikation von Kontexten mit der CORBA–IDL ähneln dagegen impliziten Eingabeparametern. Ein Klient kann einem Server einen Teil seines Zustands zur Verfügung stellen, auf den der Server über besondere Mechanismen Zugriff nehmen kann. Ein Kontext beinhaltet hierdurch Informationen eines Klienten, die nicht explizit als Eingabeparameter einer Methode enthalten sein müssen. Auf Basis der Schnittstellenspezifikation aus Abschnitt 3.4.1 demonstriert die folgende Spezifikation gemäß der CORBA–IDL den Gebrauch der Vererbung sowie benutzerdefinierter Ausnahmen.

```
interface ComplexPrintServiceInterface : PrintServiceInterface {
    // Typ fuer Fehlermeldung
    exception PrinterFailure {
      string reason;
    };

    // Laenge der Warteschlange ermitteln
    short QueueLength()
      raises( PrinterFailure );
}
```

Die Schnittstellenspezifikation `ComplexPrintServiceInterface` ist von der Schnittstellenspezifikation `PrintServiceInterface` abgeleitet. D.h. alle Definitionen der Schnittstelle `PrintServiceInterface` werden an `ComplexPrintServiceInterface` vererbt. Auf die in `PrintServiceInterface` definierten Funktionen `SubmitJob` und `KillJob` kann in der Schnittstellenspezifikation von `ComplexPrintServiceInterface` Bezug genommen werden, als ob sie dort definiert wären. Die Funktion `QueueLength`, die in der Schnittstellenspezifikation von `ComplexPrintServiceInterface` zusätzlich definiert ist, liefert im Fehlerfall eine Ausnahme. Die Ausnahme mit dem Namen `PrinterFailure` enthält eine Klartextfehlermeldung in Form einer Zeichenkette.

Die CORBA–IDL unterstützt die Strukturierung einer Schnittstellenspezifikation durch Module. Ein Modul ist ein in sich abgeschlossener Namensraum, der Namenskonflikte vermeiden hilft. Der Zugriff auf einen Namen eines Moduls geschieht durch qualifizierte Namen, bestehend aus dem Modulnamen und dem Namen selbst (die Qualifizierung von Namen unter CORBA ist mit der von C++ identisch). Die CORBA–IDL erlaubt *First–Class*–Schnittstellenspezifikationen. Die Schnittstellenspezifikation einer operationalen Schnittstelle kann als Parametertyp in einer anderen Schnittstellenspezifikation eingesetzt werden. Der Bezeichner, der dem Schlüsselwort `interface` folgt, ist ein gültiger Typbezeichner, der an jeder Stelle einsetzbar ist, wo benutzerdefinierte Typen erlaubt sind. So kann beispielsweise der Bezeichner `ComplexPrintServiceInterface` in einer Schnittstellenspezifikation als Parametertyp dienen. Dies ist die Voraussetzung für die Weitergabe von Objektreferenzen.

Die Architektur von CORBA sieht ein *Interface Repository* vor, welches zur Laufzeit Auskunft über Typspezifikationen zur Verfügung stellt. Dies erlaubt im Vergleich zu

DCE eine zur Laufzeit überprüfbare Definition der Typkonformität. Die Typkonformität ist eine Verallgemeinerung der bei DCE vorgestellten Definition. Der Typ T_1 ist konform zu einem Typ T_2 genau dann, wenn für jede Funktion f_2 des Typs T_2 eine Funktion f_1 des Typs T_1 mit folgenden Eigenschaften existiert:

1. Der Funktionsname von f_1 und f_2 sind identisch.
2. Die Anzahl der Eingabeparameter von f_1 und f_2 sowie deren Typen stimmen überein.
3. Die Anzahl der Ausgabeparameter von f_1 und f_2 sowie deren Typen stimmen überein.

Diese Definition schließt nicht aus, daß der Typ T_1 mehr Funktionen als der von T_2 umfassen kann. Die Typkonformität ist unabhängig von CORBAs Vererbungsmechanismus für Schnittstellenspezifikationen. Ist ein Typ von einem anderen abgeleitet, so ist er nach obiger Definition konform zu diesem, aber die Umkehrung der Aussage gilt allgemein nicht. Da das *Interface Repository* zur Laufzeit Informationen über Typen bereitstellt, kommt die Schnittstellenbeschreibungssprache von CORBA ohne eine UUID und eine Versionsnummer aus. Die Eindeutigkeit einer Schnittstellenspezifikation wird durch den qualifizierten Namen der Schnittstelle erreicht. Die Typkonformität ist allgemeiner als die von DCE, weil die Reihenfolge der Funktionen einer Schnittstelle keine Rolle spielt. In diesem Sinne kann die Konformität zweier Typen nach CORBA als Teilmengenbeziehung ihrer Mengen von Funktionen angesehen werden.

4.1.3 Open Distributed Processing

Das RM–ODP wurde gemeinsam von den beiden Standardisierungsgremien *International Organization for Standardization* (ISO) und *International Telecommunication Union, Telecommunication Standardization Sector* (ITU–T) entwickelt, um ein Rahmenwerk für die Standardisierungsbemühungen im Bereich des *Open Distributed Processing* (ODP) zu schaffen. Das Modell beschreibt eine Architektur, die die Unterstützung von Verteilung, Interoperabilität und Portabilität bietet. Das RM–ODP bedient sich dabei verschiedener *Sichten* (englisch: *Viewpoints*) auf ein System, die verschiedene Abstraktionen der Beschreibung des Systems erlauben und hierdurch unterschiedliche Eigenschaften in Abhängigkeit der Sichtweise hervorheben (siehe [42], [43], [44] und [45]).

Da das RM–ODP keine konkrete Technologie für ein offenes verteiltes System vorschreibt, sondern nur als Rahmenwerk gilt, wird die Wahl einer geeigneten Typbeschreibungssprache offengelassen. Das RM–ODP schreibt lediglich vor, welche Bedingungen eine Typbeschreibungssprache erfüllen muß. Dazu definieren die Standards den Begriff des *Dienstes* (englisch: *Service*), der an einer Schnittstelle eines Objekts erbracht wird. Nach der Terminologie des RM–ODP heißt ein Objekt, welches einen Dienst in Anspruch nimmt, *Dienstnutzer*. Ein Objekt, das einen Dienst anderen Objekten anbietet, wird als *Dienstanbieter* bezeichnet.

Ein Dienst ist Instanz eines *Diensttyps* (englisch: *Service Type*), der die relevanten Eigenschaften seiner Instanzen spezifiziert (siehe auch Abbildung 4.1). Ein Diensttyp besteht aus einem *Schnittstellentyp* (englisch: *Interface Type*) und den *Diensteigenschaftstypen* (englisch: *Service Property Types*). Der Schnittstellentyp spezifiziert eine operationale

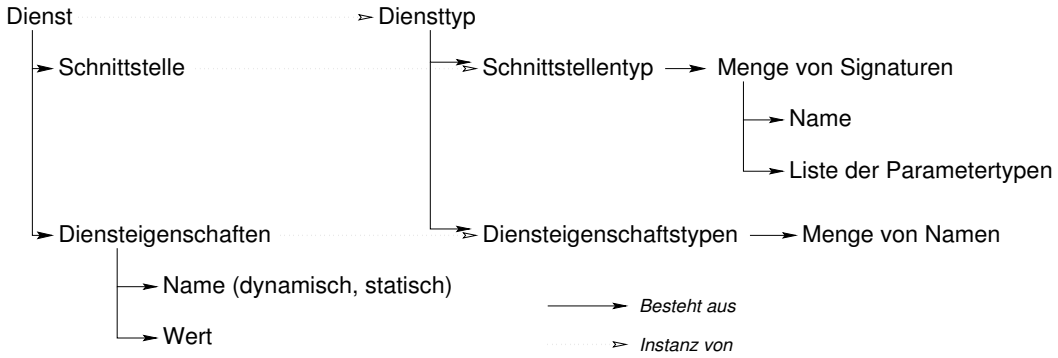


Abbildung 4.1: Definition eines Diensttyps in ODP.

Schnittstelle als eine Menge von Funktionen. Die Diensteigenschaften erlauben weitere Aussagen über die Charakteristika eines Dienstes, beispielsweise in Form von beschreibenden Attributen. Das RM–ODP unterscheidet zwischen statischen und dynamischen Attributen. Der Wert eines statischen Attributs ändert sich im Unterschied zu dem eines dynamischen über die Lebensdauer eines Dienstanbieters nicht.

Das RM–ODP definiert einen Diensttyp als ein Tupel, bestehend aus einer Schnittstellenbeschreibung und einer Menge von Eigenschaften in Form von Attributen dieses Dienstes. Die Schnittstellenbeschreibung ist in einer eigenständigen Komponente, dem Typmanager[1] (englisch: *Type Repository*) untergebracht. Obwohl das RM–ODP keine bestimmte Schnittstellenbeschreibungssprache vorschreibt, definiert es dennoch, unter welchen Bedingungen eine Schnittstelle durch eine andere ausgetauscht werden kann. Die abstrakte Spezifikation der Typkonformität ist auf unterschiedliche Schnittstellenbeschreibungssprachen anwendbar und kann beispielsweise auf das Typsystem von CORBA oder DCE übertragen werden. Dabei zeigt sich, daß die Definition der Typkonformität nach dem RM–ODP flexibler als die von DCE und CORBA ist, d.h., zwei Schnittstellen, die in CORBA (bzw. DCE) zueinander konform sind, sind es auch nach dem RM–ODP. Die Umkehrung gilt jedoch allgemein nicht. Ein Schnittstellentyp T_1 ist konform zu einem Schnittstellentyp T_2 genau dann, wenn es für jede Funktion f_2 von T_2 eine Funktion f_1 von T_1 mit folgenden Eigenschaften existiert:

1. Der Funktionsname und die Parameternamen von f_1 stimmen mit denen von f_2 überein.

2. Jeder Typ eines Eingabeparameters von f_2 ist konform zu dem entsprechenden Typ des Eingabeparameters von f_1.

3. Jeder Typ eines Ausgabeparameters von f_1 ist konform zu dem entsprechenden Typ des Ausgabeparameters von f_2.

Das RM–ODP geht bei der Definition der Typkonformität über die reine Erweiterbarkeit von Schnittstellen hinaus. Während bei DCE und CORBA die Menge der gemeinsamen Operationen identisch sein muß (in DCE müssen sie zusätzlich die gleiche

[1]Der Typmanager ist mit dem *Interface Repository* von CORBA vergleichbar; dessen Funktionalität ist jedoch im RM–ODP noch nicht standardisiert.

Reihenfolge aufweisen), erlaubt das RM–ODP eine Austauschbarkeit auf Parameterebene durch eine eigene Konformität für Parametertypen. Die Typkonformität von Parametertypen ist definiert über deren Wertebereich. Ein Parametertyp wird hier als eine Menge von gültigen Werten aufgefaßt. Ein Parametertyp T_1^P ist typkonform zu einem Parametertyp T_2^P genau dann, wenn der Wertebereich von T_1^P eine Teilmenge des Wertebereichs von T_2^P ist.

Regel 2 wird als *Kontravarianz* der Eingabetypen bezeichnet und Regel 3 als *Kovarianz* der Ausgabetypen. Diese Bezeichnungen beziehen sich auf die Richtung, in der die Parametertypen von f_1 und f_2 konform zueinander sind. In dem Fall der Ausgabetypen entspricht sie mit der Richtung der Schnittstellentypen T_1 und T_2 überein (Kovarianz), während sie bei den Eingabetypen gerade anders herum verläuft (Kontravarianz). Die folgende Schnittstellenspezifikation ist beispielsweise konform zu der Schnittstellenspezifikation `SimplePrintServiceInterface` aus Abschnitt 3.4.1, da der Ausgabeparametertyp `short` mit dem Wertebereich $[-2^{15} \ldots 2^{15}-1]$ konform zu dem Ausgabeparametertyp `long` mit dem Wertebereich $[-2^{31} \ldots 2^{31}-1]$ ist (Kovarianz):

```
interface AnotherSimplePrintServiceInterface  {
    // Druckauftrag absetzen
    void SubmitJob (
      in  string Data,
      out short  JobNumber
    );
}
```

Die formale Definition der Typkonformität innerhalb des RM–ODP bezieht sich nur auf den Vergleich der Schnittstellentypen. Der Vergleich von Diensteigenschaftstypen wird durch diese Definition nicht abgedeckt. Ein Dienstanbieter kann über Diensteigenschaften seinen Dienst weiter charakterisieren. Für das obige Beispiel eines Druckdienstes wäre beispielsweise die Druckgeschwindigkeit eine Diensteigenschaft. Der Export eines Dienstanbieters enthält die Spezifikation der operationalen Schnittstelle und eine Menge von Diensteigenschaften. Der Dienstnutzer charakterisiert während eines Imports seinen Wunsch in Form einer operationalen Schnittstellenspezifikation und einer Menge von *Auswahlregeln* (englisch: *Matching Contraints*) für die Diensteigenschaften. Die Ausdruckskraft der Auswahlregeln entspricht denen von arithmetischen Ausdrücken.

4.2 Semantische Typsysteme

Ein semantisches Typsystem besitzt eine Typbeschreibungssprache, die eine explizite semantische Spezifikation von Diensttypen erlaubt. Neben der höheren Ausdruckskraft der Notation erlauben semantische Typspezifikationen eine erweiterte Interpretation des Austauschbarkeitsprinzips. Dies beruht auf der Tatsache, daß die Zugriffssicherheit bei semantischen Typsystemen nicht nur über die an einer Objektschnittstelle angeboteten Funktionen definiert ist, sondern ebenfalls von der Semantik der Funktionen abhängig ist. Aufgrund der Relevanz der syntaktischen Typsysteme in offenen verteilten Systemen wird in Abschnitt 4.2.1 zunächst eine semantische Erweiterung einer syntaktischen Typbeschreibungssprache vorgestellt. Anschließend wird in Abschnitt 4.2.2 ein semantisches

Typsystem vorgestellt, welches im Unterschied zu allen bisher diskutierten Typsystemen benutzerorientierte Spezifikationen erlaubt.

4.2.1 Semantische Erweiterung syntaktischer Typsysteme

Die im letzten Abschnitt vorgestellten syntaktischen Typbeschreibungssprachen erlauben die Spezifikation von operationalen Schnittstellen. Ein Dienst, der von einem Objekt erbracht wird, ist charakterisiert als eine Menge von Nachrichten, die es an seiner Schnittstelle akzeptiert. Die auf dem Austauschbarkeitsprinzip basierende Typkonformität garantiert, daß, wenn ein Typ T_1 konform zu einem Typ T_2 ist, alle Nachrichten, die von einem Objekt vom Typ T_2 akzeptiert werden, auch von einem Objekt mit Typ T_1 angenommen werden. Diese Definition der Typkonformität nimmt lediglich Bezug auf die Signatur der Nachrichten, jedoch nicht auf die Semantik der sie implementierenden Methoden.

Daß die Semantik nicht nur die Ausdruckskraft einer Typspezifikation erhöht, sondern darüber hinaus eine flexiblere Definition der Typkonformität ermöglicht, zeigt folgendes Beispiel. Gegeben seien zwei Schnittstellenspezifikationen auf Basis einer syntaktischen Typbeschreibungssprache. Instanzen der Typen, die durch die Schnittstellenspezifikationen definiert werden, dienen dem Speichern und Auslesen von numerischen Werten. Beide Typen spezifizieren hierdurch die Funktionalität eines *Container*. Während jedoch bei dem einen Typ die Reihenfolge, in der die numerischen Datenwerte abgespeichert werden, signifikant ist, spielt dies bei dem anderen Typ keine Rolle. Eine mögliche Spezifikation dieser beiden Typen könnte auf Basis der CORBA–IDL wie folgt aussehen:

```
interface Bag {
     void Put( in long n );
     void Get( out long n );
}
```

```
interface Stack {
     void Push( in long n );
     void Pop( out long n );
}
```

Aufgrund ihrer Semantik sei der linke Typ T_B als *Bag* bezeichnet, und der rechte Typ T_S als *Stack*. Die `Get`–Methode des *Bag* ist nicht–deterministisch, d.h. die Auslesereihenfolge der in dem *Bag* zuvor gespeicherten Werte ist nicht relevant. Dahingegen arbeitet ein *Stack* nach dem LIFO–Prinzip: Die `Pop`–Methode liefert das zuletzt gespeicherte Element zurück. Bezüglich aller im Abschnitt 4.1 vorgestellten Definitionen für Typkonformität ist festzustellen, daß weder T_B konform zu T_S ist, noch T_S konform zu T_B ist. Die Definition einer Typkonformität für syntaktische Typsysteme nehmen Bezug auf die Signaturen der zu vergleichenden Typen, die in diesem Fall wechselseitig nicht übereinstimmen.

Das Austauschbarkeitsprinzip, das dem Inklusions–Polymorphismus zugrunde liegt, garantiert eine Zugriffssicherheit eines Dienstnutzers auf einen Dienstanbieter. Im Kontext der syntaktischen Typsysteme wird unter der Zugriffssicherheit die Garantie verstanden, daß eine Menge von Nachrichten mit bestimmten Signaturen an der Schnittstelle eines Dienstanbieters akzeptiert werden. Unter einer erweiterten Interpretation des Austauschbarkeitsprinzips, die die Semantik der Methoden berücksichtigt, wäre jedoch der Typ T_S konform zum Typ T_B. Greift ein Dienstnutzer auf einen Dienstanbieter vom Typ *Bag* zu, so bliebe die Zugriffssicherheit gewahrt, wenn der Dienstanbieter durch eine Instanz vom Typ *Stack* ausgetauscht würde (bei gleichzeitiger Anpassung der unterschiedlichen Funktionsnamen). Die LIFO–Eigenschaft des *Stack* widerspricht nicht dem Nicht–Determinismus des *Bag*.

Während eine syntaktische Schnittstellenspezifikation festlegt, welche Nachrichten akzeptiert werden, beschreibt die Semantik deren Auswirkungen und hierdurch das Verhalten eines Dienstes. Ein Programmierer assoziiert bei einer gegebenen Schnittstelle eine informale Semantik, die seine intuitive Vorstellung über die Funktionsweise eines Dienstes widerspiegelt. Die Semantik der Typspezifikationen T_S und T_B läßt sich somit nur indirekt aus der Wahl der Bezeichner ableiten. Da die Beschreibung der Semantik nicht expliziter Bestandteil einer syntaktischen Typbeschreibungssprache ist, kann sie einem Vermittler nicht als Entscheidungsgrundlage für die Überprüfung der Typkonformität dienen. Dienstangebot und –nachfrage auf Basis der syntaktischen Schnittstellenspezifikationen T_S und T_B können hierdurch nicht zugeordnet werden.

Der Typ T_S kann nur dann als typkonform zu T_B erkannt werden, wenn eine Beschreibung der Semantik ihrer Methoden, die die Funktionen implementieren, in den Typspezifikationen enthalten ist. Eine semantische Typbeschreibungssprache bietet eine Notation, mit der dies möglich ist. Aufgrund der Relevanz von syntaktischen Typbeschreibungssprachen bei der Spezifikation operationaler Objektschnittstellen in offenen verteilten Systemen wurde im Rahmen des Larch–Projekts (siehe [24]) eine semantische Erweiterung syntaktischer Typsysteme vorgeschlagen. Die Notation einer Typbeschreibungssprache wird erweitert, so daß die Semantik der einzelnen Funktionen expliziter Bestandteil einer Typspezifikation ist. Als Ergebnis des Larch–Projekts existieren zahlreiche Anbindungen an bestehende Typbeschreibungssprachen, die diese semantisch erweitern, wie beispielsweise für die CORBA–IDL oder der Programmiersprache C++ (siehe beispielsweise [53]).

In Larch wird die Semantik eines Typs über einen Zustandsautomaten spezifiziert. Instanzen des Typs befinden sich während ihrer Lebensdauer in genau einem der Zustände des Automaten. Eine Nachricht, die an die Schnittstelle einer Instanz gerichtet wird, bewirkt eine Zustandstransition. Die Larch–Notation erlaubt die Spezifikation einer Transition in Form von Vor– und Nachbedingungen, die den Funktionen des Typs zugeordnet sind. Analog zu dem Hoare–Kalkül (siehe [3]) beschreibt die Vorbedingung, unter welchen Bedingungen die Funktion aufgerufen werden darf. Die Nachbedingung setzt über einen prädikatenlogischen Ausdruck den Zustand vor und nach der Ausführung der Funktion logisch in Beziehung zueinander. Das folgende Beispiel erweitert die Schnittstellenspezifikation des *Bag* um eine Verhaltensbeschreibung. Die Notation ist an Larch angelehnt.

```
STATE-SPACE content: SET OF long;
INIT        content = ∅;

interface Bag {
    PRE true
    void Put( in long n );
    POST content' = content ∪ {n}

    PRE content ≠ ∅
    void Get( out long n );
    POST n ∈ content ∧ content' = content \ {n}
}
```

Der Zustandsraum wird über das Schlüsselwort `STATE-SPACE` beschrieben. Einer In-

stanz des Typs *Bag* ist zu einem Zeitpunkt eine Menge von numerischen Werten des Typs `long` zugeordnet. Zu Beginn besitzt die Instanz keine Werte. Die Spezifikation des initialen Zustands geschieht über das Schlüsselwort `INIT`. Jeder Funktion der Schnittstellenspezifikation des *Bag* werden prädikatenlogische Ausdrücke als Vor– und Nachbedingung hinzugefügt, eingeleitet durch die Schlüsselwörter `PRE` und `POST`. Die Funktion `Put` besitzt keine Vorbedingung. Die Funktion `Get` darf hingegen nur dann aufgerufen werden, wenn der *Container* mindestens einen Wert enthält. Die Nachbedingung der Funktion `Put` besagt, daß im neuen Zustand der Eingabeparameter `n` gespeichert wurde. Die Nachbedingung der Funktion `Get` besagt, daß dem *Container* ein Wert entnommen wird. Das nicht–deterministische Verhalten des *Bag* drückt sich in dem Ausdruck $\texttt{n} \in \textit{content}$ aus, mit dem ein beliebiges Element der Menge *content* auswählt wird.

Eine Erweiterung einer syntaktischen Typbeschreibungssprache nach Larch erlaubt eine semantische Spezifikation eines Typs. Die Definition einer Typkonformität ist in Larch nicht enthalten. Aus Sicht der Larch–Notation dient die semantische Erweiterung als Annotation, die Auskunft für das Verhalten der Instanzen eines Typs gibt. Die Arbeiten von Liskov/Wing definieren auf Basis der Larch–Notation eine semantische Typkonformität (siehe [58]), so daß die Notation von Larch zu einem Typsystem erweitert wird. Die Definition der Typkonformität nach Liskov/Wing besteht aus einer Menge von Regeln, die einem Programmierer helfen sollen, die Typkonformität zwischen Typen festzustellen. Allerdings ist die von Liskov/Wing eingeführte Definition nicht entscheidbar und kann hierdurch nicht maschinell überprüft werden.

Nach Liskov/Wing ist eine Typspezifikation T_1 semantisch konform zu einer Typspezifikation T_2 genau dann, wenn sich einige Abbildungen von T_1 nach T_2 nach bestimmten Regeln konstruieren lassen. Eine *Abstraktionsfunktion* (englisch: *Abstraction mapping*) muß den Zustandsraum von T_1 auf den von T_2 abbilden. Ferner muß eine *Umbenennungsfunktion* (englisch: *Renaming mapping*) alle Funktionsnamen von T_2 auf die von T_1 abbilden. Alle Funktionen von T_1, die nicht im Bildbereich der Umbenennungsfunktion liegen, müssen durch eine *Erklärung* deren Auswirkungen auf den Objektzustand beschreiben (englisch: *Extension map*). Dadurch wird gewährleistet, daß Funktionen, die in T_1, aber nicht in T_2 definiert sind, das Austauschbarkeitsprinzip der semantischen Typkonformität nicht verletzen. Im Vergleich dazu werden zusätzliche Funktionen bei der syntaktischen Typkonformität nicht beachtet, da hier das Austauschbarkeitsprinzip lediglich sich auf die an einer Objektschnittstelle akzeptierten Nachrichten bezieht.

Die Definition der semantischen Typkonformität nach Liskov/Wing beinhaltet neben den oben angegebenen Abbildungen noch logische Bedingungen zwischen Vor– und Nachbedingungen der in T_1 und T_2 bzgl. der Umbenennungsfunktion zugeordneten Funktionen. Diese Bedingungen bilden das semantische Pendant zu den Kontra– und Kovarianzregeln der syntaktischen Typkonformität. Für zwei Funktionen f_1 und f_2 der Typen T_1 und T_2, die über die Umbenennungsfunktion aufeinander abgebildet werden, müssen folgende Bedingungen gelten:

$$\begin{aligned} \textit{pre}(f_2) &\Leftrightarrow \textit{pre}(f_1) \\ \textit{post}(f_1) &\Rightarrow \textit{post}(f_2) \end{aligned}$$

Die erste Regel wird *Pre–condition Rule* genannt und die zweite *Post–condition Rule*.

Die *Pre–condition Rule* garantiert, daß die Vorbedingungen der beiden Funktionen logisch äquivalent sind, während die *Post–condition Rule* ausdrückt, daß die Nachbedingung von f_2 schwächer als die von f_1 ist. Eine schwächere Nachbedingung von f_2 besagt, daß Nicht–Determinismen im Verhalten der Funktion f_2 in f_1 aufgelöst werden.

Angewandt auf die Typen T_S und T_B, ist nach Liskov/Wing T_S semantisch konform zu T_B. Die Abstraktionsfunktion kann den Zustandsraum eines *Stack* auf den eines *Bag* abbilden, indem die Einfügereihenfolge der Elemente einfach ignoriert wird. Die Umbenennungsfunktion bildet die Funktion `Put` auf `Push` und `Get` auf `Pop` ab. T_S enthält keine zusätzlichen Funktionen, so daß keine Erklärung benötigt wird. Bzgl. der abweichenden Semantik der Ausleseoperation gilt, daß die Nachbedingung der Funktion `Get` schwächer ist, als die der Funktion `Pop`, da `Get` ein beliebiges Element ausliest, während `Pop` immer das zuletzt abgespeicherte zurückliefert.

Ein semantisches Typsystem auf Basis der Larch–Notation und der Definition einer semantischen Typkonformität nach Liskov/Wing erfüllt die Anforderungen an ein Typsystem für offene verteilte Systeme nur teilweise. Die semantische Typkonformität ist nicht endscheidbar, da deren Definition lediglich als Richtlinien für Programmierer gedacht waren. Wegen der Unentscheidbarkeit der Prädikatenlogik erster Stufe können die *Pre–condition Rule* und die *Post–condition Rule* nicht maschinell bewiesen werden. Die Anforderungen an ein Typsystem für offene verteilte Systeme werden ebenfalls bzgl. der Abstraktion nicht erfüllt. Da die Larch–Notation eine syntaktische Typbeschreibungssprache semantisch erweitert, indem sie Vor– und Nachbedingungen zu jeder Funktion hinzufügt, ist die resultierende Typbeschreibungssprache für Anwender ungeeignet.

4.2.2 Benutzerorientierte Typsysteme

Die bisher vorgestellten Typbeschreibungssprachen dienen der Spezifikation von operationalen Schnittstellen. Deren Spezifikation wird auf einer Implementationsebene benötigt, um mit entfernten Objekten kooperieren zu können. Voraussetzung für einen Programmierer, der einen Dienstnutzer implementiert, ist die Kenntnis dieser operationalen Schnittstellen. Ein Dienstnutzer wird zur Laufzeit von einem Anwender gesteuert. Der Anwender bedient sich des Dienstnutzers, um Zugang zu dem Dienstangebot eines offenen verteilten Systems zu erlangen. Ob und wann der Dienstnutzer bei der Bewältigung seiner Aufgaben auf entfernte Diensterbringer zugreifen muß, ist transparent für den Anwender. Für einen Anwender besteht keine Notwendigkeit für den Umgang mit operationalen Schnittstellenspezifikationen.

Im Gegensatz dazu binden *generische Dienstnutzer* die Gruppe der Anwender aktiv in die Vermittlung von Diensten mit ein. Charakteristisch für einen generischen Dienstnutzer ist, daß er mit einer Menge von Dienstanbietern kooperiert, die auf einer operationalen Ebene eine identische Schnittstelle besitzen (siehe Abbildung 4.2). Ein generischer Dienstnutzer präsentiert dem Anwender die Funktionalität eines entfernten Diensterbringers über graphische Oberflächenelemente. Diese ermöglichen dem Anwender die Interaktion mit dem Diensterbringer. Vor der Interaktion nimmt der Anwender aktiv an der Auswahl eines geeigneten Diensterbringers bei. Zu den Architekturen, die generische Dienstnutzer unterstützen, gehören beispielsweise OpenDoc, OLE2, COSM oder das *World–Wide Web* (siehe [19], [104], [67] und [14]).

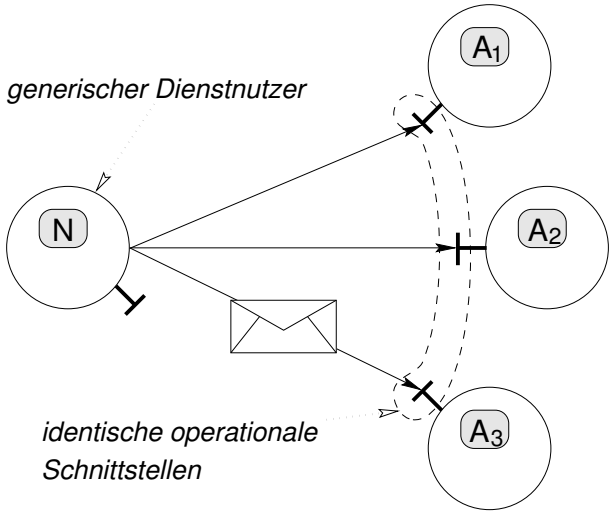


Abbildung 4.2: Generischer Dienstnutzer.

Eine syntaktische Typbeschreibungssprache ist für die Typisierung von Diensterbringern nicht mehr ausreichend, da sie ihre Funktionalität an identischen operationalen Schnittstellen anbieten. Der Einsatz von semantischen Typsystemen ist hierdurch unabdingbar, um zwischen unterschiedlichen Diensttypen unterscheiden zu können. Analog zu der im letzten Abschnitt vorgestellten Larch–Notation muß die syntaktische Schnittstellenspezifikation eines Dienstes um eine Verhaltensbeschreibung erweitert werden. Das Abstraktionsniveau einer auf Larch basierenden Spezifikation ist jedoch für einen Anwender nicht geeignet. Für diesen ist eine Typbeschreibungssprache notwendig, die möglichst seiner Vorstellungswelt entspringt und hierdurch intuitiv im Umgang ist. Ein benutzerorientiertes Typsystem, welches dies leistet, muß darüber hinaus eine Vermittlung von Typen unterstützen, da von der Gruppe der Anwender keine *a–priori*–Kenntnis der von ihnen gewünschten Diensttypen vorausgesetzt werden kann.

Unter den Architekturen, die generische Dienstnutzer unterstützen, hat das *World–Wide Web* (WWW) eine weite Verbreitung erlangt. Das Dienstangebot des WWW besteht aus multimedialen Informationen, die mit Hilfe von einfach zu bedienenden *Navigatoren* (englisch: *Browser*) Anwendern Zugang zu dem Informationsangebot bieten. Der Navigator entspricht einem generischen Dienstnutzer, der über eine Benutzerschnittstelle von einem Anwender gesteuert wird. Die operationale Schnittstelle zwischen generischen Dienstnutzern und Dienstanbietern wird durch das *Hypertext Transfer Protocol* (HTTP) induziert (siehe [12]). Das Protokoll definiert die Art und Weise, über die ein generischer Dienstnutzer Dokumente mit multimedialem Inhalt beziehen kann. Die Struktur der Dokumente — die im Rahmen des Protokolls als Ausgabeparameter der generischen Schnittstelle betrachtet werden können — ist über einen eigenen Standard festgelegt, dem *Hypertext Markup Language* (HTML) (siehe [13]). Die Information, die in mehreren logisch zusammengehörigen HTML–Dokumenten enthalten ist, wird im folgenden als Dienst angesehen. Mit einem Diensttyp sind entsprechend eine Menge von logisch zusammengehörigen HTML–Dokumenten assoziiert, die Instanzen dieses Diensttyps sind.

Gerade der stetige Zuwachs im Informationsangebot des WWW zeigt die Notwen-

digkeit einer Typvermittlung (siehe [14]). Ein Anwender, der über einen generischen Dienstnutzer Zugang zum WWW erhält, hat zunächst nur vage Vorstellungen über den Diensttyp, den er sucht. Vor der Vermittlung von Instanzen muß daher erst die Sichtbarkeit auf den gewünschten Diensttyp hergestellt werden. Da ein Anwender an der Vermittlung von Typen aktiv partizipiert, muß die Typbeschreibungssprache Typspezifikationen auf seinem Abstraktionsniveau erlauben. Bisher gibt es keine formalen Untersuchungen über Typsysteme, die sich auf das Abstraktionsniveau von Anwendern beziehen. Mit dem Erfolg des WWW ist ein *ad–hoc*–Typsystem entstanden. Spezielle *Suchmaschinen* (englisch: *Web–Crawler*), die die Aufgaben eines Dienstvermittler übernehmen, traversieren das Netzwerk. Anhand der HTML–Dokumente, die sie während der Traversierung vorfinden, erstellen sie automatisch eine Dienstdatenbank. Ein Dienstanbieter im WWW muß sein Angebot hierdurch nicht explizit exportieren. Die Typbeschreibungssprache besteht hier aus Wörtern einer Sprache wie Englisch oder Deutsch, die den textuellen Teilen der HTML–Dokumente entnommen sind. Anwender können durch eine Menge von Suchbegriffen (d.h. Wörtern) Anfragen formulieren. Die Typkonformität wird als Teilmengenbeziehung von Wörtern definiert und ist hiernach gewahrt, wenn die Menge von Wörtern, die ein Anwender vorgibt, eine Teilmenge der Wörter bildet, die zu der Typspezifikation eines Dienstanbieters gehören.

Die Definition der Typkonformität wird bei einigen Suchmaschinen basierend auf Erfahrungen aus dem *Information Retrieval* (siehe [57]) erweitert. Oftmals wird zwischen einfachen und erweiterten Suchanfragen unterschieden. Während einfache Suchanfragen lediglich auf einer Menge von Wörtern basieren, können diese bei erweiterten Anfragen mit Gewichtungen versehen oder über bestimmte Operatoren verknüpft werden. Logische Operatoren erhöhen die Ausdruckskraft einer Anfrage, indem Suchbegriffe logisch verknüpft werden. Suchbegriffe, die über einen Proximity–Operator verknüpft sind, müssen in dem Zieldokument in einem bestimmten Wortabstand auftreten. Einige Suchmaschinen unterstützen syntaktische bzw. semantische Normalisierungen von Wörtern. Bei einer syntaktischen Normalisierung werden Wortstämme erkannt und die Wörter auf eine kanonische Form zurückgeführt (Rechtstrunkierung). Die semantische Normalisierung bedient sich eines elektronischen Wörterbuches, um die Suche auf Synonyme auszudehnen. Tabelle 4.1 gibt einen Überblick über die Suchmaschinen AltaVista, Yahoo!, InfoSeek und Excite in bezug auf die Eigenschaften ihrer Typsysteme (siehe [4], [115], [40] und [21]).

Typsysteme auf Basis von Wörtern eines HTML–Dokuments besitzen eine Vielzahl von Nachteilen:

- Es werden ausschließlich alphanumerische Informationen eines Dokuments spezifiziert. Die in HTML–Dokumenten zusätzlich enthaltenen Bilder, Audiodaten o.ä. spielen für die Diensttypspezifikation keine Rolle.
- Die Wörter stehen in keinem Kontext zueinander (d.h. zwei HTML–Dokumente, die die gleichen Wörter enthalten, besitzen denselben Typ). Ein Treffer wird ausschließlich durch gewisse „Reizwörter“ festgestellt.
- Unterschiedliche Benutzer haben unterschiedliche Sichten auf einen Dienst. Die Definition der Typkonformität basiert aber in der Regel auf den in einem HTML–Dokument verwendeten Wörtern.

	AltaVista	**Yahoo!**	**InfoSeek**	**Excite**
Suchmodi	Einfache/Erweiterte Suche	nur ein Suchmodus	nur ein Suchmodus	Schlüsselwortsuche mit Thesaurus
Suchbegriffe	Rechtstrunkierung; exakte Begriffe; *–Operator	Rechtstrunkierung; exakte Begriffe	Rechtstrunkierung; exakte Begriffe	Rechtstrunkierung; Gewichtung
Boole'sche Operatoren	AND, OR, NOT mit Klammerung	AND, OR	AND, OR, NOT	AND, OR, NOT mit Klammerung
Proximity Operator	Ja	Nein	Ja	Nein

Tabelle 4.1: Unterschiedliche Suchmaschinen im World–Wide Web.

- Eine Menge von HTML–Dokumenten können logisch gesehen einen Dienst repräsentieren, was von einer Suchmaschine nicht automatisch erkannt werden kann.
- Die Traversierung einer Menge von HTML–Dokumenten ist auf die transitive Hülle beschränkt. Eine Partitionierung des WWW kann von einer Suchmaschine weder erkannt noch überwunden werden.
- Die automatisch erstellten Indizes sind wegen der hohen Dynamik des WWW schnell veraltet. Ein Dienstanbieter weiß nach einer Rekonfiguration seiner HTML–Dokumente nicht, welche Suchmaschinen über die Änderung benachrichtigt werden müssen.

Die Aufzählung zeigt, daß eine Menge von Wörtern, die einem HTML–Dokument entnommen sind, keine ausreichende Ausdruckskraft bietet, um ein quantitativ großes Dienstangebot zu spezifizieren. Obwohl Techniken wie die semantische Normalisierung die Notwendigkeit der *a–priori*–Kenntnis von Diensttypen vermindern, ist zu erwarten, daß derartige Typbeschreibungssprachen in einem offenen verteilten System nicht skalierbar ist. Neben einzelnen Wörtern, die optional über Boolesche Operatoren miteinander verknüpft sind, fehlt die Beschreibung eines Kontextes, der die Wörter einer Suchanfrage zueinander in Beziehung setzt. Die erhöhte Ausdruckskraft einer solchen Notation macht die automatische Traversierung der Dokumente von Suchmaschinen komplizierter, da diese den Dokumentinhalt syntaktisch und semantisch analysieren müssen.

4.3 Zusammenfassung

Typsysteme spielen eine zentrale Rolle bei der Dienstvermittlung in offenen verteilten Systemen. Ein Dienstanbieter charakterisiert seine Funktionalität durch eine Typspezifikation auf Basis einer Typbeschreibungssprache. Ein Dienstnutzer seinerseits spezifiziert

Kriterium	**DCE**	**CORBA**	**ODP**	**Larch**	**Wörter**
TS_1: Skalierbarkeit	+	+	+	+	−
TS_2: OWA	−	−	−	−	+
TS_3: Polymorphismus	+	+	+	+	−
TS_4: Entscheidbarkeit	+	+	+	−	+
TS_5: Abstraktion	−	−	−	−	−

Tabelle 4.2: Bewertung bestehender Typsysteme.

über eine Typbeschreibungssprache den Diensttyp, an dessen Instanzen er gebunden werden möchte. Ein Vermittler führt Angebot und Nachfrage gemäß den Regeln einer Typkonformität zusammen, die eine Zugriffssicherheit zwischen Dienstnutzer und –anbieter garantiert.

In diesem Kapitel wurden unterschiedliche Typsysteme vorgestellt, die in offenen verteilten Systemen Einsatz finden. Die syntaktischen Typsysteme erlauben die Spezifikation von operationalen Schnittstellen. Die Typsysteme von DCE, CORBA und dem RM–ODP besitzen eine syntaktische Typbeschreibungssprache mit unterschiedlicher Ausdruckskraft. Die IDLs der vorgestellten syntaktischen Typsysteme unterscheiden sich nur in wenigen Details. Die Definition der Typkonformität ist bei dem RM–ODP am flexiblesten, die von CORBA ist flexibler als die von DCE.

Mit syntaktischen Typsystemen kann lediglich die Struktur der Nachrichten spezifiziert werden, die ein Objekt an seiner Schnittstelle akzeptiert. Die Beschreibung der Semantik der Funktionen ist erst mit einer semantischen Typbeschreibungssprache möglich. Die Arbeiten im Rahmen des Larch–Projekts ermöglichen die semantische Erweiterung einer syntaktischen Typbeschreibungssprache in Form von Vor– und Nachbedingungen. Eine Typkonformität zwischen zwei semantischen Typspezifikationen basiert auf einer semantischen Variante des Austauschbarkeitsprinzips. Trotz einer Verhaltensbeschreibung von Typspezifikationen ist die Larch–Notation zu abstrakt für Anwender. Innerhalb des WWW sind *ad–hoc*–Typsysteme entstanden, die einem Anwender die Spezifikation eines Diensttyps als Menge von Wörtern erlaubt. Die Einbeziehung von elektronischen Wörterbüchern während der Überprüfung der Typkonformität gewährt Anwendern ihre eigene Sicht auf einen Diensttyp. Tabelle 4.2 gibt eine zusammenfassende Bewertung der in diesem Kapitel vorgestellten Typsysteme gemäß der in Abschnitt 3.6 eingeführten Kriterien.

Kapitel 5

Formales Modell der Dienstvermittlung

In diesem Kapitel wird das Modell der Dienstvermittlung in offenen verteilten Systemen formal spezifiziert. Eine *formale Spezifikation* bedient sich einer mathematischen Schreibweise, um präzise Aussagen über ein informationsverarbeitendes System zu machen, ohne jedoch auf implementationsspezifische Details einzugehen. Eine Spezifikation beschreibt, *was* ein System leistet, jedoch nicht, *wie* diese Leistung erbracht wird. Es stellt ein abstraktes Entwurfsartefakt dar, welches als Vorgabe einer Implementation dienen kann. Das in diesem Kapitel vorgestellte Modell der Dienstvermittlung erweitert die innerhalb der ODP–Standardisierungen zu findende formale Spezifikation eines ODP–konformen Vermittlers (siehe [46]). Während dort Import und Export als allgemeine Operationen eines Vermittlers definiert sind, wird in der folgenden Spezifikation zwischen der in Abschnitt 3.3.2 informell eingeführten *Instanzvermittlung* und *Typvermittlung* unterschieden.

Das formale Modell ist, analog zu der formalen Spezifikation des ODP–konformen Vermittlers, in der Spezifikationssprache Z beschrieben (siehe [99]). In Z wird die Semantik von Operationen durch Transitionen in einem abstrakten Zustandsraum definiert. Ein Z–Schema erlaubt sowohl die Darstellung des Zustandsraums und seiner Invarianten, als auch die Spezifikation der Zustandsübergänge. Der Aufbau dieses Kapitels reflektiert diese Unterscheidung. In Abschnitt 5.1 werden die statischen Schemata vorgestellt, die den Zustandsraum eines allgemeinen Objektgraphen spezifizieren. Unterschiedliche Operationen und Invarianten der dynamischen Schemata erlauben die präzise Unterscheidung zwischen einem *geschlossenen Objektsystem* (Abschnitt 5.2.1), einem *offenen Objektsystem* (siehe Abschnitt 5.2.2) und einem *Vermittlungssystem* (siehe Abschnitt 5.2.3).

5.1 Basistypen und statische Schemata

Die statischen Schemata einer Z–Spezifikation repräsentieren den abstrakten Zustandsraum, in dem sich das Modell bewegt. Die Struktur des Zustands basiert auf einer Menge von Basistypen, die die grundlegenden Voraussetzungen für das formale Modell eines Objektgraphen bilden. Die Spezifikation setzt typisierte Objekte voraus, die durch folgende Deklaration in Z eingeführt werden:

$[OBJECT, TYPE]$

Die Bezeichner *OBJECT* und *TYPE* stellen die Basistypen der Z–Spezifikation dar. Eine weitere Verfeinerung der Begriffe ist für die formale Darstellung der Dienstvermittlung nicht notwendig; d.h. das Modell schreibt weder eine konkrete Objektinfrastruktur, noch ein spezielles Typsystem vor. Lediglich deren Eigenschaften werden im folgenden weiter präzisiert.

Neben den Basistypen werden für die Modellierung der auf Seite 28 erläuterten Erreichbarkeit und Sichtbarkeit zusammengesetzte Typen eingeführt, ausschließlich beschrieben durch eine Komposition der Basistypen. Z ermöglicht die Spezifikation eines zusammengesetzten Typs ebenfalls durch ein statisches Schema. Die Erreichbarkeit greift auf die Objektidentität zurück, die ein Objekt in Kontext zu anderen Objekten stellt. Das Modell formalisiert Eins–zu–Eins–Zuordnungen zwischen Objekten durch typisierte Referenzen. Das Schema *ObjectLink* leistet die formale Darstellung einer Referenz. Die Komponenten *from* und *to* des Schemas beschreiben eine gerichtete Kante von einem Objekt zu einem anderen. Die Referenz besitzt den Typ *type*, der die Sicht des Dienstnutzerobjekts auf das Diensterbringerobjekt beschreibt. Die Typkonformität, die in einer Invariante zwischen dem Referenztyp und dem Typ des Diensterbringerobjekts resultiert, ist nicht Teil dieses Schemas.

ObjectLink

$from : OBJECT$
$to : OBJECT$
$type : TYPE$

Eine Instanz des Schemas *ObjectLink* entspricht einer Instanz der in Abschnitt 3.3.2 eingeführten Erreichbarkeit zwischen zwei Objekten. Die ebenfalls an dieser Stelle vorgestellte Sichtbarkeit wird durch das Schema *TypeLink* beschrieben. Eine Instanz dieses Schemas sagt aus, daß das Objekt *obj* Kenntnis des Typs *type* besitzt.

TypeLink

$obj : OBJECT$
$type : TYPE$

Das Schema *TypeSystem* definiert die grundlegenden statischen Eigenschaften eines Typsystems auf Basis des Inklusions–Polymorphismus. Die Menge *objects* repräsentiert alle zu einem Zeitpunkt bekannten Objekte. Analoges gilt für die Menge *types*. Jedem Objekt wird über die Funktion *has_type* genau ein Typ zugeordnet. Die Relation *is_conform* gibt Auskunft über polymorphe Beziehungen zwischen den im System bekannten Typen. Die Invarianten des Schemas formalisieren die Eigenschaften des Austauschbarkeitsprinzips: die Relation *is_conform* ist reflexiv, anti–symmetrisch und transitiv, die die Menge aller Typen zu einer Hierarchie anordnet. Ein Typ besitzt eine Menge von *Supertypen* und *Subtypen*. Jeder Typ ist konform zu all seinen Supertypen und alle Subtypen sind konform zu dem Typ selbst.

TypeSystem

$$
\begin{array}{l}
objects : \mathbb{P}\, OBJECT \\
types : \mathbb{P}\, TYPE \\
has\_type : OBJECT \rightarrow TYPE \\
is\_conform : TYPE \leftrightarrow TYPE \\
\hline
\mathrm{dom}\, has\_type = objects \\
\mathrm{ran}\, has\_type \subseteq types \\
\mathrm{dom}\, is\_conform = types \\
\mathrm{ran}\, is\_conform = types \\
\forall\, o : objects \bullet (\exists_1\, t : types \bullet has\_type(o) = t) \\
\forall\, t : types \bullet ((t, t) \in is\_conform) \\
\forall\, t_1, t_2 : types \bullet ((t_1 \neq t_2 \wedge (t_1, t_2) \in is\_conform) \Rightarrow (t_2, t_1) \notin is\_conform) \\
\forall\, t_1, t_2, t_3 : types \bullet (((t_1, t_2) \in is\_conform \wedge (t_2, t_3) \in is\_conform) \Rightarrow \\
\qquad (t_1, t_3) \in is\_conform)
\end{array}
$$

Ein Schnappschuß der Sichtbarkeit aller Objekte wird durch das Schema *Visibility* dargestellt. Konzeptionell ist jedem Objekt eine Menge von Typen zugeordnet, die es kennt. Anfragen eines Dienstnutzers an einen Vermittler beschränken sich auf die Typen, von denen der Dienstnutzer Kenntnis besitzt. Analog zu der Sichtbarkeit beschreibt das Schema *Reachability* einen Schnappschuß der Erreichbarkeit zwischen Objekten. Die Komponente *reachable* enthält eine Menge von *ObjectLink*–Instanzen, welche die zu diesem Zeitpunkt existierenden Referenzen angeben. Dieses Schema definiert auch die Invariante, die für alle Referenzen gelten muß. Da eine Referenz eine Sicht eines Dienstnutzers auf einen Dienstanbieter darstellt, muß der Typ des Dienstanbieters konform zu dem Referenztyp sein. Eine weitere Invariante besteht in der Forderung, daß, wenn ein Dienstnutzer eine Referenz auf ein anderes Objekt besitzt, der Referenztyp ihm bekannt (d.h. sichtbar) sein muß.

Visibility

$$
\begin{array}{l}
TypeSystem \\
visible : \mathbb{P}\, TypeLink \\
\hline
\forall\, v : visible \bullet (v.obj \in objects \wedge v.type \in types) \\
\forall\, o : objects \bullet (\exists\, v : visible \bullet (v.obj = o \wedge v.type = has\_type(o)))
\end{array}
$$

Reachability

$$
\begin{array}{l}
TypeSystem \\
Visibility \\
reachable : \mathbb{P}\, ObjectLink \\
\hline
\forall\, r : reachable \bullet (r.from \in objects \wedge r.to \in objects \wedge r.type \in types \wedge \\
\qquad (has\_type(r.to), r.type) \in is\_conform \wedge \\
\qquad (\exists\, v : visible \bullet (v.obj = r.from \wedge (v.type, r.type) \in is\_conform)))
\end{array}
$$

Das Schema *ObjectGraph* beschreibt letztendlich einen Schnappschuß eines Objektgraphen und seiner Invarianten. Der abstrakte Zustandsraum wird durch dieses Schema beschrieben. Ein Objektgraph besteht aus einem Typsystem mit einer Menge von Typen und Objekten, sowie einer Menge von Instanzen der Erreichbarkeits– und Sichtbarkeitsschemata. Formal wird ein Objektgraph durch die logische Konjunktion der Schemata *TypeSystem*, *Reachability* und *Visibility* dargestellt. Das Schema *ObjectGraph* beschreibt lediglich die Struktur des Zustandraums; der initiale Zustand hängt von dem betrachteten System ab und wird bei der Unterscheidung zwischen einem geschlossenen, einem offenen Objektsystem und einem Vermittlungssystem zu einem späteren Zeitpunkt nachgeholt.

$$ObjectGraph \mathrel{\widehat=} TypeSystem \land Reachability \land Visibility$$

5.2 Dynamische Schemata

Die statischen Schemata definieren eine Menge von gültigen Zuständen, die ein System einnehmen kann. Die Invarianten partitionieren den Zustandsraum in gültige und ungültige Zustände. Die möglichen Operationen eines Systems werden in der formalen Spezifikation durch Zustandsänderungen dargestellt. Die Operationen müssen einen gültigen Zustand in einen gültigen Folgezustand überführen; d.h. die Invarianten der statischen Schemata werden eingehalten.

Das Schema *CreateType* fügt einen neuen Typ in das System ein. Auslöser der Operation ist ein Objekt, welches den Typ zu der Menge der bekannten Typen hinzufügt. Das Schema nimmt zunächst den Typ *new_type*? zu der Menge *types* auf, und definiert dann eine Sichtbarkeit zwischen dem Erzeuger *obj*? und dem neuen Typ. Der neue Typ kann in der Typhierarchie an einer bestimmten Stelle eingeordnet werden. Die Parameter *sub_types*? und *super_types*? geben die Menge der Subtypen, bzw. Supertypen an. Die Relation *is_conform* wird entsprechend der Vorgaben verändert.

CreateType

$$\Delta TypeSystem$$
$$\Delta Visibility$$
$$creator? : OBJECT$$
$$new\_type? : TYPE$$
$$sub\_types? : \mathbb{P}\ TYPE$$
$$super\_types? : \mathbb{P}\ TYPE$$

$$creator? \in objects$$
$$new\_type? \notin types$$
$$(sub\_types? \cup super\_types?) \subseteq types$$

$$types' = types \cup \{new\_type?\}$$
$$visible' = visible \cup \{v : TypeLink \mid v.obj = creator? \land v.type = new\_type?\}$$
$$is\_conform' = (is\_conform \cup \{t_1, t_2 : TYPE \mid (t_1 \in sub\_types? \land t_2 = new\_type?) \lor (t_1 = new\_type? \land t_2 \in super\_types?) \bullet t_1 \mapsto t_2\})^*$$
$$objects' = objects \land has\_type' = has\_type$$

Das Schema *DeleteType* entfernt einen Typ aus dem System. Das Objekt *obj*?, welches die Operation auslöst, muß das einzige Objekt mit einer Sichtbarkeit auf den zu löschenden Typ *type*? sein. Ferner darf kein Objekt im System mit genau diesem Typ existieren. Die Operation löscht den Typ und entfernt ihn aus der Typhierarchie.

DeleteType

$$
\begin{array}{l}
\Delta\, TypeSystem \\
\Delta\, Visibility \\
obj? : OBJECT \\
type? : TYPE \\
\hline
obj? \in objects \land type? \in types \\
\exists\, v : visible \bullet (v.obj = obj? \land v.type = type?) \\
\forall\, o : objects \bullet (has\_type(o) \neq type? \land \\
\qquad (\forall\, v : visible \bullet (v.type = type? \Rightarrow v.obj = obj?))) \\
types' = types \setminus \{type?\} \\
visible' = visible \setminus \{v : visible \mid v.type = type?\} \\
is\_conform' = ((\{type?\} \lhd (is\_conform^*)) \rhd \{type?\})^* \\
objects' = objects \land has\_type' = has\_type
\end{array}
$$

Analog zu den dynamischen Schemata, die Typen in ein System einführen bzw. wieder löschen, werden im folgenden die Operationen zum Erzeugen und Löschen von Objekten formal spezifiziert. Das Schema *CreateObject* spezifiziert die Objekterzeugung. Auslöser der Operation ist das Objekt *creator*?, welches das neue Objekt *new_object*? mit dem Typ *object_type*? erzeugt. Das erzeugende Objekt muß den Typ *object_type*? kennen und erhält als Seiteneffekt eine Referenz dieses Typs auf das neue Objekt. Um die Invarianten des Schemas *ObjectGraph* zu erfüllen, erhält das neue Objekt zudem die Kenntnis seines eigenen Typs.

CreateObject

$\Delta TypeSystem$
$\Delta Reachability$
$\Delta Visibility$
$creator? : OBJECT$
$new\_object? : OBJECT$
$object\_type? : TYPE$

$creator? \in objects$
$new\_object? \notin objects$
$object\_type? \in types$
$\exists\, v : visible \bullet (v.obj = creator? \land v.type = object\_type?)$

$objects' = objects \cup \{new\_object?\}$
$has\_type' = has\_type \oplus \{(new\_object?, object\_type?)\}$
$reachable' = reachable \cup \{r : ObjectLink \mid r.from = creator? \land$
$\qquad r.to = new\_object? \land r.type = object\_type?\}$
$visible' = visible \cup \{v : TypeLink \mid v.obj = new\_object? \land$
$\qquad v.type = object\_type?\}$
$is\_conform' = is\_conform$

Die komplementäre Operation ist das Löschen eines Objekts. Das Schema *DeleteObject* entfernt ein zuvor erzeugtes Objekt. Ein Objekt entscheidet selbst über den Zeitpunkt seiner Löschung und ist ohne äußeren Einfluß. Diese konservative Sichtweise wird beispielsweise von Systemen mit *Garbage Collection* unterstützt, die generell weniger fehleranfällig sind. Die Voraussetzung, daß ein Objekt aus dem System entfernt werden kann, ist, daß es zum betreffenden Zeitpunkt von keinem anderen Objekt referenziert wird.

DeleteObject

$\Delta TypeSystem$
$\Delta Reachability$
$\Delta Visibility$
$obj? : OBJECT$

$\forall\, r : reachable \bullet (obj? \neq r.to)$

$objects' = objects \setminus \{obj?\}$
$has\_type' = \{obj?\} \lhd has\_type$
$reachable' = reachable \setminus \{r : reachable \mid r.from = obj?\}$
$visible' = visible \setminus \{v : visible \mid v.obj = obj?\}$
$types' = types \land is\_conform' = is\_conform$

Das Schema *AddReachability* erweitert die Relation *Reachability* um eine weitere Referenz und modelliert so die Propagation der Erreichbarkeit. Eine Referenz kann nur als Parameter eines Methodenaufrufes weitergereicht werden, der seinerseits eine Erreichbarkeit voraussetzt. Nach dieser Operation existieren zwei unabhängige Referenzen auf

ein Objekt. Die neue Referenz, die eine Kopie einer bereits bestehenden ist, muß nicht notwendigerweise den gleichen Typ besitzen. Da der Typ einer Referenz eine Sicht auf ein Objekt repräsentiert, kann die kopierte Referenz auch einen Supertyp des ursprünglichen Referenztyps besitzen. Der Parameter *link?* beschreibt durch seine Komponenten Ausgangs– und Ziel–Objekt der neuen Referenz, sowie deren Typ.

AddReachability

$$
\begin{array}{l}
\Xi TypeSystem \\
\Xi Visibility \\
\Delta Reachability \\
link? : ObjectLink \\
\hline
link?.from \in objects \land link?.to \in objects \land link?.type \in types \\
(has\_type(link?.to), link?.type) \in is\_conform \\
\exists\, v : visible \bullet (v.obj = link?.from \land v.type = link?.type) \\
(\exists\, r_1, r_2 : reachable \bullet (r_1.from = r_2.from \land r_1.to = link?.from \land \\
\qquad r_2.to = link?.to \land (r_2.type, link?.type) \in is\_conform)) \lor \\
\quad (\exists\, r : reachable \bullet (r.from = link?.to \land r.to = link?.from \land \\
\qquad (has\_type(r.from), link?.type) \in is\_conform)) \\
\\
reachable' = reachable \cup \{link?\}
\end{array}
$$

Ein Objekt kann eine Referenz über einen längeren Zeitraum hinweg speichern, indem die Referenz zu einem Teil des Objektzustands wird. Wird eine Referenz nicht mehr benötigt, modelliert das Schema *RemoveReachability* das Entfernen einer Referenz. Die einzige Vorbedingung dieser Operation ist, daß die Referenz existieren muß.

RemoveReachability

$$
\begin{array}{l}
\Xi TypeSystem \\
\Delta Reachability \\
link? : ObjectLink \\
\hline
\exists\, r : reachable \bullet (r = link?) \\
\\
reachable' = reachable \setminus \{link?\}
\end{array}
$$

Die letzten beiden Schemata dienen der Verwaltung der Erreichbarkeit zwischen Objekten. Die folgenden beiden Schemata verwalten analog dazu die Sichtbarkeit von Typen. Das Schema *AddVisibility* gibt einem Objekt die Kenntnis über einen Typ des Systems. Dieses Wissen ist für den noch vorzustellenden Vermittlungsvorgang notwendig. Ein Objekt kann das Wissen über einen Typ nur dann erlangen, wenn es von einem anderen Objekt erreichbar ist und welches eine ausreichende Kenntnis über den Typ besitzt. Der Begriff „ausreichend“ ist in Bezug auf die Typkonformität gemeint; d.h. es können nur Verallgemeinerungen (Supertypen) eines Typs propagiert werden.

AddVisibility

$\Xi TypeSystem$
$\Xi Reachability$
$\Delta Visibility$
$vis? : TypeLink$

$vis?.obj \in objects \land vis?.type \in types$
$\exists\, r : reachable \bullet (\exists\, v : visible \bullet (r.from = v.obj \land r.to = vis?.obj \land$
$\quad (v.type, vis?.type) \in is\_conform))$

$visible' = visible \cup \{vis?\}$

Durch die Anwendung des Schemas *RemoveVisibility* verliert ein Objekt die Kenntnis über einen Typ. Einzige Voraussetzung ist, daß die Kenntnis zuvor bestanden haben muß.

RemoveVisibility

$\Xi TypeSystem$
$\Xi Reachability$
$\Delta Visibility$
$vis? : TypeLink$

$\exists\, v : visible \bullet (v = vis?)$
$\forall\, r : reachable \bullet (r.from = vis?.obj \Rightarrow r.type \neq vis?.type)$

$visible' = visible \setminus \{vis?\}$

5.2.1 Geschlossenes Objektsystem

Anhand der bis zu diesem Punkt eingeführten statischen und dynamischen Schemata werden die Begriffe *geschlossenes Objektsystem*, *offenes Objektsystem* und *Vermittlungssystem* formal definiert. Diese Systeme werden durch Angabe eines Zustandsraums, eines initialen Zustands und einer Menge von Transitionen spezifiziert.

Ein geschlossenes Objektsystem ist durch die vollständige Sichtbarkeit aller im System vorhandenen Typen charakterisiert. Jedes Objekt besitzt deshalb Kenntnis aller Typen. Die Spezifikation des Zustandsraums erweitert das Schema *ObjectGraph* um weitere Invarianten und einen vordefinierten Typ *absurd_type*. Dieser Typ ist als die Spezialisierung aller bekannten Typen definiert. Sein Name deutet an, daß ihm keine Instanzen zugeordnet sind. Jedem Objekt wird bei seiner Erzeugung die Sichtbarkeit auf *absurd_type* gegeben, so daß wegen der Transitivität der Typkonformität dem neuen Objekt hierdurch alle anderen Typen sichtbar sind. Das folgende Schema *CompleteVisibility* schränkt den Zustandsraum ein, indem als Invariante gefordert wird, daß alle Objekte Kenntnis aller bekannten Typen besitzen. Der vollständige Zustandsraum eines geschlossenen Systems ist durch die konjunktive Verknüpfung der Schemata *ObjectGraph* und *CompleteVisibility* festgelegt (siehe *ClosedSystem*).

CompleteVisibility

$$
\begin{array}{l}
TypeSystem \\
Visibility \\
absurd\_type : TYPE \\
\hline
absurd\_type \in types \\
\forall\, t : types \bullet ((absurd\_type, t) \in is\_conform) \\
\forall\, o : objects \bullet (has\_type(o) \neq absurd\_type \wedge \\
\qquad (\forall\, t : types \bullet (\exists\, v : visible \bullet (v.obj = o \wedge (v.type, t) \in is\_conform))))
\end{array}
$$

$$ClosedSystem \mathrel{\widehat=} ObjectGraph \wedge CompleteVisibility$$

Das Schema *InitClosedSystem* definiert den initialen Zustand eines geschlossenen Objektsystems. Es existiert zunächst nur ein Wurzelobjekt *root_obj* vom Typ *root_type*. Der Typ *absurd_type* ist ein Subtyp von *root_type* und das Wurzelobjekt erhält die Kenntnis über *absurd_type*, um die Bedingungen des Schemas *CompleteVisibility* zu erfüllen.

InitClosedSystem

$$
\begin{array}{l}
ClosedSystem \\
root\_obj : OBJECT \\
root\_type : TYPE \\
\hline
objects = \{root\_obj\} \\
types = \{root\_type, absurd\_type\} \\
has\_type = \{root\_obj \mapsto root\_type\} \\
is\_conform = \{absurd\_type \mapsto absurd\_type, absurd\_type \mapsto root\_type, \\
\qquad root\_type \mapsto root\_type\} \\
visible = \{v : TypeLink \mid v.obj = root\_obj \wedge v.type = absurd\_type\} \\
reachable = \emptyset
\end{array}
$$

Zu den in einem geschlossenen Objektsystem möglichen Operationen gehören das Erzeugen eines Typs oder eines Objekts (*CSCreateType* bzw. *CSCreateObject*), Löschen eines Typs oder eines Objekts (*CSDeleteType* bzw. *CSDeleteObject*) und dem Hinzufügen oder Wegnehmen der Erreichbarkeit zwischen Objekten (*CSAddReachability* bzw. *CSRemoveReachability*). Die Operation *CSDeleteType* wird durch das Schema *CSPreDeleteType* erweitert, das garantiert, daß *absurd_type* nicht gelöscht werden kann. Das Schema *CSPostCreateObject* sorgt dafür, daß jedem Objekt die Kenntnis des Typs *absurd_type*, und somit indirekt jedes anderen Typs, mitgegeben wird. Ein Objekt darf jedoch nicht vom Typ *absurd_type* sein, welcher nur für die Spezifikation der globalen Sichtbarkeit notwendig ist.

CSPreDeleteType

$$
\begin{array}{l}
\Xi CompleteVisibility \\
type? : TYPE \\
\hline
type? \neq absurd\_type
\end{array}
$$

CSPreCreateObject

$\Xi CompleteVisibility$
$object\_type? : TYPE$

$object\_type? \neq absurd\_type$

CSPostCreateObject

$\Delta Visibility$
$\Xi CompleteVisibility$
$new\_object? : OBJECT$

$visible' = visible \cup \{v : TypeLink \mid v.obj = new\_object? \wedge$
$v.type = absurd\_type\}$

$CSCreateType \mathrel{\widehat=} CreateType$

$CSDeleteType \mathrel{\widehat=} CSPreDeleteType \fatsemi DeleteType$

$CSCreateObject \mathrel{\widehat=} CSPreCreateObject \fatsemi CreateObject \fatsemi CSPostCreateObject$

$CSDeleteObject \mathrel{\widehat=} DeleteObject$

$CSAddReachability \mathrel{\widehat=} AddReachability$

$CSRemoveReachability \mathrel{\widehat=} RemoveReachability$

5.2.2 Offenes Objektsystem

Ein *offenes Objektsystem* unterscheidet sich von einem geschlossenen Objektsystem in der Sichtbarkeit der Typen. Während bei einem geschlossenen Objektsystem gefordert wird, daß allen Objekten alle Typen bekannt sind, fällt diese Bedingung bei einem offenen Objektsystem weg. Der Zustandsraum ist identisch mit dem eines Objektgraphen (siehe Schema *OpenSystem*). Initial besteht das System lediglich aus dem Wurzelobjekt *root_obj* mit seinem Typ *root_type* (siehe Schema *InitOpenSystem*).

$OpenSystem \mathrel{\widehat=} ObjectGraph$

InitOpenSystem

$OpenSystem$
$root\_obj : OBJECT$
$root\_type : TYPE$

$objects = \{root\_obj\}$
$types = \{root\_type\}$
$has\_type = \{root\_obj \mapsto root\_type\}$
$is\_conform = \{root\_type \mapsto root\_type\}$
$visible = \{v : TypeLink \mid v.obj = root\_obj \wedge v.type = root\_type\}$
$reachable = \emptyset$

Zu den in einem offenen System möglichen Operationen gehören das Erzeugen und Löschen von Typen oder Objekten sowie der Änderung der Erreichbarkeit bzw. Sichtbarkeit. Alle Operationen entsprechen den zuvor vorgestellten dynamischen Schemata des Objektgraphen.

$$OSCreateType \mathrel{\widehat=} CreateType$$

$$OSDeleteType \mathrel{\widehat=} DeleteType$$

$$OSCreateObject \mathrel{\widehat=} CreateObject$$

$$OSDeleteObject \mathrel{\widehat=} DeleteObject$$

$$OSAddReachability \mathrel{\widehat=} AddReachability$$

$$OSRemoveReachability \mathrel{\widehat=} RemoveReachability$$

$$OSAddVisibility \mathrel{\widehat=} AddVisibility$$

$$OSRemoveVisibility \mathrel{\widehat=} RemoveVisibility$$

5.2.3 Vermittlungssystem

Dieser Abschnitt erweitert die formale Spezifikation eines offenen Objektsystems um einen Vermittler. Die Propagation der Erreichbarkeit sowie der Sichtbarkeit entspricht der Vermittlung von Instanzen bzw. von Typen. Unter einem *Vermittlungssystem* soll ein System verstanden werden, welches ein für den Vermittlungsvorgang dediziertes Objekt besitzt. Das ausgezeichnete Objekt übernimmt die Rolle eines Vermittlers, welches an seiner Schnittstelle den Import und Export von Objekten und Typen anbietet.

Das Schema *Trader* legt die notwendigen Erweiterungen fest, die den Zustandsraum eines Vermittlungssystems von dem eines offenen Systems unterscheidet. Ein ausgezeichnetes Objekt, durch den Bezeichner *trader* benannt, stellt den Vermittler dar. Dieses Objekt ist global bekannt und eine Invariante garantiert, daß jedes Objekt im System eine Referenz auf den Vermittler besitzt (d.h. der Vermittler ist von jedem Objekt aus erreichbar). Der Zustandsraum ist durch die Konjunktion der Schemata *OpenSystem* und *Trader* beschrieben. Der initiale Zustand *InitTradingSystem* erzeugt zunächst das Wurzelobjekt *root_obj* und den Vermittler *trader*, sowie deren Typen *root_type* und *trader_type*. Die beiden Typen stehen in keiner Subtyprelation. Das Wurzelobjekt erhält ferner eine Referenz auf den Vermittler, um die Invariante des Schemas *Trader* zu erfüllen.

Trader

$$TypeSystem$$
$$Reachability$$
$$trader : OBJECT$$
$$trader\_type : TYPE$$

$$trader \in objects \land trader\_type \in types \land has\_type(trader) = trader\_type$$
$$\forall o : objects \bullet (o \neq trader \Rightarrow (\exists r : reachable \bullet (r.to = trader \land r.from = o)))$$

$$TradingSystem \mathrel{\widehat=} OpenSystem \land Trader$$

InitTradingSystem

$$\begin{array}{l} TradingSystem \\ root\_obj : OBJECT \\ root\_type : TYPE \\ \hline objects = \{root\_obj, trader\} \\ types = \{root\_type, trader\_type\} \\ has\_type = \{root\_obj \mapsto root\_type, trader \mapsto trader\_type\} \\ is\_conform = \{root\_type \mapsto root\_type, trader\_type \mapsto trader\_type\} \\ visible = \{v : TypeLink \mid v.obj = root\_obj \land v.type = root\_type\} \cup \\ \qquad \{v : TypeLink \mid v.obj = trader \land v.type = trader\_type\} \\ reachable = \{r : ObjectLink \mid r.from = root\_obj \land r.to = trader \land \\ \qquad r.type = trader\_type\} \end{array}$$

Die Standardoperationen zum Anlegen und Löschen von Objekten bzw. Typen müssen durch Vor– und Nachbedingungen auf die Besonderheiten eines Vermittlungssystems angepaßt werden. Das Vermittlerobjekt mit seiner global bekannten Identität darf nicht gelöscht werden. Das betrifft sowohl die Instanz selbst, als auch dessen Typ (siehe *TSPreDeleteType* und *TSPreDeleteObject*). Ein neu erzeugtes Objekt erhält automatisch eine Referenz auf den Vermittler, d.h. die Erreichbarkeit um einen entsprechenden Eintrag erweitert (siehe *TSPostCreateObject*). Ferner muß gewährleistet werden, daß ein Objekt über seine ganze Lebensdauer hinweg diese Referenz nicht verliert (siehe *CheckRemoveReachability*).

TSPreDeleteType

$$\begin{array}{l} \Xi Trader \\ type? : TYPE \\ \hline type? \neq trader\_type \end{array}$$

TSPostCreateObject

$$\begin{array}{l} \Delta Reachability \\ \Delta Visibility \\ \Xi Trader \\ new\_object? : OBJECT \\ \hline reachable' = reachable \cup \{r : ObjectLink \mid r.from = new\_object? \land \\ \qquad r.to = trader \land r.type = has\_type(trader)\} \\ visible' = visible \cup \{v : TypeLink \mid v.obj = new\_object? \land \\ \qquad v.type = trader\_type\} \end{array}$$

TSPreDeleteObject

$\Xi TradingSystem$
$obj? : OBJECT$

$obj? \neq trader$

CheckRemoveReachability

$\Xi TradingSystem$
$link? : ObjectLink$

$link?.to \neq trader$

Der Vermittler bietet an seiner Schnittstelle Operationen für die Vermittlung von Diensten (d.h. Objekten) und Typen an, die im folgenden durch eigene Schemata formal spezifiziert werden. Sowohl der Import als auch der Export von Instanzen und Typen sind Spezialisierungen der Schemata *AddReachability* und *AddVisibility*. Während das Schema *AddReachability* die Basis für die Vermittlung von Instanzen bildet, übernimmt das Schema *AddVisibility* diese Aufgabe bei der Vermittlung von Typen. Die folgenden Vorbedingungen garantieren, daß die Vermittlung über das global bekannte Vermittlungsobjekt abgewickelt wird.

CheckExportInstance

$\Xi TradingSystem$
$link? : ObjectLink$

$link?.from = trader$

CheckImportInstance

$\Xi TradingSystem$
$link? : ObjectLink$

$\exists\, l : reachable \bullet (l.from = trader \land l.to = link?.to)$

CheckExportType

$\Xi TradingSystem$
$vis? : TypeLink$

$vis?.obj = trader$

CheckImportType

$\Xi TradingSystem$
$vis? : TypeLink$

$\exists\, v : visible \bullet (v.obj = trader \land v.type = vis?.type)$

Die in einem Vermittlungssystem erforderlichen Operationen sind durch die folgenden zehn dynamischen Schemata festgelegt. Bis auf die letzten vier korrespondieren die Operationen mit denen des offenen Systems modulo erweiterter Vorbedingungen. Die letzten vier spezifizieren das Verhalten des Vermittlerobjekts. Die Schemata *ExportInstance* und *ImportInstance* beschreiben die Vermittlung von Instanzen, während die Schemata *ExportType* und *ImportType* die Vermittlung von Typen formal spezifizieren.

$$TSCreateType \mathrel{\widehat=} CreateType$$

$$TSDeleteType \mathrel{\widehat=} TSPreDeleteType \fatsemi DeleteType$$

$$TSCreateObject \mathrel{\widehat=} CreateObject \fatsemi TSPostCreateObject$$

$$TSDeleteObject \mathrel{\widehat=} TSPreDeleteObject \fatsemi DeleteObject$$

$$TSAddReachability \mathrel{\widehat=} AddReachability$$

$$TSRemoveReachability \mathrel{\widehat=} CheckRemoveReachability \fatsemi RemoveReachability$$

$$ExportInstance \mathrel{\widehat=} CheckExportInstance \fatsemi AddReachability$$

$$ImportInstance \mathrel{\widehat=} CheckImportInstance \fatsemi AddReachability$$

$$ExportType \mathrel{\widehat=} CheckExportType \fatsemi AddVisibility$$

$$ImportType \mathrel{\widehat=} CheckImportType \fatsemi AddVisibility$$

5.3 Zusammenfassung

Eine formale Spezifikation liefert eine abstrakte Beschreibung eines Systems, indem es dessen Eigenschaften beschreibt (das *was*), ohne jedoch Aussagen über eine Implementation zu machen (das *wie*). Ein System wird beschrieben durch einen Zustandsraum, einem initialen Zustand und einer Menge von Transitionen, die auf dem Zustandsraum definiert sind. Eine korrekte Beschreibung gewährleistet, daß der initiale Zustand den Invarianten des Zustandsraums entspricht und die Transitionen das System von einem gültigen Zustand zu einem anderen gültigen Zustand überführt.

In dieser Arbeit wird zwischen der Vermittlung von Instanzen und Typen unterschieden. Auf Basis des Objektmodells läßt sich ein Vermittlungssystem formal als eine Folge von Objektgraphen beschreiben, bei der ein global bekanntes Objekt die Rolle des Vermittlers übernimmt.

Kapitel 6

Deklaratives Typsystem

Ziel dieses Kapitels ist die Definition eines Typsystems, welches ein syntaktisches Typsystem dahingehend erweitert, daß es Unterstützung für die *Open World Assumption* bietet. Wegen seiner Bedeutung bei der Spezifikation von operationalen Schnittstellen dient ein syntaktisches Typsystem als Ausgangspunkt für die Diskussionen in diesem Kapitel. Die *Open World Assumption*, die die Möglichkeit von nicht–injektiven Interpretationen zwischen Extension und Intension eines Typs fordert, kann nur mit Hilfe einer semantischen Typbeschreibungssprache umgesetzt werden. Die in diesem Kapitel vorgeschlagene Erweiterung eines syntaktischen Typsystems hat zum Ziel, semantische Typspezifikationen zu ermöglichen und gleichzeitig der *Open World Assumption* gerecht zu werden. Als Grundlage dafür dient die aus der Logik bekannte deklarative Semantik. Ein Charakteristikum der Erweiterung ist, daß dafür kein spezielles syntaktisches Typsystem vorausgesetzt wird. Mit dem hier vorgeschlagenen deklarativen Typsystem können beliebige syntaktische Typbeschreibungssprachen erweitert werden. Als Voraussetzung hierfür wird in Abschnitt 6.1 eine formale Darstellung eines Typsystems entwickelt. Die semantische Spezifikation eines Typs basiert auf der deklarativen Semantik, die in Form eines ausführlichen Beispiels in Abschnitt 6.2 eingeführt wird. Die formale Definition eines deklarativen Typsystems schließt sich in Abschnitt 6.3 an.

6.1 Formale Darstellung eines Typsystems

Um die nachfolgenden Diskussionen über eine Erweiterung eines syntaktischen Typsystems auf eine fundierte Grundlage zu stellen, wird zunächst ein Typsystem formal dargestellt. Dazu ist eine formale Notation für die Typbeschreibungssprache und der Typkonformität eines Typsystems notwendig. Ausgangspunkt für die formale Darstellung ist die Theorie der Grammatiken, die im folgenden vorgestellt wird. Basierend auf dieser Grundlage folgt die formale Darstellung eines Typsystems.

6.1.1 Grammatiken

Eine formale Sprache ist eine beliebige Menge L von Wörtern über einem festen Alphabet Σ, d.h. $L \subseteq \Sigma^*$. Formale Sprachen werden in vielfältiger Weise als Beschreibungsmittel eingesetzt (siehe [37]), beispielsweise für die Charakterisierung der Menge aller:

- gültigen C++ Programme
- HTML–Dokumente
- Typspezifikationen

Für die endliche Beschreibung einer formalen Sprache bieten sich zwei Methoden an. Es kann eine Vorschrift angegeben werden, die genau die Elemente der zu beschreibenden Sprache erzeugt (erzeugendes System). Ein anderes Verfahren besteht in einem Entscheidungsverfahren, welches zu jedem $x \in \Sigma^*$ angibt, ob $x \in L$ gilt (akzeptierendes System). Die Darstellung einer formalen Sprache durch eine Grammatik entspricht einem erzeugenden System. Diese Art der Charakterisierung einer formalen Sprache bietet zum einen den Vorteil einer strukturierten und übersichtlichen Darstellung. Zum anderen existieren zahlreiche Hilfswerkzeuge, die aus einer Grammatik automatisch für die maschinelle Verarbeitung einen Parser generieren (siehe [2]). Die in Abschnitt 3.4.1 beschriebene Generierung von Klienten– und Server–Stub wird beispielsweise typischerweise von Parser–Generatoren unterstützt. Alle formalen Sprachen in diesem und den folgenden Kapiteln werden in Form von Grammatiken definiert werden.

Definition 6.1 (Grammatik): Eine *Grammatik* ist ein 4–Tupel $G = (N, \Sigma, P, S)$.

- N endliche Menge der nichtterminalen Symbolen.
- Σ endliche Menge der terminalen Symbolen mit $N \cap \Sigma = \emptyset$.
- $P \subseteq V_0 \times V^*$ ist eine endliche Menge von Regeln (Produktionen) mit $V_0 = V^*NV^*$. Die Schreibweisen $(u, v) \in P$ und $u \to v$, bzw. $v \leftarrow u$ in P sind äquivalent.
- $S \in N$ ist das Startsymbol.

□

Schreibweise 6.1: Auf V^* ist die Relation $\Rightarrow_G$ erklärt durch $x \Rightarrow_G y$, falls es eine Regel $u \to v$ in P und Wörter $w, w' \in \Sigma^*$ gibt mit $x = wuw'$ und $y = wvw'$.

Die Relation $\overset{n}{\Rightarrow}$ für $n \in \mathbb{N}$ und $\overset{*}{\Rightarrow}$ sind definiert durch:

- $x \overset{n}{\Rightarrow} y$ falls es $w_0, w_1, \ldots, w_n$ gibt mit $x = w_0$, $y = w_n$, $w_i \Rightarrow w_{i+1}$ für $0 \leq i < n$.
- $x \overset{*}{\Rightarrow} y$ falls es ein $n \geq 0$ gibt mit $x \overset{n}{\Rightarrow} y$.

Die von G erzeugte Sprache ist $L(G) =_{df} \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$. Das Produkt zweier Sprachen ist $L_1 \cdot L_2 =_{df} \{uv \mid u \in L_1, v \in L_2\}$.

6.1.2 Typsysteme

Ein Typsystem setzt sich aus einer Typbeschreibungssprache und der Definition einer Typkonformität zusammen. Die Typbeschreibungssprache ist eine formale Sprache, die die gültigen Typspezifikationen eines Typraums festlegt. Jedes Objekt besitzt einen Typ, der auf einer extensionalen Ebene durch eine Typspezifikation repräsentiert wird. Die Typkonformität entspricht einer binären Relation über Typspezifikationen. Mit der obigen Definition einer Grammatik sei ein Typsystem wie folgt formal charakterisiert:

Definition 6.2 (Typsystem): Sei $G_T = (N, \Sigma_T, P, S)$ eine Grammatik mit $L_T =_{df} L(G_T)$. Ein *Typsystem* ist ein Tupel $\mathcal{T} = (L_T, \leq_T)$ mit:

- L_T einer Typbeschreibungssprache.
- $\leq_T \subseteq L_T \times L_T$ einer Typkonformitätsrelation.

□

Die formale Definition eines Typsystems beschränkt sich auf die Darstellungsebene. Die Intension eines Typs wird in der Definition nicht formalisiert. Die Menge L_T entspricht dem Typraum des Typsystems $\mathcal{T}$ und die einzelnen Elemente repräsentieren Extensionen in Form von Typspezifikationen. Eine Typkonformität $\leq_T$ läßt sich als Teilmenge des kartesischen Produkts $L_T \times L_T$ formal beschreiben. Die Typkonformität stellt genau zwei Typen in eine Konformitätsrelation. Da ein Typ konform zu mehreren Typen sein kann, ist die Typkonformität nicht durch eine Abbildung darstellbar. Die Wahl des Symbols $\leq_T$ für die Typkonformität deutet an, daß aufgrund des Austauschbarkeitsprinzips die Typkonformität eine Halbordnung auf der Menge der Typspezifikationen induziert.

6.2 Semantische Typspezifikationen durch kleinste Herbrand–Modelle

Die zentrale Idee des deklarativen Typsystems besteht darin, eine fest aber beliebige syntaktische Typbeschreibungssprache dergestalt zu erweitern, daß mit ihr semantische Typspezifikationen möglich sind. Der Umfang der semantischen Spezifikation beschränkt sich jedoch darauf, Mitglieder einer Typfamilie zu spezifizieren. Der Begriff der Typfamilie wird in Abschnitt 6.2.1 eingeführt. Die semantische Spezifikation basiert auf der deklarativen Semantik, deren Grundlagen in Abschnitt 6.2.2 vorgestellt werden. In Abschnitt 6.2.3 wird dann die Technik der semantischen Typspezifikation durch kleinste Herbrand–Modelle an einem konkreten Beispiel systematisch entwickelt.

6.2.1 Typfamilien

Typsysteme, die auf einer syntaktischen Typbeschreibungssprache beruhen, spielen in offenen verteilten Systemen eine besondere Rolle. Einerseits hilft die Spezifikation operationaler Schnittstellen die Überwindung der Heterogenität in Bezug auf Programmiersprachen

und Datendarstellungen. Andererseits bieten sie eine einfache und handhabbare Typisierung von Objektschnittstellen. Im folgenden bezeichne $\mathcal{T}_{IDL} = (L_{IDL}, \leq_{IDL})$ ein fest, aber beliebiges syntaktisches Typsystem, wie beispielsweise das in Kapitel 4 vorgestellte Typsystem von DCE oder CORBA. L_{IDL} ist die syntaktische Typbeschreibungssprache und $\leq_{IDL}$ die Typkonformität des syntaktischen Typsystems $\mathcal{T}_{IDL}$.

Die Spezifikation der operationalen Schnittstelle eines Objekts auf Basis eines syntaktischen Typsystems ist notwendige Voraussetzung, um auf der operationalen Ebene auf dessen Funktionalität zuzugreifen. Die Typkonformität eines syntaktischen Typsystems kann hierdurch lediglich auf einer operationalen Ebene betrachtet werden. Die Semantik einer syntaktischen Typspezifikation kann nur indirekt anhand der Wahl der Bezeichner der Methodennamen abgeleitet werden. Ein Programmierer assoziiert durch die Bezeichner indirekt eine intendierte Semantik der syntaktischen Typspezifikation, die jedoch zur Laufzeit von einer Objektinfrastruktur nicht überprüfbar ist. Das Austauschbarkeitsprinzip garantiert nur, daß die Nachrichten eines Dienstnutzers von einem zuvor vermittelten Dienstanbieter akzeptiert werden.

Da die syntaktische Typkonformität nur auf die Syntax der zu vergleichenden Typspezifikationen Bezug nehmen kann, muß durch geeignete Wahl der Bezeichner die Typkonformität gesteuert werden. Besitzen zwei Typen eine ähnliche operationale Schnittstelle, die jedoch nicht welchselseitig typkonform zueinander sind, so garantiert die Wahl unterschiedlicher Bezeichner, daß die Zugriffssicherheit nicht verletzt wird. Die bereits in Abschnitt 4.2.1 vorgestellten Schnittstellen eines *Bag* und eines *Stack* sollen dies verdeutlichen. Für die folgende Diskussion seien die beiden operationalen Schnittstellenspezifikationen auf Basis der CORBA–IDL zugrundegelegt:

```
interface Bag {
     void GetChar( out char c );
     void PutChar( in char c );
     bool IsEmpty();
};
```

```
interface Stack {
     void PopChar( out char c );
     void PushChar( in char c );
     bool IsEmpty();
};
```

In Abschnitt 4.2.1 wurde bereits darauf hingewiesen, daß unter der Berücksichtigung der Semantik nach dem Austauschbarkeitsprinzip ein *Stack* typkonform zu einem *Bag* ist. Wegen der unterschiedlichen Wahl der Bezeichner für die Methodennamen, sind bzgl. einer syntaktischen Typkonformität die Typspezifikationen eines *Bag* und eines *Stack* wechselseitig nicht typkonform zueinander. Als erster Schritt zu einer semantischen Typkonformität müssen Mengen von Typspezifikationen zu *Typfamilien* zusammengezogen werden. Mitglieder einer Typfamilie zeichnen sich trotz abweichender Semantik ihrer Methoden durch eine ähnliche operationale Schnittstelle aus (siehe Abbildung 6.1). In Bezug auf einen *Bag* und einen *Stack* ähneln sich die Methoden `GetChar()` und `PopChar()` (Auslesen von Elementen) sowie `PutChar()` und `PushChar()` (Speichern von Elementen).

Jede Typfamilie ist durch eine *generische operationale Schnittstelle* charakterisiert, die einen homogenen Zugriff auf einer operationalen Ebene aller ihrer Mitglieder definiert. Bei einem *Bag* und einem *Stack* handelt es sich um *Container*, die beliebige Elemente speichern und wieder auslesen können. Die Spezifikation der generischen operationalen Schnittstelle basiert selbst wieder auf einem syntaktischen Typsystem. Die operationale Schnittstelle der Typfamilie der *Container* ließe sich beispielsweise mit Hilfe der CORBA–IDL wie folgt spezifizieren:

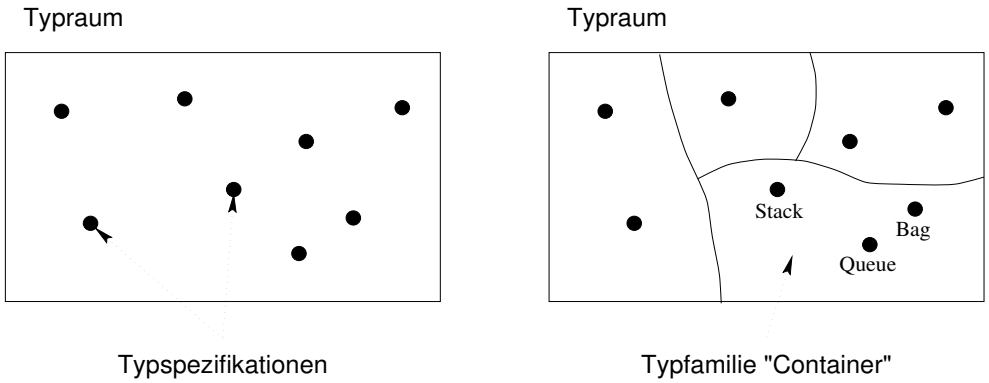


Abbildung 6.1: Partitionierung eines Typraums in Typfamilien.

```
interface Container {
     void RetrieveChar( out char c );
     void DepositChar( in char c );
     bool IsEmpty();
};
```

Die Typspezifikation `Container` beschreibt eine generische operationale Schnittstelle zu allen Mitgliedern der Typfamilie. Die rein syntaktische Spezifikation der operationalen Schnittstelle ist jedoch nicht mehr ausreichend, um zwischen den Mitgliedern einer Typfamilie zu unterscheiden. Die dem im folgenden beschriebenen deklarativen Typsystem zugrundeliegende Idee ist, die „fehlende“ Spezifikation in Form eines eingeschränkten prädikatenlogischen Programms an die syntaktische Typspezifikation `Container` anzuhängen. Dadurch besitzt das deklarative Typsystem eine semantische Typbeschreibungssprache.

6.2.2 Grundlagen der deklarativen Semantik

Die Menge der Objekte, deren Typen zu einer Typfamilie gehören, besitzen eine generische operationale Schnittstelle. Wie im letzten Abschnitt bereits gezeigt wurde, läßt sich diese generische operationale Schnittstelle ebenfalls als Spezifikation einer syntaktischen Schnittstellenbeschreibungssprache repräsentieren. Eine Typfamilie wird somit auf Basis eines syntaktischen Typsystems spezifiziert. Da eine Typfamilie jedoch von einer Menge von Typen abstrahiert, ist eine syntaktische Typspezifikation einer Typfamilie nicht ausreichend, um ein spezielles Mitglied innerhalb der Typfamilie zu benennen.

Die „fehlende“ Spezifikation, die eindeutig neben der Typfamilie noch ein Mitglied benennt, soll auf einer semantischen Beschreibung beruhen. Das resultierende Typsystem ist hybrid in dem Sinne, daß eine Spezifikation seiner Typbeschreibungssprache sowohl eine Beschreibung einer operationalen Schnittstelle als auch die Semantik des Typs beinhaltet. Eine Erweiterung einer syntaktischen Typbeschreibungssprache wurde bereits in verschiedenen Arbeiten vorgeschlagen (siehe beispielsweise [6], [58] oder [53]). Das in diesem Kapitel vorgestellte deklarative Typsystem erlaubt semantische Spezifikationen, die sich in mehreren Aspekten von diesen Ansätzen unterscheiden:

- Art der semantischen Spezifikation: die semantische Spezifikation eines Typs basiert auf der deklarativen Semantik.
- Umfang der semantischen Spezifikation: anstatt die „vollständige“ Semantik eines Typs zu spezifizieren, wird mit der deklarativen Semantik nur ein Typ relativ zu einer Typfamilie spezifiziert.
- Integration der semantischen Spezifikation: die deklarative Spezifikation nimmt keinen Bezug auf die zugrunde liegende operationale Schnittstellenspezifikation. Jedes syntaktische Typsystem kann zu einem deklarativen Typsystem erweitert werden.

Gerade der letzte Punkt resultiert in der Notwendigkeit, die semantische Spezifikation nicht in Form von Vor– und Nachbedingungen durchzuführen, wie es in Abschnitt 4.2.1 dargestellt wurde. Der Grund dafür ist, daß bei dieser Form der semantischen Spezifikation auf Teile der syntaktischen Spezifikation Bezug genommen wird, beispielsweise bei den Ein– und Ausgabeparametern der Methoden der syntaktischen Spezifikation. Die semantische Spezifikation auf Basis der deklarativen Semantik wird vielmehr als ein *Etikett* an die syntaktische Typspezifikation angehängt. Der Begriff Etikett soll unterstreichen, daß bei der Konstruktion einer semantischen Typbeschreibungssprache die Syntax der zugrundeliegenden syntaktischen Typbeschreibungssprache nicht modifiziert, sondern lediglich erweitert wird (Etikettierung).

Der Vorteil der deklarativen Semantik liegt in ihrer Eigenschaft begründet, daß sie nicht–injektive Interpretationen zwischen Extension und Intension einer Spezifikation erlaubt. Diese Eigenschaft ist Voraussetzung für die Erfüllung der *Open World Assumption*. Die deklarative Semantik basiert auf der Definition eines definiten Programms (siehe [59]):

Definition 6.3 (Definites Programm): Ein *Term* ist induktiv wie folgt definiert:

- Eine Variable ist ein Term.
- Eine Konstante ist ein Term.
- Ist f eine n–stellige Funktion und $t_1, \ldots, t_n$ sind Terme, dann ist $f(t_1, \ldots, t_n)$ ein Term.

Ist p ein n–stelliges Prädikatsymbol und $t_1, \ldots, t_n$ sind Terme, dann ist $p(t_1, \ldots, t_n)$ eine *atomare Formel*. Eine *Grundinstanz* ist eine atomre Formel, die keine Variablen enthält. Eine *definite Programmklausel* ist ein Ausdruck der Form $\forall\, x_1 \ldots \forall\, x_s(A \leftarrow B_1 \wedge \ldots \wedge B_n)$, wobei $A, B_1, \ldots, B_n$ atomare Formeln sind und $x_1, \ldots, x_s$ alle in den atomaren Formeln vorkommenden Variablen. Im folgenden wird eine definite Programmklausel abkürzend geschrieben als $A \leftarrow B_1, \ldots, B_n$. Eine *Einheitsklausel* ist eine definite Programmklausel der Form $A \leftarrow$, d.h. eine Klausel mit leerem Rumpf. Ein *definites Programm* ist eine endliche Menge von definiten Programmklauseln. Die Menge aller Programmklauseln eines definiten Programms mit dem gleichen Prädikatsymbol P als Kopf ist die *Definition von* P. Ein definites Programm ist *hierarchisch*, wenn eine Abbildung der Prädikatsymbole nach $\mathbb{N}$ existiert, so daß für jede definite Programmklausel $A \leftarrow B_1, \ldots, B_n$ der Wert der Prädikate B_i ist kleiner als der Wert von A bzgl. der Abbildung ist. □

Die informelle Semantik einer Klausel $A \leftarrow B_1, \ldots, B_n$ ist: „für jede Belegung der in der Klausel vorkommenden Variablen, wenn $B_1, \ldots, B_n$ wahr sind, dann ist auch A wahr“. Die logische Konsequenz eines definiten Programms läßt sich als Menge aller Grundinstanzen der Prädikate definieren, die aus dem Programm gefolgert werden können. Ist P ein definites Programm, so gehört das Prädikat A zu der logischen Konsequenz genau dann, wenn $P \models A$ gilt. Diese Schreibweise wird als *semantische Folgerung* bezeichnet. $P \models A$ besagt, daß jede Interpretation, die Modell für P ist, auch ein Modell für A ist.

Während ein definites Programm eine Extension repräsentiert, charakterisiert seine logische Konsequenz die Intension. Ein Vorteil eines definiten Programms ist, daß wenn dieses Programm ein Modell besitzt, es auch ein sogenanntes Herbrand–Modell besitzt. Es genügt somit, eine Diskussion über die formalen Eigenschaften von definiten Programmen auf Herbrand–Modelle zu beschränken. Das Herbrand–Modell eines definiten Programms wird über folgende Definition formal eingeführt.

Definition 6.4 (Herbrand–Modell): Sei L eine prädikatenlogische Sprache erster Ordnung. Das *Herbrand–Universum* U_L von L ist die Menge aller Grundterme, die mit den Funktions– und Konstantensymbolen aus L gebildet werden können. Die *Herbrand–Basis* B_L von L ist die Menge aller Grundatome, die aus den Prädikatsymbolen aus L mit den Grundtermen des Herbrand–Universums U_L als Parameter gebildet werden können. Eine *Vor–Interpretation* einer prädikatenlogischen Sprache L erster Ordnung besteht aus:

- einer nicht leeren Menge D; der Wertebereich der Vor–Interpretation.
- einer Zuordnung für jedes Konstantensymbol in L mit einem Element aus D.
- einer Abbildung von D^n nach D für jedes n–stellige Funktionssymbol in L.

Eine *Herbrand–Interpretation* von L basiert auf einer Vor–Interpretation mit:

- der Wertebereich der Vor–Interpretation ist das Herbrand–Universum U_L.
- die Konstanten in L werden sich selbst zugeordnet.
- jedem n–stelligen Funktionssymbol f in L wird eine Abbildung von $(U_L)^n$ nach U_L zugeordnet, die definiert ist durch $(t_1, \ldots, t_n) \rightarrow f(t_1, \ldots, t_n)$.

Eine Herbrand–Interpretation I eines definiten Programms P ist ein *Herbrand–Modell*, wenn es ein Modell für P ist. Sei P ein definites Programm und $\{M_i\}_{i \in I}$ eine nicht leere Menge von Herbrand–Modellen von P. Dann ist $\bigcap_{i \in I} M_i$ das *kleinste Herbrand–Modell* von P, bezeichnet als M_P. □

Die Herbrand–Basis definiert den Wertebereich, die Prädikate durch eine Interpretation zugeordnet werden können. Die Interpretation ist eine Abbildung von Prädikaten auf Elemente der Herbrand–Basis. Durch das Konstruktionsprinzip der Herbrand–Basis, die jedem Prädikat sich selbst zuordnet, genügt es, für eine Interpretation eine Teilmenge der Herbrand–Basis anzugeben. D.h. die logische Konsequenz eines definiten Programms ist eine Teilmenge der Herbrand–Basis. Wie mit jeder Herbrand–Interpretation ist das kleinste Herbrand–Modell ebenfalls eine Teilmenge der Herbrand–Basis ($M_P \subseteq B_L$). Obwohl ein Programmierer eine andere Interpretation im Sinn haben kann, ist M_P jedoch

eine „natürliche“ Interpretation in dem Sinne, daß die Grundinstanzen in M_P genau diejenigen sind, die logische Konsequenz des Programms P sind, d.h. $A \in M_P$ genau dann, wenn $P \models A$.

6.2.3 Semantisches Etikett

In Bezug auf die im letzten Abschnitt vorgestellte Typfamilie der *Container* muß bei einer deklarativen semantischen Spezifikation überlegt werden, wodurch sich die einzelnen Mitglieder voneinander unterscheiden. Eine semantische Spezifikation des Verhaltens aller Methoden der Typspezifikation `Container` aus dem letzten Abschnitt ist nicht notwendig, da beispielsweise das Verhalten der Methode `IsEmpty()` bei allen Mitgliedern identisch ist. Die Mitglieder der Typfamilie *Container* unterscheiden sich bzgl. der Auslesereihenfolge der gespeicherten Elemente. Ein semantisches Etikett muß daher genau diesen Unterschied in Form eines definiten Programms spezifizieren.

Um eine bestimmte Auslesereihenfolge über ein definites Programm zu spezifizieren, muß ein Prädikat eingeführt werden, welches den Effekt des Auslesens eines Elements aus einem *Container* beschreibt. Dies kann beispielsweise mit einem drei–stelligen Prädikat erreicht werden, welches genau den Zustandsübergang mit den Ausgabeparametern beschreibt, der sich aus einmaliger Anwendung der `RetrieveChar()` Methode eines *Container* ergibt. Es ist somit notwendig, den Zustand eines *Container* formal zu beschreiben. Dabei ist zu bemerken, daß die Modellierung des Zustands nicht an die tatsächliche Struktur des Zustands eines *Container* gekoppelt sein muß. Gemäß der Schnittstellenspezifikation `Container` des letzten Abschnitts vermag eine Instanz dieser Typfamilie Werte vom Typ `char` mit einem Wertebereich $[0 \ldots 2^8 - 1]$ zu speichern. Um jedoch die Auslesereihenfolge formal zu spezifizieren, genügt es, den Wertebereich auf zwei unterschiedliche Werte einzuschränken.

Der Zustand eines *Container* wird per Definition durch einen prädikatenlogischen Term modelliert. Die Konstante ϵ repräsentiert einen leeren *Container* und die ein–stelligen Funktionssymbole f und g zwei unterschiedliche Werte, die ein *Container* speichern kann. Ein Term, der sich aus diesen drei Symbolen zusammensetzt, repräsentiert einen Zustand des *Container*. Das Herbrand–Universum U_L beinhaltet alle möglichen Zustände, die ein *Container* annehmen kann. U_L ist nicht endlich, was für den *Container* bedeutet, daß er beliebig viele Elemente aufnehmen kann. Obwohl es sich hierbei um keine reale Annahme handelt, sei nochmals darauf hingewiesen, daß lediglich die Auslesereihenfolge Gegenstand der Spezifikation ist.

Der Term $f(f(g(\epsilon)))$ repräsentiert einen Zustand, bei dem der *Container* zweimal den Wert f und einmal den Wert g beinhaltet. Wiederum per Definition sei festgelegt, daß tiefer geschachtelte Terme zu einem früheren Zeitpunkt in den *Container* eingefügt wurden. Für den Term $f(f(g(\epsilon)))$ heißt dies, daß der Wert g zuerst in den *Container* eingefügt wurde. Um die Semantik der Einfügeoperation eines *Container* über ein definites Programm zu spezifizieren, muß zunächst ein Prädikatsymbol eingeführt werden, das im folgenden *put* bezeichnet wird. Mit den bisherigen Konventionen läßt sich die Methode `DepositChar()` der Schnittstelle `Container` wie folgt über ein definites Programm spezifizieren:

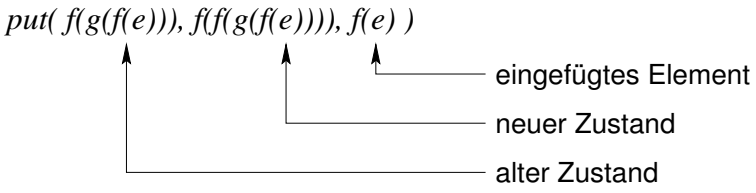


Abbildung 6.2: Bedeutung der Argumente des *put*–Prädikats.

$$put(x, f(x), f(\epsilon)) \quad \leftarrow \tag{6.1}$$
$$put(x, g(x), g(\epsilon)) \quad \leftarrow \tag{6.2}$$

Die informelle Semantik des Prädikats *put* ist, daß wenn ein Element (drittes Argument) zu einem *Container* (erstes Argument) hinzugefügt wird, das Ergebnis ein *Container* mit dem zusätzlichen Element ist (zweites Argument). Eine Instanz des Prädikats *put* beschreibt hierdurch einen Zustandswechsel. Eine korrekte Grundinstanz des Prädikats ist beispielsweise $put(f(g(f(\epsilon))), f(f(g(f(\epsilon)))), f(\epsilon))$ (siehe auch Abbildung 6.2). In diesem Fall wird ein f zu einem *Container* hinzugefügt, der bereits zwei f's und ein g enthält. Eine ungültige Grundinstanz, die nicht Element des kleinsten Herbrand–Modells des oben stehenden definiten Programms ist, ist dagegen $put(f(\epsilon), f(\epsilon), g(\epsilon))$. Obwohl das Prädikat *put* den Sachverhalt formal spezifiziert, daß später eingefügte Elemente „außen“ an einen Term angefügt werden, ist es nicht notwendig, es in das semantische Etikett aufzunehmen. Es wird angenommen, daß alle Mitglieder der Typfamilie *Container* die gleiche Semantik bzgl. der Methode `DepositChar()` aufweisen.

Um den Unterschied zwischen den verschiedenen Mitgliedern einer Typfamilie in Form eines definiten Programms zu spezifizieren, wird ein neues Prädikat *get* eingeführt. Es handelt sich dabei analog zu dem Prädikat *put* um ein drei–stelliges Prädikat, dessen erstes Argument den alten *Container*–Inhalt, das zweite Argument den neuen Inhalt und das dritte Argument das ausgelesene Element repräsentiert. Eine gültige Grundinstanz dieses Prädikats ist beispielsweise $get(f(g(f(\epsilon))), f(f(\epsilon)), g(\epsilon))$. Hier wird aus einem *Container*, der zwei f's und ein g enthält, das g ausgelesen. Eine ungültige Grundinstanz des Prädikats *get* ist $get(f(\epsilon), f(\epsilon), g(\epsilon))$.

Das Mitglied *Bag* der Typfamilie *Container* unterscheidet sich von den anderen Mitglieder dieser Typfamilie durch seine nicht–deterministische Auslesemethode. Ein definites Programm, welches alle Grundinstanzen des Prädikats *get* erzeugt, die gerade dieses Verhalten reflektieren, ist:

$$P_B : get(f(x), x, f(\epsilon)) \quad \leftarrow \tag{6.3}$$
$$get(g(x), x, g(\epsilon)) \quad \leftarrow \tag{6.4}$$
$$get(f(x), f(y), z) \quad \leftarrow \quad get(x, y, z) \tag{6.5}$$
$$get(g(x), g(y), z) \quad \leftarrow \quad get(x, y, z) \tag{6.6}$$

Die Grundinstanzen des kleinsten Herbrand–Modells von P_B zeichnen sich dadurch aus, daß jedes Element eines *Container* potentiell ausgelesen werden kann. Die folgen-

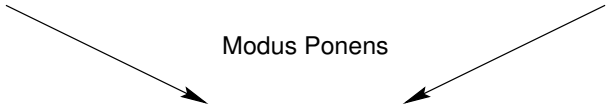
de Ableitung zeigt, daß die Grundinstanz $get(f(g(f(\epsilon))), f(f(\epsilon)), g(\epsilon))$, die im kleinsten Herbrand–Modell M_{P_B} von P_B enthalten ist, mit Hilfe der Spezialisierung von Prädikaten, der Modus–Ponens Regel und den Klauseln aus P_B abgeleitet werden kann:

Klausel (6.5): Ersetze: x durch $g(f(\epsilon))$
y durch $f(\epsilon)$
z durch $g(\epsilon)$

$get(f(g(f(\epsilon))), f(f(\epsilon)), g(\epsilon)) \leftarrow get(g(f(\epsilon)), f(\epsilon), g(\epsilon))$

Klausel (6.4): Ersetze: x durch $f(\epsilon)$

$get(g(f(\epsilon)), f(\epsilon), g(\epsilon)) \leftarrow$



$$get(f(g(f(\epsilon))), f(f(\epsilon)), g(\epsilon)) \leftarrow$$

Das definite Programm P_B teilt die Herbrand–Basis in zwei disjunkte Teilmengen. Die erste enthält die logische Konsequenz des Programms P_B, die identisch mit dem kleinsten Herbrand–Modell M_{P_B} ist. Die andere Teilmenge enthält diejenigen Grundinstanzen des Prädikats *get*, welche bzgl. der intendierten Semantik des Prädikats keine sinnvolle Interpretation zulassen. Die Grundinstanz $get(f(g(\epsilon)), f(\epsilon), g(\epsilon))$ ist Element von M_{P_B} und Ausdruck der nicht–deterministischen Auslesemethode eines *Bag*. Ein *Stack* schränkt die Auslesereihenfolge gemäß dem LIFO–Prinzip ein. Die Grundinstanz $get(f(g(\epsilon)), f(\epsilon), g(\epsilon))$ repräsentiert hierdurch kein gültiges Verhalten für einen *Stack*, der nur das zuletzt eingefügte Element auslesen kann. Das folgende definite Programm P_S beschreibt die gültige Auslesereihenfolge eines *Stack*:

$$P_S : get(f(x), x, f(\epsilon)) \quad \leftarrow \tag{6.7}$$
$$get(g(x), x, g(\epsilon)) \quad \leftarrow \tag{6.8}$$

Die beiden definiten Klauseln des Programms P_S stimmen mit den ersten beiden Klauseln des Programms P_B überein. Für die kleinsten Herbrand–Modelle gilt entsprechend die Teilmengenbeziehung $M_{P_S} \subset M_{P_B} \subset B_L$. Dem kleinsten Herbrand–Modell des *Stack* fehlen diejenigen Grundinstanzen, die nicht das zuletzt gespeicherte Element auslesen. Das Austauschbarkeitsprinzip, welches in Abschnitt 4.2.1 auf das Eliminieren von Nicht–Determinismen im Verhalten definiert wurde, drückt sich bei der deklarativen Semantik durch eine Teilmengenbeziehung der kleinsten Herbrand–Modelle aus. Deterministisches Verhalten ist gegenüber nicht–deterministischem Verhalten eingeschränkt, was in „kleineren“ Herbrand–Modellen resultiert.

Eine *Queue* gehört ebenfalls zu der Typfamilie der *Container*. Im Unterschied zu einem *Stack* genügt die Auslesereihenfolge einer *Queue* dem FIFO–Prinzip. In Bezug auf die semantische Spezifikation durch ein definites Programm können Elemente nur aus dem zutiefst geschachtelten Term entnommen werden. Das folgende Programm P_Q besitzt als logische Konsequenz gerade die Grundinstanzen des *get*–Prädikats, die dem FIFO–Prinzip genügen:

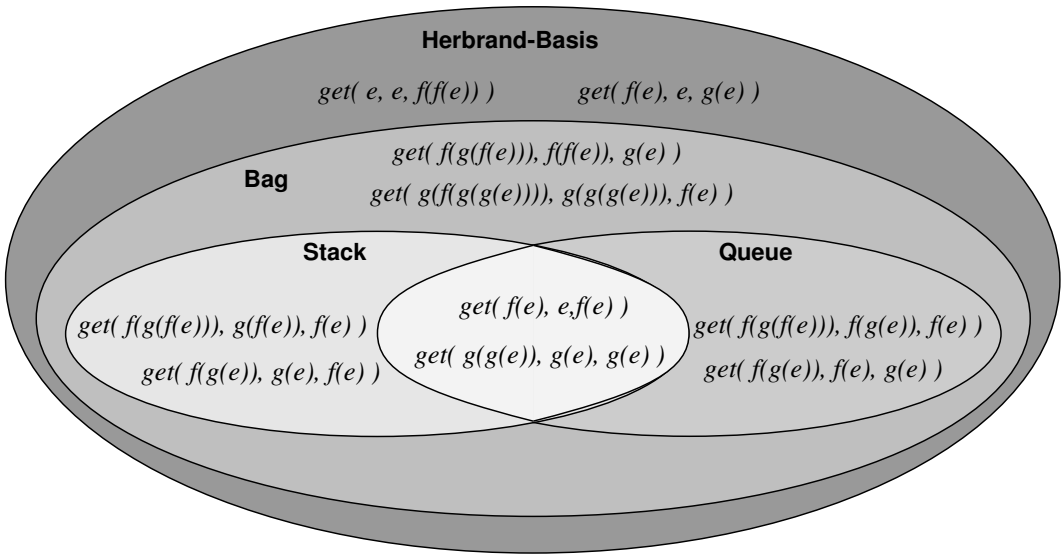


Abbildung 6.3: Beziehungen zwischen den kleinsten Herbrand–Modellen von P_B, P_S und P_Q.

$$P_Q : get(f(\epsilon), \epsilon, f(\epsilon)) \leftarrow \tag{6.9}$$
$$get(g(\epsilon), \epsilon, g(\epsilon)) \leftarrow \tag{6.10}$$
$$get(f(x), f(y), z) \leftarrow get(x, y, z) \tag{6.11}$$
$$get(g(x), g(y), z) \leftarrow get(x, y, z) \tag{6.12}$$

Das kleinste Herbrand–Modell M_{P_Q} von P_Q ist ebenfalls eine Teilmenge von M_{P_B}. Aus dieser Tatsache läßt sich ableiten, daß die *Queue* typkonform zu einem *Bag* ist. Die kleinsten Herbrand–Modelle aller bis jetzt diskutierten definiten Programme stehen in den nachstehenden Teilmengenbeziehungen (siehe auch Abbildung 6.3):

1. $M_{P_B} \subset B_L$
2. $M_{P_S} \subset M_{P_B}$
3. $M_{P_Q} \subset M_{P_B}$
4. $M_{P_S} \cap M_{P_Q} \neq \emptyset$
5. $M_{P_S} \neq M_{P_Q}$

Die Extension eines deklarativen Typs besteht aus einer operationalen Schnittstellenspezifikation und einem semantischen Etikett. Die Notwendigkeit der *a–priori*–Standardisierung von Typen in einem offenen verteilten System betrifft die Spezifikation der generischen operationalen Schnittstelle einer Typfamilie, sowie die intendierte Semantik einer ebenfalls festzulegenden Herbrand–Basis. Jeder Typfamilie ist eine eigene

Herbrand–Basis zugeordnet. Die Grundinstanzen, die Elemente der Herbrand–Basis sind, müssen ausdrucksstark genug sein, um die Unterschiede zwischen allen Mitgliedern der zugehörigen Typfamilie zu erklären. Ein konkretes Mitglied einer Typfamilie wird durch das semantische Etikett spezifiziert, was in Form eines definiten Programms angegeben werden muß. Das definite Programm als Teil der Extension einer deklarativen Typspezifikation beschreibt die Intension als Teilmenge der Herbrand–Basis. Die Notwendigkeit der *a–priori*–Standardisierung von Typspezifikationen in offenen verteilten Systemen bezieht sich *nicht* auf die syntaktische Struktur des semantischen Etiketts. Folgendes definite Programm $P_{S'}$ repräsentiert beispielsweise eine alternative Formulierung des LIFO–Verhaltens eines *Stack*:

$$P_{S'} : get(x, y, f(\epsilon)) \leftarrow equal(x, f(y)) \tag{6.13}$$
$$get(x, y, g(\epsilon)) \leftarrow equal(x, g(y)) \tag{6.14}$$
$$equal(x, x) \leftarrow \tag{6.15}$$

Obwohl die Programme P_S und $P_{S'}$ syntaktisch voneinander unterschieden sind, besitzen ihre kleinesten Herbrand–Modelle dennoch die gleichen Grundinstanzen des Prädikats *get*. Das Programm $P_{S'}$ besitzt ein weiteres zweistelliges Prädikatsymbol *equal*, welches die syntaktische Gleichheit von zwei Termen definiert. Die *Open World Assumption*, die die Möglichkeit einer nicht–injektiven Interpretation zwischen Extension und Intension von Typen zuläßt, ist durch die definiten Programme P_S und $P_{S'}$ nachgewiesen. Dieses Beispiel zeigt auch, daß bei der Spezifikation des semantischen Etiketts „Hilfsprädikate“ erlaubt sein sollten, die jedoch bei der Überprüfung der Typkonformität keinen Einfluß haben.

6.3 Definition des deklarativen Typsystems

Als zentrales Ergebnis dieses Kapitels wird in diesem Abschnitt eine formale Definition eines deklarativen Typsystems angegeben. Ausgangspunkt ist ein syntaktisches Typsystem, dessen Typbeschreibungssprache erweitert wird. Die Erweiterung ermöglicht die Spezifikation eines semantischen Etiketts in Form eines definiten Programms und führt somit keine Änderungen in der zugrundeliegenden syntaktischen Typbeschreibungssprache nach sich. $\mathcal{T}_{IDL}$ bezeichne im folgenden ein fest aber beliebiges syntaktisches Typsystem.

Definition 6.5 (Sprache für ein semantisches Etikett): Die Sprache L_E für ein *semantisches Etikett* ist definiert durch die Grammatik $G_E = (N_E, \Sigma_E, P_E, S_E)$, mit $N_E =$ { `dt_extension`, `header`, `pred_list`, `body`, `clause`, `clause_head`, `clause_body`, `expr`, `expr_list`, `id` }, Σ_E dem ISO8859–1 Zeichensatz und $S_E =$ `dt_extension`. Sei $\Upsilon =_{df} \Sigma_E \setminus$ { `"("`, `")"`, `","`, `":"`, `"-"` }. Die Menge der Produktionen P_E ist gegeben durch:

```
dt_extension   ← header body
header         ← "RELEVANT PREDICATES" pred_list
pred_list      ← id
```

```
pred_list      ← pred_list "," id
body           ← clause
body           ← body clause
clause         ← clause_head ":-" clause_body
clause_head    ← expr
clause_body    ← ϵ
clause_body    ← expr_list
expr           ← id
expr           ← id "(" expr_list ")"
expr_list      ← expr
expr_list      ← expr "," expr_list
id             ← ϒ+
```

□

Ein semantisches Etikett nach Definition 6.5 besteht aus einer Deklaration der relevanten Prädikate und einer Menge von definiten Programmklauseln. Die relevanten Prädikate folgen dem Schlüsselwort `RELEVANT PREDICATES` und benennen diejenigen Prädikate, die bei der Überprüfung der Typkonformität berücksichtigt werden sollen. Alle anderen Prädikate dienen nur als Hilfsprädikate. Die Syntax der definiten Programmklauseln ist an die der Sprache Prolog angelehnt (siehe [100]). Das terminale Symbol `:-` übernimmt die Rolle der Implikation „←“.

Definition 6.6 (Deklarative Erweiterung einer syntaktischen Typbeschreibungssprache): Sei L_{IDL} eine syntaktische Typbeschreibungssprache. Es gelte für alle $w \in L_{IDL}$ und jeder Zerlegung von $w = uv$, ist $v \notin L_E$. Eine *deklarative Erweiterung* der Sprache L_{IDL} ist eine Sprache L_{DT}, die sich aus dem Produkt der Sprachen L_{IDL} und L_E ergibt, d.h. $L_{DT} =_{df} L_{IDL} \cdot L_E$. □

Die Bedingung, daß für jede Zerlegung eines Wortes aus L_{IDL} der Suffix nicht ein Wort der Sprache L_E ist, garantiert, daß keine Zweideutigkeiten durch das semantische Etikett entstehen. Eine Typspezifikation auf Basis einer deklarativen Erweiterung kann hierdurch eindeutig in einen Präfix und einen Suffix aufgeteilt werden. Der Präfix korrespondiert mit einer operationalen Schnittstellenspezifikation und der Suffix mit dem semantischen Etikett. Wird beispielsweise als zugrundeliegende syntaktische Typbeschreibungssprache die CORBA–IDL angenommen, so entspricht die folgende Typspezifikation einem *Stack*:

```
interface Container {
     void RetrieveChar( out char c );
     void DepositChar( in char c );
     bool IsEmpty();
};
RELEVANT PREDICATES get
get( X, Y, f(e) ) :- equal( X, f(Y) )
get( X, Y, g(e) ) :- equal( X, g(Y) )
equal( X, X ) :-
```

Die operationale Schnittstellenspezifikation beschreibt die bereits vorher angegebene generische Schnittstelle der Typfamilie der *Container*. Das semantische Etikett entspricht

dem definiten Programm $P_{S'}$. Das Prädikat *get* ist das einzige relevante Prädikat der Spezifikation; *equal* ist somit ein Hilfsprädikat. Die Groß– und Kleinschreibung von Bezeichnern im semantischen Etikett unterscheidet zwischen Variablen und Konstanten und entspricht den Konventionen in Prolog (siehe [100]). Fängt ein Bezeichner mit einem Großbuchstaben an, handelt es sich um eine Variable, ansonsten um eine Konstante bzw. Funktion.

6.3.1 Modelltheoretische Definition der Typkonformität

Nach der Definition der deklarativen Erweiterung einer syntaktischen Typbeschreibungssprache, folgt eine formale Definition der deklarativen Typkonformität. Diese Definition ist geprägt von zwei unterschiedlichen Sichtweisen. Die in diesem Abschnitt vorgestellte modelltheoretische Definition führt den Begriff der deklarativen Typkonformität ein, ohne ein Entscheidungsverfahren anzugeben. Entsprechend den Grundprinzipien der deklarativen Semantik, beschäftigt sich dieser Abschnitt zunächst mit dem *was* und der folgende Abschnitt mit dem *wie* der deklarativen Typkonformität.

Die modelltheoretische Definition ist aus den Beispielen des Abschnitts 6.2.3 motiviert. Ein definites Programm induziert ein kleinstes Herbrand–Modell. Das Austauschbarkeitsprinzip des Inklusions–Polymorphismus resultiert in einem Vergleich zweier kleinster Herbrand–Modelle. Die semantische Spezifikation in Form eines Etiketts muß dabei lediglich den Unterschied zwischen Mitgliedern einer Typfamilie beschreiben. Für die Spezifikation können Hilfsprädikate eingeführt werden, die bei einem Vergleich der kleinsten Herbrand–Modelle nicht berücksichtigt werden dürfen.

Die Definition der deklarativen Typkonformität erfolgt in zwei Schritten. Zunächst müssen die in den deklarativen Typspezifikationen enthaltenen operationalen Schnittstellenspezifikationen verglichen werden. Dieser Vergleich beruht auf der Definition der Typkonformität des zugrundeliegenden syntaktischen Typsystems. Erweisen sich die beiden operationalen Schnittstellen als typkonform (d.h. es werden typkonforme Mitglieder einer Typfamilie gesucht), werden anschließend deren semantische Etiketten verglichen. Diese informelle Beschreibung der deklarativen Typkonformität resultiert in der folgenden formalen Definition:

Definition 6.7 (Deklarative Typkonformität $\leq_{DT}$): Sei $\mathcal{T}_{IDL} = (L_{IDL}, \leq_{IDL})$ ein syntaktisches Typsystem und L_{DT} eine deklarative Erweiterung der syntaktischen Typbeschreibungssprache L_{IDL}. Sei ferner $u_1, u_2 \in L_{DT}$ wobei $u_i = t_i p_i$ mit $t_i \in L_{IDL}$, $p_i \in L_E$ für $i = 1, 2$. Weiterhin sei R_i die Menge der in p_i aufgeführten relevanten Prädikate und M_{p_i} das kleinste Herbrand–Modell von p_i. Sei $M'_{p_i} =_{df} \{q \in M_{p_i} \mid$ das Prädikatsymbol der Grundinstanz q ist Element von $R_i\}$. Es gilt $t_1 \leq_{DT} t_2$ genau dann, wenn:

1. $t_1 \leq_{IDL} t_2$
2. $R_1 = R_2$
3. $M'_{p_1} \subseteq M'_{p_2}$

□

Die Typkonformität des deklarativen Typsystems $\mathcal{T}_{DT} = (L_{DT}, \leq_{DT})$ basiert auf der Typkonformität des zugrundeliegenden syntaktischen Typsystems. Die Notwendigkeit der *a–priori*–Standardisierung von Typspezifikationen eines deklarativen Typsystems erstreckt sich auf den Teil der Spezifikation, der die operationale Schnittstellenbeschreibung enthält, sowie der Definition einer Herbrand–Basis. Was nicht *a priori* standardisiert werden muß, ist die Art und Weise, wie eine Teilmenge der Herbrand–Basis durch ein definites Programm charakterisiert wird.

6.3.2 Operationale Definition der Typkonformität

Die modelltheoretische Definition der deklarativen Typkonformität legt fest, *wann* zwei Typspezifikationen typkonform zueinander sind. Die Definition enthält keinen Entscheidungsalgorithmus, *wie* die Typkonformität festzustellen ist. Die operationale Definition der deklarativen Typkonformität ergänzt die modelltheoretische Definition durch einen Entscheidungsalgorithmus. Es wird dabei vorausgesetzt, daß die Typkonformität des zugrundeliegenden syntaktischen Typsystems entscheidbar ist. In diesem Abschnitt wird ein Entscheidungsverfahren für Punkt 3 der Definition 6.7 vorgestellt.

Der Kern der Definition 6.7 beruht auf einem Vergleich zweier kleinster Herbrand–Modelle. Um die deklarative Typkonformität zu überprüfen, muß eine Teilmengenbeziehung zwischen den beiden kleinsten Herbrand–Modellen festgestellt werden. Das Resolutionsprinzip von Prolog muß im Vergleich dazu nur die Mengenzugehörigkeit eines Elements zu einem kleinsten Herbrand–Modell testen (siehe [100]). Das Resolutionsprinzip berechnet, ob eine vorgegebene Zielklausel in dem kleinsten Herbrand–Modell eines definiten Programms enthalten ist. Der Vergleich zeigt, daß die Überprüfung der Teilmengenbeziehung zweier kleinster Herbrand–Modelle wesentlich komplexer als das Resolutionsprinzip ist. In [64] wurde gezeigt, daß das Problem der Teilmengenbeziehung der kleinsten Herbrand–Modelle zweier beliebiger definiter Programme nicht entscheidbar ist. Damit ist auch die deklarative Typkonformität aus Definition 6.7 nicht entscheidbar.

Um dennoch eine entscheidbare, operationale Definition zu präsentieren, muß die Ausdruckskraft eines definiten Programms eingeschränkt werden. Im folgenden wird ein Entscheidungsverfahren für den Fall angegeben, daß die beiden definiten Programme hierarchisch sind (siehe Definition 6.3). Ein hierarchisches Programm erlaubt keine Rekursionen. Der Vorteil eines hierarchischen definiten Programms ist, daß es in ein semantisch äquivalentes definites Programm transformiert werden kann, welches nur aus Einheitsklauseln besteht. Die Teilmengenbeziehung der kleinsten Herbrand–Modelle von zwei definiten hierarchischen Programmen kann hierdurch mit Hilfe des Subsumptionsprinzips überprüft werden. Zunächst folgen die notwendigen Definitionen des allgemeinsten Unifikators und des Subsumptionsprinzips.

Definition 6.8 (Allgemeinster Unifikator): Eine *Substitution* θ ist eine endliche Menge der Form $\{v_1/t_1, \ldots, v_n/t_n\}$, wobei jedes v_i eine Variable, jedes t_i ein Term unterschiedlich von v_i ist, und die Variablen $v_1, \ldots, v_n$ voneinander unterschieden sind. Jedes Element v_i/t_i ist eine *Bindung* für v_i. Sei $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$ eine Substitution und E ein Ausdruck. Dann bezeichnet $E\theta$ einen Ausdruck, bei dem jedes Vorkommen der Variable v_i in E gleichzeitig durch den Term t_i ausgetauscht wird. Sei S eine endliche Menge von Ausdrücken. Eine Substitution θ ist ein *Unifikator* von S,

wenn $S\theta$ genau ein Element enthält. Ein Unifikator θ ist ein *allgemeinster Unifikator* (a.U.), wenn für jeden Unifikator σ von S eine Substitution γ existiert mit $\sigma = \theta\gamma$. □

Satz 6.1 (Subsumptionsprinzip): Seien P_1 und P_2 zwei definite Programme, die ausschließlich Einheitsklauseln enthalten. Es gilt $M_{P_1} \subseteq M_{P_2}$ genau dann, wenn für jede Klausel $A \leftarrow$ aus P_1 eine Klausel $B \leftarrow$ aus P_2 und ein allgemeinster Unifikator σ existiert, mit $A = B\sigma$. ■

Algorithmus:	Transformation eines hierarchischen Programms in eine Menge von Einheitsklauseln.
Eingabe:	P: ein endliches, hierarchisches, definites Programm.
Ausgabe:	P_u: semantisch äquivalentes Programm, welches nur Einheitsklauseln enthält.

1. $k := 0;\ P_0 := P$
2. Wenn alle Klauseln in P_k Einheitsklauseln sind, gehe zu Schritt 6.
3. Sei $A \leftarrow B_1, \ldots, B_n$ eine beliebige nicht–Einheitsklausel aus P_k, bei der die Definition aller Prädikatssymbole B_i nur aus Einheitsklauseln in P_k besteht.
4. Sei $\tilde{P} := \{A' \leftarrow |$ es existiert ein a.U. σ, so daß $A' = A\sigma$ und für jedes B_i existiert eine Einheitsklausel $\{C \leftarrow\} \in P_k$ mit $B_i\sigma = C\sigma\}$. Beachte das $\tilde{P}$ endlich ist.
5. Sei $P_{k+1} := (P_k \setminus \{A \leftarrow B_1, \ldots, B_n\}) \cup \tilde{P};\ k := k + 1;$ gehe zu Schritt 2.
6. Sei $P_u := P_k$; Stop.

Der Algorithmus terminiert, da in Schritt 5 immer eine nicht–Einheitsklausel pro Iteration eliminiert wird. Die Schritte 3—5 ersetzen alle nicht–Einheitsklauseln durch eine endliche Menge semantisch äquivalenter Einheitsklauseln. Der Transformationsalgorithmus übersetzt beispielsweise das definite Programm $P_{S'}$ aus Abschnitt 6.2.3 in folgendes semantisch äquivalente definite Programm P_u:

$$P_u : get(f(x), x, f(\epsilon)) \leftarrow \tag{6.16}$$
$$get(g(x), x, g(\epsilon)) \leftarrow \tag{6.17}$$
$$equal(x, x) \leftarrow \tag{6.18}$$

Das resultierende definite Programm P_u ist identisch mit dem Programm P_S bzgl. dem relevanten Prädikat *get*. Der formale Beweis, daß der oben angegebene Transformationsalgorithmus korrekt ist, benötigt einen Satz aus der Logik. Der Satz charakterisiert das kleinste Herbrand–Modell als Fixpunkt des monotonen Operators T_P, der wie folgt definiert ist.

Definition 6.9 (T_P–Operator): Sei P ein definites Programm. Die Abbildung $T_P : Pot(B_P) \rightarrow Pot(B_P)$ ist wie folgt definiert: Sei I eine Herbrand–Interpretation.

Dann ist $T_P(I) =_{df} \{A \in B_P \mid A \leftarrow B_1, \ldots, B_n$ ist eine Grundinstanz einer Klausel in P und $\{B_1, \ldots, B_n\} \subseteq I\}$. □

Definition 6.10 (Kleinster Fixpunkt): Sei L ein vollständiger Verband und $T : L \to L$ eine Abbildung. Ein $a \in L$ ist ein *kleinster Fixpunkt* (lfp) von T, wenn a ein Fixpunkt ist (d.h. $T(a) = a$) und für alle Fixpunkte b von T gilt $a \leq b$. □

Satz 6.2 (Fixpunktcharakterisierung des kleinsten Herbrand–Modells): Sei P ein definites Programm. Dann ist $M_P = lfp(T_P) = T_P \uparrow \omega$. ■

Der Vorteil der Fixpunktcharakterisierung des kleinsten Herbrand–Modells ist darin begründet, daß für formale Aussagen, wie den nachfolgenden Beweis, daß Induktionsprinzip angewandt werden kann. Der folgende Beweis weißt nach, daß die einmalige Anwendung der Schritte 3—5 des Transformationsalgorithmus das definite Programm semantisch nicht verändert.

Satz 6.3 (Korrektheit des Transformationsprogramms): Sei P ein hierarisches Programm, das nicht ausschließlich Einheitsklauseln enthält. Sei P' aus P gemäß der Konstruktionsvorschrift der Schritte 3—5 des Transformationsalgorithmus hervorgegangen. Sei $A \leftarrow A_1, \ldots, A_n$ die in Schritt 3 ausgewählte Klausel. Zu zeigen ist, daß $lfp(T_P) = lfp(T_{P'})$.

Beweis: Es ist zu zeigen, daß $lfp(T_{P'}) \subseteq lfp(T_P)$ und $lfp(T_P) \subseteq lfp(T_{P'})$.

1.) $lfp(T_{P'}) \subseteq lfp(T_P)$

Es genügt zu zeigen, daß $T_{P'} \uparrow 1 \subseteq T_P \uparrow \omega$, da die Klausel $A \leftarrow A_1, \ldots, A_n$ nur in P enthalten ist und P' mehr Einheitsklauseln als P besitzt. Sei $B \in T_{P'} \uparrow 1$.

Fall 1: Für einen a.U. σ und eine Einheitsklausel $\{C \leftarrow\} \in P' \setminus \tilde{P}$ ist $B = C\sigma \Longrightarrow \{C \leftarrow\} \in P \Longrightarrow B \in T_P \uparrow 1 \Longrightarrow B \in T_P \uparrow \omega$.

Fall 2: Für einen a.U. σ und eine Einheitsklausel $\{C \leftarrow\} \in \tilde{P}$ is $B = C\sigma$. Da nach Konstruktionsvorschrift von $\tilde{P}$ in Schritt 4 $\{C \leftarrow\} \in \tilde{P}$, existiert ein a.U. τ mit $C = A\tau$. Weiterhin sind alle Atome im Rumpf von $A\tau \leftarrow A_1\tau, \ldots, A_n\tau$ mit einer Einheitsklausel aus P unifizierbar (nach Schritt 4). Alle Grundinstanzen dieser Einheitsklauseln sind enthalten in $T_P \uparrow 1 \Longrightarrow$ alle Grundinstanzen von $A\tau$ sind Element von $T_P \uparrow 2$. Da $B = C\sigma$ und $C = A\tau \Longrightarrow B = (A\tau)\sigma \Longrightarrow B \in T_P \uparrow 2 \Longrightarrow B \in T_P \uparrow \omega$.

2.) $lfp(T_P) \subseteq lfp(T_{P'})$

Induktion über n: $T_P \uparrow n \subseteq T_{P'} \uparrow \omega$

$n = 1$: Alle Einheitsklauseln in P sind enthalten in $P' \Longrightarrow T_P \uparrow 1 \subseteq T_{P'} \uparrow 1 \subseteq T_{P'} \uparrow \omega$

$n \rightarrow n+1$: I.H. $T_P \uparrow n \subseteq T_{P'} \uparrow \omega$
Sei $B \in T_P \uparrow (n+1)$. Zu zeigen ist, daß $B \in T_{P'} \uparrow \omega$.
$B \in T_P \uparrow (n+1) \Longrightarrow$ es gibt eine Grundinstanz einer Klausel in P, $B \leftarrow B_1, \ldots, B_n$ mit $\{B_1, \ldots, B_n\} \subseteq T_P \uparrow n \Longrightarrow \{B_1, \ldots, B_n\} \subseteq T_{P'} \uparrow \omega$.

Fall 1: $B \leftarrow B_1, \ldots, B_n$ ist Grundinstanz einer Klausel in $P' \Longrightarrow B \in T_{P'} \uparrow \omega$.

Fall 2: $B \leftarrow B_1, \ldots, B_n$ ist eine Grundinstanz von $A \leftarrow A_1, \ldots, A_n$ für einen a.U. σ. Somit ist keine Grundinstanz einer Klausel in P'. Gemäß der Konstruktionsvorschrift in Schritt 4 sind $B_1, \ldots, B_n$ alle Grundinstanzen bestimmter Einheitsklauseln in P (und somit auch von P'). B ist somit Grundinstanz einer Einheitsklausel in $\tilde{P}$ (und somit in P') $\Longrightarrow B \in T_{P'} \uparrow 1 \Longrightarrow B \in T_{P'} \uparrow \omega$.

■

Sind zwei hierarchische Programme mit Hilfe des oben angegebenen Transformationsalgorithmus in Mengen von Einheitsklauseln übersetzt worden, kann die Teilmengenbeziehung der kleinsten Herbrand–Modelle mit dem Subsumptionsprinzip überprüft werden. Zuvor müssen von den Mengen der Einheitsklauseln die nicht–relevanten Prädikate entfernt werden, um der Bedingung 3 der Definition 6.7 zu genügen.

6.4 Zusammenfassung

Ein deklaratives Typsystem erweitert ein syntaktisches Typsystem. Die Erweiterung erlaubt neben der Spezifikation einer operationalen Schnittstelle, die eine generische Schnittstelle zu einer Typfamilie beschreibt, eine semantische Spezifikation. Die semantische Spezifikation in Form eines Etiketts basierend auf der deklaratiken Semantik beschreibt relativ zu einer Typfamilie ein Mitglied dieser Typfamilie. Die *Open World Assumption*, die nicht–injektive Interpretationen zwischen Extension und Intension eines Typs zuläßt, wird bzgl. des semantischen Etiketts erfüllt. Zwei unterschiedliche Extensionen in Form von definiten Programmen können die gleiche Intension in Form eines kleinsten Herbrand–Modells besitzen. Die deklarative Typkonformität beruht auf einer Teilmengenbeziehung von zwei kleinsten Herbrand–Modellen. Dieses Problem ist für allgemeine definite Programme nicht entscheidbar. Für hierarchische definite Programme, bei denen die Ausdruckskraft eingeschränkt ist, kann jedoch ein Entscheidungsverfahren angegeben werden.

Die Notwendigkeit der *a–priori*–Standardisierung von Typspezifikationen in einem deklarativen Typsystem betrifft die operationale Schnittstelle und die Struktur der Herbrand–Basis für jede Typfamilie. Da die *Open World Assumption* hier nur innerhalb einer Typfamilie unterstützt wird, erscheint es sinnvoll, Typfamilien mit möglichst vielen Mitgliedern zu definieren. Würden jeweils nur wenige Mitglieder pro Typfamilie

existieren, wäre der Vorteil der deklarativen Semantik nicht mehr gegeben. Die in Abschnitt 4.2.2 eingeführte Technik der generischen Dienstnutzer würde in *einer* Typfamilie mit *einer* generischen operationalen Schnittstelle resultieren. Die Aufgabe des sematischen Etiketts bestünde hier darin, zwischen *allen* Typspezifikationen des Typraums zu unterscheiden. Das kann jedoch aufgrund der Unentscheidbarkeit der deklarativen Typkonformität nicht geleistet werden. Dieser Mangel führt u.a. zu dem im folgenden Kapitel eingeführten wissensbasierten Typsystem.

Kapitel 7

Wissensbasiertes Typsystem

In offenen verteilten Systemen treten zunehmend Anwender in den Rollen der Dienstnutzer und –anbieter auf. Die Einbeziehung von Anwendern bei der Dienstvermittlung bedingt einerseits eine benutzerorientierte Typbeschreibungssprache und andererseits eine maschinelle Unterstützung bei der Vermittlung von Typen. Ziel dieses Kapitels ist die Entwicklung eines *wissensbasierten Typsystems*, welches die Abstraktionsebene eines Anwenders unterstützt. Im Unterschied zu dem im letzten Kapitel vorgestellten deklarativen Typsystem zeichnet sich ein wissensbasiertes Typsystem durch eine benutzerorientierte Typbeschreibungssprache aus. Eine Möglichkeit die Anwender bei der Spezifikation von Diensttypen zu unterstützen ist durch den Einsatz einer Wissensrepräsentationstechnik. Eine Wissensrepräsentationstechnik, deren Ausdruckskraft an die der Prädikatenlogik angelehnt ist, bietet eine Notation, die die Kodierung beliebiger Aussagen ermöglicht, wie beispielsweise eine Diensttypspezifikation. Eine Wissensrepräsentationstechnik, die für benutzerorientierte Diensttypspezifikationen geeignet ist, sind *Konzeptgraphen*.

Vor der Definition eines wissensbasierten Typsystems auf Basis der Konzeptgraphen wird in Abschnitt 7.1 zunächst auf die Diensttypvermittlung über Typhierarchien eingegangen. Bei dieser Variante traversiert der Anwender eine Menge von Typen, die in einer Hierarchie angeordnet sind, um den gewünschten Diensttyp zu ermittlen. Die Schwachstellen dieser Form der Diensttypvermittlung, die aus heutiger Sicht die einzige Form der Diensttypvermittlung für Anwender darstellt, bildet den Ausgangspunkt für eine Typbeschreibungssprache mit Konzeptgraphen, deren Syntax und Semantik Gegenstand des Abschnitts 7.2 ist. Für den Vermittlungsvorgang in offenen verteilten Systemen ist neben der Spezifikation von Diensttypen die Definition einer Typkonformität von Relevanz. Das Verhältnis in dem ein Konzeptgraph in Beziehung zu einem anderen steht, wird durch eine Ontologie bestimmt, die Abschnitt 7.3 beschrieben ist. Auf Basis der Ontologie können Vergleichsregeln definiert werden, die die Typkonformität zwischen Konzeptgraphen überprüfen. Die allgemeine Struktur einer Vergleichsregel wird in Abschnitt 7.4 vorgestellt. Eine Zusammenfassung in Abschnitt 7.5 beschließt das Kapitel.

7.1 Diensttypvermittlung über Typhierarchien

Ausgangspunkt der Betrachtungen dieses Kapitels ist die Annahme, daß in offenen verteilten Systemen *Anwender* aktiv an der Dienstvermittlung teilnehmen. Anwender treten

in den Rollen von Dienstnutzern und Dienstanbietern auf, so daß ein Typsystem für offene verteilte Systeme eine Unterstützung dieser Gruppe bieten muß. Bei der Diskussion bestehender Typsysteme wurde bereits in Abschnitt 4.2.2 eine Typbeschreibungssprache auf Basis von Schlüsselwörtern vorgestellt. Obwohl diese Technik dem Anwender in seiner Vorstellungswelt über Typspezifikationen entgegenkommt, reicht die Ausdruckskraft für die zu erwartende Entwicklung offener verteilter Systeme hin zu elektronischen Dienstmärkten nicht aus.

Eine Erweiterung der Diensttypvermittlung basiert auf *Typhierarchien*, wie sie beispielsweise im Kontext der ODP–Vermittlungsfunktionalität vorgeschlagen wird (siehe [17]). Die Menge aller im System existierenden Diensttypen werden in einer Typhierarchie angeordnet. Die Knoten entsprechen den Diensttypen und die Kanten fest definierten Beziehungen zwischen diesen. Der Export eines Typs erweitert die Typhierarchie, indem der Typ an geeigneter Stelle in die Hierarchie eingefügt wird. Der Import eines Typs basiert auf der Traversierung der Typhierarchie. Die Typhierarchie bietet einem Anwender in der Rolle des Dienstnutzers allgemeine Einstiegspunkte an, die mit dessen vagen Vorstellungen über einen Diensttyp korrespondieren. Während der Traversierung der Typhierarchie werden die vagen Vorstellungen des Anwenders konkretisiert, so daß letztendlich die Sichtbarkeit auf den gewünschten Diensttyp hergestellt wird.

Der Begriff der Typhierarchie deutet bereits an, daß die hierarchische Anordnung der Diensttypen dem Anwender die Navigation der Typhierarchie erleichtert. Zwei Beispiele für die Organisation einer Typhierarchie bestehen in der Klassifikation der Diensttypen nach *geographischen* oder *funktionalen* Aspekten. Eine Typhierarchie basierend auf einer geographischen Klassifikation bietet dem Anwender auf jeder Ebene eine Auswahl unterschiedlicher, sich gegenseitig ausschließender Regionen. Je tiefer innerhalb der Hierarchie abgestiegen wird, desto spezifischer wird eine Region eingegrenzt. Diese Vorgehensweise setzt voraus, daß der Anwender mit einem Diensttyp eine geographische Lokation assoziiert. In der Regel besitzt ein Dienstnutzer jedoch lediglich eine Vorstellung über die gewünschte Funktionalität.

Die funktionale Klassifikation von Diensttypen zu einer Hierarchie ordnet die Diensttypen gemäß ihrer Funktionalität an. Grundlage für die Anordnung bildet der Inklusions–Polymorphismus. Ein Diensttyp T_1 steht unterhalb eines anderen Diensttyps T_2 entlang eines Astes der Hierarchie genau dann, wenn in jedem Kontext, in dem Typ T_2 erlaubt ist, auch der Typ T_1 erlaubt ist. Dieses Austauschbarkeitsprinzip, welches auf einer polymorphen Beziehung zwischen den Diensttypen beruht, resultiert wegen seiner transitiven und azyklischen Eigenschaft in einer hierarchischen Anordnung der Diensttypen. Die Größe der Typhierarchie, im Sinne von Anzahl der Knoten (d.h. Diensttypen), reflektiert das qualitative Dienstangebot. Unterschiedliche Diensttypen werden auf unterschiedliche Knoten der Typhierarchie abgebildet. Unter der Annahme eines sowohl quantitativ, als auch qualitativ großen Dienstangebots übersteigt der Umfang einer Typhierarchie in offenen verteilten Systemen den einer isolierten Klassenbibliothek.

Zu den Typhierarchien offener verteilter Systeme existieren Parallelen aus dem Bereich der Linguistik. Wörter aus einem Wörterbuch stehen durch ihre Bedeutung in vielfältigen Beziehungen zueinander. Eine Beziehungsart, die der funktionalen Spezialisierung als Ordnungsprinzip von Typhierarchien entspricht, ist die *Hyponym–Beziehung* (siehe [71]). Die linguistische Definition basiert auf der Akzeptanz im Sprachgebrauch

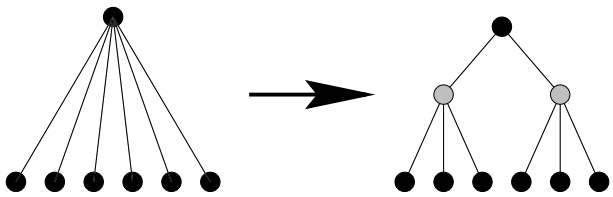


Abbildung 7.1: Erweiterung einer Typhierarchie.

eines bestimmten Ausdrucks. Ein Wort w_1 ist ein Hyponym eines Worts w_2 genau dann, wenn im allgemeinen Sprachgebrauch der Satz *„ein w_1 ist ein w_2“* akzeptiert wird (der Satz *„ein Drucker ist ein Gerät“* entspricht beispielsweise diesem Empfinden). Konkrete Projekte, die den Wortschatz einer Sprache gemäß der Hyponym–Beziehung klassifiziert haben, beweisen, daß die Tiefe der daraus resultierenden Hierarchie nicht sehr groß wird (siehe [71] oder [98]). Die Konsequenz daraus ist, daß ein Knoten innerhalb der Hierarchie im Durchschnitt über viele Nachfolger verfügt. Die Aufgabe eines Dienstnutzers bei der Traversierung der Hierarchie besteht darin, sich aufgrund eines Knotens und dessen unmittelbarer Nachfolger, für einen Ast zu entscheiden. Je mehr Nachfolger ein Knoten besitzt, desto schwieriger ist die Aufgabe, sich für einen Ast zu entscheiden.

In [39] wird aus diesem Grund vorgeschlagen, die Hierarchie künstlich tiefer zu machen, indem neue innere Knoten eingefügt werden. In Abbildung 7.1 besitzt der Wurzelknoten vor der Erweiterung sechs Nachfolger. Durch Einfügen von zwei (in der Abbildung grau gefärbten) Zwischenknoten besitzt der Wurzelknoten zwei, und die Zwischenknoten jeweils drei unmittelbare Nachfolger. Das Ergebnis ist eine Typhierarchie, die mehr Ebenen besitzt und deren Knoten weniger unmittelbare Nachfolger haben. Einem Dienstnutzer sollte somit die Entscheidung, in welchem Teilbaum er den zu suchenden Diensttyp vermutet, leichter fallen. Der Vergleich mit bereits bestehenden linguistischen Datenbanken zeigt jedoch, daß dieses Argument nicht zutreffend ist, wie im folgenden gezeigt wird.

Abbildung 7.2 zeigt einen Auszug der Hyponym–Hierarchie des WordNet–Wörterbuchs (siehe [70]). An den beiden Enden des Astes befinden sich die englischen Substantive *cat* und *animal*[1]. Die Anordnung der beiden Substantive leitet sich von dem semantisch sinnvollen englischen Satz *„a cat is an animal“* ab, gemäß der Definition der Hyponym–Beziehung. Wie der Abbildung zu entnehmen ist, befinden sich sechs Zwischenknoten entlang des Astes von *cat* nach *animal*. Viele der innerhalb des Astes auftretenden Begriffe dienen allein dem Zweck, die Hierarchie tiefer zu machen, um eine umfassende Klassifikation aller Substantive zu erzielen. Der Anwender startet seine Navigation bei dem ihm bekannten abstrakten Einstiegspunkten (in diesem Fall *animal*) und muß auf jeder Ebene der Hierarchie entscheiden, in welchem Teilast er den gewünschten Diensttyp vermutet. Es wird davon ausgegangen, daß der Anwender von dem Ziel seiner Suche (hier *cat*) zwar eine Vorstellung besitzt, aber es nicht direkt mit einem Begriff assoziieren kann. Obwohl jeder der Substantive entlang eines Astes eine feststehende Bedeutung besitzt, wird vom Anwender eine umfangreiche Kenntnis der Hierarchie gefordert[2].

[1] In der folgenden Diskussion werden die Begriffe *Substantiv* und *Diensttyp* synonym benutzt.

[2] Das Datenvolumen in WordNet, dem der in Abbildung 7.2 dargestellte Ast entnommen ist, besitzt mehr als 64.000 bedeutungsverschiedene Substantive der englischen Sprache mit einer maximalen Tiefe

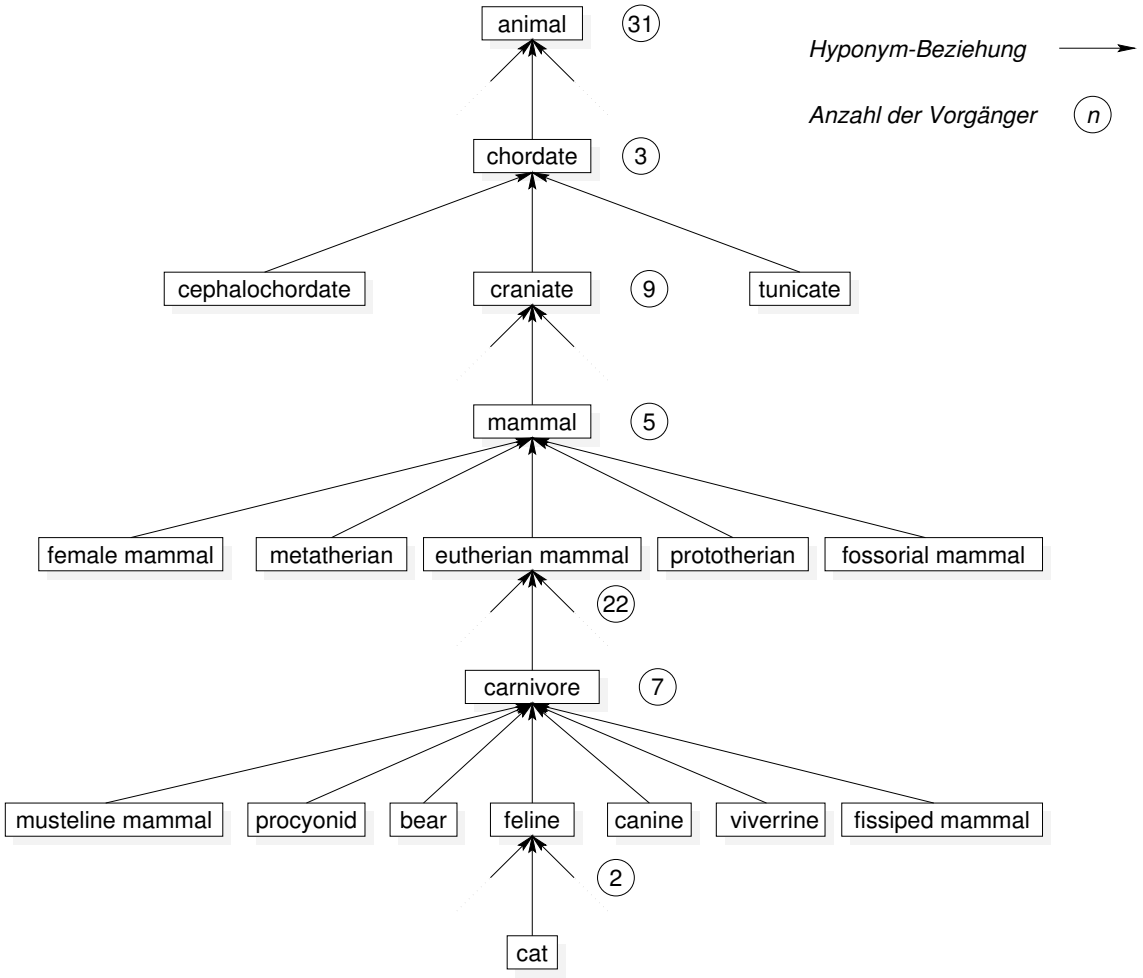


Abbildung 7.2: Auszug der Hyponym–Hierarchie in WordNet.

Dem Anwender muß die Semantik der Diensttypen bekannt sein, die ihm während der Traversierung der Hierarchie begegnen, um entscheiden zu können, in welchem Teilbaum er den gewünschten Diensttyp vermutet. Die Diensttypvermittlung auf Basis der Traversierung einer Typhierarchie ist nur für geschlossene Systeme mit einer abgeschlossenen Menge von Diensttypen sinnvoll. Im Kontext eines offenen verteilten Systems mit einer hohen Fluktuation in der Quantität und Qualität des Dienstangebots ist eine Typhierarchie nicht geeignet. Der Grund hierfür liegt in der Semantik der Zwischenknoten. Um die Diensttypen ausreichend zu klassifizieren, müßen Zwischenknoten eingeführt werden, deren Bedeutung einem Anwender bekannt sein muß.

Die Navigation eines Anwenders in einer Typhierarchie gleicht der ursprünglichen Problematik der Dienstvermittlung, bei der von einer aktuellen Position (d.h. Knoten der Hierarchie) nicht entschieden werden kann, wie ein gewünschter Diensttyp zu erreichen ist. Obwohl die Anordnung der Hierarchie auf einem Klassifikationsprinzip beruht, ähnelt die Situation dem in Abbildung 3.2 auf Seite 24 dargestellten Ausgangspunkt. Die Objekte der Objektebene korrespondieren mit Typen und die Referenzen zwischen Objekten den Kanten des Klassifikationsschemas. Bei der Traversierung des „Typgraphen“ ist ebenfalls das fehlende *a–priori*–Wissen über die Struktur des Graphen ein Problem, sowie bei der Vermittlung innerhalb des Objektgraphen.

7.2 Wissensrepräsentation durch Konzeptgraphen

Bei den Anforderungen an Typsysteme für offene verteilte Systeme wurde die Notwendigkeit der Möglichkeit von semantischen Typspezifikationen hervorgehoben. Die Semantik ist Voraussetzung bei der Zuordnung zweier Extensionen, die die gleiche Intension besitzen. Obwohl das in Kapitel 6 vorgestellte deklarative Typsystem diese Anforderung erfüllt, ist es wegen der Unentscheidbarkeit und der fehlenden Anwendernähe der Notation lediglich von theoretischem Interesse. Die weit verbreiteten Typbeschreibungssprachen, deren Spezifikationen lediglich aus einer Menge von Schlüsselwörtern bestehen, sind ebenfalls nicht geeignet, die zu erwartende hohe Dynamik im Dienstangebot zu unterstützen.

Die angesprochenen Schwächen der bisher vorgestellten Typsysteme legen einen Übergang zu einer *wissensbasierten Typbeschreibungssprache* nahe. Der Begriff *Wissensbasiert* resultiert in einer Unterscheidung zwischen *Daten* und *Wissen* (siehe [22]). Eine unstrukturierte Datenmenge wird hierbei nach einem Selektions–, Transformations– und Interpretationsprozeß in Wissen umgewandelt (siehe Abbildung 7.3). In der Selektionsphase werden relevante von nicht–relevanten Daten getrennt. Dieser Schritt dient der Reduktion des Datenvolumens. Die nachfolgende Transformation bereitet die Daten in ein einheitliches Format auf. Im letzten Schritt erhalten die Daten durch eine Interpretation eine Bedeutung, voraus dann das Wissen hervorgeht.

Die oben beschriebenen Phasen können bei der Gewinnung des Wissens mehrmals durchlaufen werden. Ein *wissensbasiertes System* unterstützt den Anwender bei der Ver-

---

der Hyponym–Hierarchie von 16 Ebenen. Der Suchdienst Yahoo!, der Dienstangebote des *World Wide Web* registiert, teilt die Angebote in ca. 20.000 Kategorien auf. Dieser Vergleich zeigt, daß bei einer weiteren Zunahme der Diensttypen im *World Wide Web*, eine Typhierarchie mit ähnlich spezialisierten Zwischenknoten angereichert werden muß.

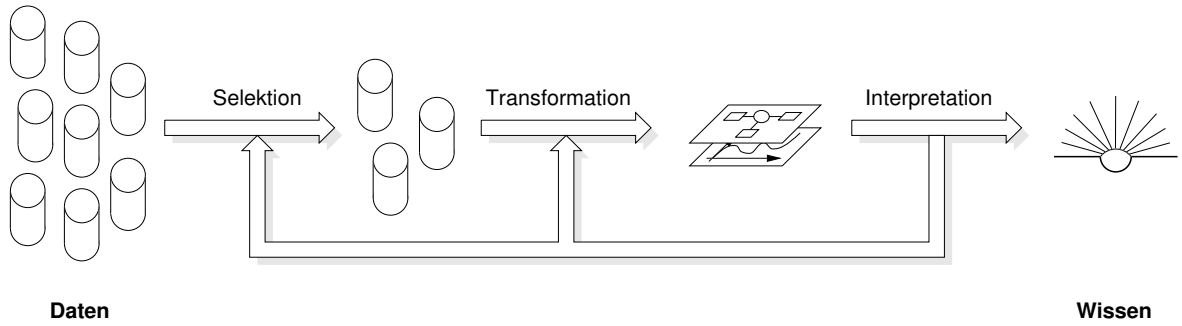


Abbildung 7.3: Herleitung von Wissen aus Daten.

arbeitung und Nutzung von Wissen. Ein wissensbasierter Dienstvermittler ist in diesem Sinne ein wissensbasiertes System. Die Typspezifikationen, die ein derartiger Dienstvermittler verarbeitet, basieren auf einem *wissensbasierten Typsystem*. Die wissensbasierte Typbeschreibungssprache, die Bestandteil des Typsystems ist, baut auf einer *Wissensrepräsentationstechnik* auf. Im folgenden werden die Syntax und Semantik der *Konzeptgraphen* (englisch: *Conceptual Graph*, siehe [98]) vorgestellt, einer Wissensrepräsentationstechnik, auf Basis derer ein wissensbasiertes Typsystem definiert wird.

In den folgenden Abschnitten wird auf die unterschiedlichen Aspekte der Konzeptgraphen eingegangen. In der Psychologie ist ein Konzeptgraph das Ergebnis eines Wahrnehmungsvorgangs. Aus dem Wahrnehmungsmodell, das in Abschnitt 7.2.1 vorgestellt wird, lassen sich die grundlegenden Eigenschaften eines Konzeptgraphen ableiten. In Abschnitt 7.2.2 wird eine formale Sprache für Konzeptgraphen beschrieben, die deren maschinelle Verarbeitung ermöglicht. Aus Anwendersicht werden in Abschnitt 7.2.3 einige Konstruktionshinweise von Konzeptgraphen aufgeführt. Für die weiteren Betrachtungen wird in Abschnitt 7.2.4 ein formales Modell eingeführt, das Aussagen über Konzeptgraphen in einer mathematischen Notation erlaubt. Schließlich wird in Abschnitt 7.2.5 der Zusammenhang zwischen Konzeptgraphen und der Prädikatenlogik erster Stufe hergestellt.

7.2.1 Wahrnehmungsmodell

Konzeptgraphen entspringen einem *Wahrnehmungsmodell* aus der Psychologie (siehe [38]). Die Wahrnehmung generiert ein abstraktes Abbild der Wirklichkeit, welches als Grundlage für höhere kognitive Prozesse dient. Das generierte Abbild als Abstraktion der Wirklichkeit wird nach [98] als *Konzeptgraph* bezeichnet. Ausgangspunkt einer Wahrnehmung sind Objekte oder Vorgänge der realen Welt, die in Form von *Sinneseindrücken* unterschiedlicher Sinnesorgane dem Gehirn mitgeteilt werden. *Perzeptuelle Merkmale* repräsentieren im Gehirn gelernte und gespeicherte Erfahrungen aus früheren Wahrnehmungen. Die Sinneseindrücke werden mit den im Gehirn gespeicherten perzeptuellen Merkmalen verglichen. Der assoziative Vergleich von Sinneseindrücken mit perzeptuellen Merkmalen erlaubt eine gewisse Unschärfe, da zwei Sinneseindrücke immer voneinander verschieden sind. Ein *Konzept* ist die Interpretation eines perzeptuellen Merkmals. Der Wahrnehmungsprozeß gliedert sich in folgende Schritte auf:

1. ein Objekt oder Ereignis wird durch den Vorgang der Wahrnehmung in eine Menge von Sinneseindrücken aufgeteilt.

2. jeder Sinneseindruck wird mit zuvor erlernten perzeptuellen Merkmalen verglichen.

3. für jedes übereinstimmende perzeptuelle Merkmal existiert ein Konzept, welches das perzeptuelle Merkmal interpretiert.

4. die so gewonnenen Konzepte werden über Relationen in Bezug zueinander gestellt, die die Beziehung der Sinneseindrücke untereinander reflektiert.

Ein Konzept ist *konkret*, wenn für dieses ein perzeptuelles Merkmal existiert. Umgekehrt ist ein Konzept *abstrakt*, wenn für dieses kein perzeptuelles Merkmal existiert. Beispielsweise kann das Konzept `[PEACE]` nicht über einen Sinneseindruck wahrgenommen werden. Alle Stimuli, die dem gleichen Konzept zugeordnet werden, sind *Instanzen* dieses Konzepts. Die *Denotation* eines Konzepts ist definiert als die Menge aller Instanzen des Konzepts. Manche Instanzen sind mehreren Konzepten zugeordnet (beispielsweise ist `HP-DESKJET` sowohl Instanz des Konzepts `[PRINTER]`, als auch des Konzepts `[HARDWARE]`).

Ein Konzeptgraph ist im mathematischen Sinne ein endlicher, zusammenhängender, gerichteter, bipartiter Graph, der aus Konzept– und Relationsknoten besteht. Ein Relationsknoten verbindet zwei oder mehrere Konzeptknoten. Die Orientierung der Kanten definiert eine Anordnung der Konzepte, die durch eine Relation in Beziehung stehen. Der Graph ist bipartit, da zwei Konzepte nie direkt verbunden sind, sondern nur über eine Relation. Wegen der endlichen Speicherkapazität des menschlichen Gehirns sind Konzeptgraphen, da sie einem Wahrnehmungsprozeß entspringen, ebenfalls endlich. Ein Konzeptgraph ist zusammenhängend. Ein nicht zusammenhängender Graph wird lediglich als zwei unabhängige Konzeptgraphen angesehen.

Das Ergebnis eines Wahrnehmungsvorgangs ist ein Konzeptgraph, der Fakten, Beobachtungen und Hypothesen kodiert. Es wurde nachgewiesen, daß Konzeptgraphen äquivalent zu anderen Wissensrepräsentationstechniken sind, wie beispielsweise KIF oder Frames (siehe [29] und [89]). Konzeptgraphen bieten den Vorteil, daß sie wegen ihrer Abstammung aus einem psychologischen Wahrnehmungsmodell für einen Anwender intuitiv sind. Das Konstruktionsprinzip von Konzeptgraphen ist an kognitive Vorgänge angelehnt, so daß Anwender ohne große Vorkenntnisse diese nachvollziehen können. Weiterhin lassen sich Konzeptgraphen visualisieren, was den interaktiven Umgang erleichtert. Konzeptgraphen werden von dem *ANSI Technical Committee X3T2* standardisiert (siehe ??), so daß mit einer weiten Verbreitung dieser Wissensrepräsentationstechnik zu rechnen ist.

7.2.2 Syntax der Konzeptgraphen

Die textuelle Repräsentation eines Konzeptgraphen heißt *lineare Notation*. Neben der maschinell verarbeitbaren Notation existiert eine *graphische Darstellung*. Jeder Konzeptgraph kann von der linearen Notation in eine äquivalente graphische Darstellung überführt werden und umgekehrt. Die lineare Notation basiert auf einer formalen Sprache, die die Konzeptgraphen einer maschinellen Verarbeitung zugänglich machen. Die formale Sprache

ist definiert über eine Grammatik, deren Wörter den syntaktisch korrekten Konzeptgraphen entsprechen. Grundlage für eine wissensbasierte Typbeschreibungssprache auf Basis der Konzeptgraphen ist die in Definition 7.1 angegebene Grammatik.

Definition 7.1 (Grammatik für Konzeptgraphen): Die Syntax eines Konzeptgraphen ist definiert über eine Grammatik $G_{CG} = (N, \Sigma, P, S)$, mit $N =$ { `cg`, `con_connects_to`, `rel_graph`, `con_subgraph`, `rel_subgraph`, `concept_node`, `instance_list`, `relation_node`, `id`, `value` }, Σ dem ISO8859–1 Zeichensatz mit $\Upsilon =_{df} \Sigma \setminus$ { `"0"`, ..., `"9"`, `"-"`, `"["`, `"]"`, `"("`, `")"`, `":"`, `","`, `"."`, `">"`, `"{"`, `"}"` }, $S =$ `cg` und einer Menge von Produktionen P:

```
cg               ← concept_node con_connects_to
con_connects_to  ←
con_connects_to  ← "->" rel_graph
con_connects_to  ← "-" rel_subgraph "."
rel_graph        ← relation_node "->" cg
rel_graph        ← relation_node "-" con_subgraph "."
con_subgraph     ← "->" cg
con_subgraph     ← "->" cg "," con_subgraph
rel_subgraph     ← "->" rel_graph
rel_subgraph     ← "->" rel_graph "," rel_subgraph
concept_node     ← "[" id "]"
concept_node     ← "[" id ":" value "]"
concept_node     ← "[" id ": {" instance_list "}]"
instance_list    ← id
instance_list    ← id "," instance_list
relation_node    ← "(" word ")"
id               ← Υ+
value            ← ["0", ..., "9"]+
```

□

In der linearen Notation sind Konzepte durch eckige Klammern gekennzeichnet, und Relationen durch runde Klammern. Ein Konzept besteht aus einem Konzeptnamen, sowie optional einem numerischen Wert oder einer Liste von Instanznamen. Ein Konzeptgraph besitzt Baumstruktur mit einem ausgezeichneten Konzeptknoten als Wurzel. Ein Relationsknoten verbindet zwei oder mehr Konzeptknoten und hat eine einlaufende Kante sowie eine oder mehrere auslaufende Kanten. Da ein Konzeptgraph bipartit ist, wechseln sich Konzept– und Relationsknoten in einem syntaktisch korrekten Graphen ab. Alle Blätter eines Konzeptgraphen sind Konzepte. Jeder Knoten (bis auf die Blätter) besitzt einen oder mehreren Nachfolger. Zwei Knoten sind in der linearen Notation durch das terminale Symbol `"->"` verbunden, das zugleich die Orientierung der Kante andeutet. Besitzt ein Knoten mehrere Nachfolger, so sind die einzelnen Äste durch die terminalen Symbole `"-"` und `"."` eingeschlossen und durch `","` voneinander getrennt.

Die hier vorgestellte Syntax der Konzeptgraphen ist eine Teilmenge des im Standards vorgesehenen Sprachumfangs. Trotz Einbußen bei der Ausdruckskraft eines Konzept-

graphen, ist die für diese Arbeit gewählte Teilmenge ausreichend. In Kapitel 8 werden Beispiele vorgeführt, die die Zweckmäßigkeit der hier eingeführten Syntax demonstrieren. In [65] sind die Unterschiede in der Syntax zu der im ANSI–Standard festgelegten Syntax beschrieben. Wenn im folgenden von Konzeptgraphen die Rede ist, wird für deren Struktur die in Definition 7.1 angegebene Grammatik angenommen. Der nachfolgende Konzeptgraph ist korrekt in dem Sinne, daß er ein Wort der formalen Sprache $L(G_{CG})$ ist:

```
[PRINTER:{HP-DESKJET}] -
        -> (VISUALIZES) -> [INFORMATION],
        -> (RESOLUTION) -> [DPI:150],
        -> (PRINTS-ON) -
                -> [PAPER] -
                       -> (DIMENSION) -> [SIZE:{A4}],
                       -> (COLOR) -> [WHITE]
                       .,
                -> [SLIDES]
                .
        .
```

Die informale Semantik des Konzeptgraphen lautet: *„Ein HP-Deskjet ist ein Drucker, der Informationen visualisiert, eine Auflösung von 150 DPI besitzt und Papier und Folien bedruckt. Das Papier besitzt die Dimensionen DIN–A4 und ist weiß.“* Der Begriff „und“ in der informalen Beschreibung stellt keinen logischen Operator dar, sondern ist als linguistisches Bindewort zu verstehen, um Elemente einer Aufzählung voneinander zu trennen. Allgemein kann ein Konzeptgraph der Form `[`K_1`] -> (`R`) -> [`K_2`]` mit entsprechenden Füllwörtern in einen natürlichsprachlichen Satz nach der Schablone *„Die R eines K_1 ist ein K_2“* übersetzt werden[3].

Ein Konzeptgraph ist eine Ansammlung von Aussagen, die über Konzepte charakterisiert werden. Die Art der Charakterisierung der Aussagen (d.h. welche Aspekte als wichtig oder unwichtig empfunden werden) sowie deren Wahrheitsgehalt hängen von den Präferenzen eines Betrachters ab. Ein Konzeptgraph liefert nicht die präzise allumfassende Beschreibung eines Konzepts, sondern lediglich eine Approximation. Die graphische Darstellung des oben angegebenen Konzeptgraphen ist in Abbildung 7.4 wiedergegeben. Rechtecke entsprechen den Konzepten und abgerundete Rechtecke den Relationen. Die Beziehung zwischen Relationen und Konzepten ist durch die gerichteten Kanten zwischen diesen gegeben.

7.2.3 Konstruktion von Konzeptgraphen

Die Sprache, die die Grammatik für Konzeptgraphen aus Definition 7.1 induziert, soll im folgenden als Typbeschreibungssprache für ein wissensbasiertes Typsystem herangezogen werden. Nicht alle Wörter (d.h. Typspezifikationen) der formalen Sprache $L(G_{CG})$

[3]Diese Schablone setzt voraus, daß die Relation R ein Substantiv ist. Handelt es sich bei R um ein Verb, muß die Schablone entsprechend grammatikalisch angepaßt werden. In einem Konzeptgraph ist nur die Semantik der Relation entscheidend, und nicht die Wortart zu der die Repräsentation von R auf einer extensionalen Ebene gehört.

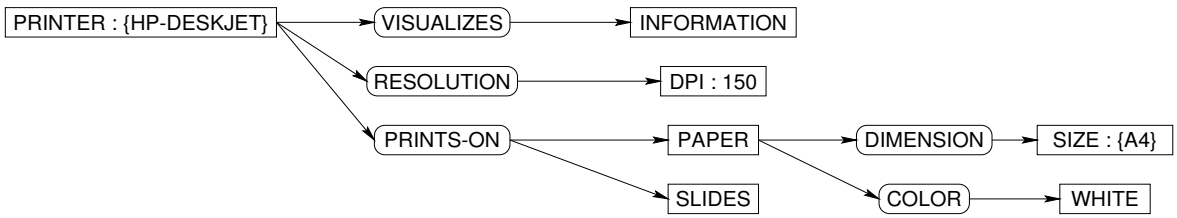


Abbildung 7.4: Graphische Repräsentation eines Konzeptgraphen.

ergeben jedoch für einen Anwender einen Sinn. Die Definition der Syntax von Konzeptgraphen alleine ist nicht ausreichend, um bedeutungsvolle Graphen zu erzeugen. Konzept– und Relationsknoten werden erst in einer bestimmten Anordnung von einem Anwender als sinnvoll erkannt. Die Menge der bedeutungsvollen Konzeptgraphen werden in [98] als *kanonische Konzeptgraphen* bezeichnet. Ein Konzeptgraph ist per Definition kanonisch, wenn er das Ergebnis eines Wahrnehmungsprozesses ist. Eine Grundmenge von kanonischen Graphen entsteht somit auf Basis der Wahrnehmung. Ausgehend von dieser Grundmenge können weitere kanonische Konzeptgraphen erzeugt werden. Nach [98] wird zwischen drei Mechanismen unterschieden, die kanonische Konzeptgraphen erzeugen:

- *Wahrnehmung*: Konzeptgraphen sind kanonisch, die aus einer Wahrnehmung hervorgehen.
- *Ableitungsregeln*: Neue kanonische Graphen können über Regeln von bereits existierenden kanonischen Konzeptgraphen abgeleitet werden (beispielsweise durch Generalisierung).
- *Introspektion*: Ein beliebiger Konzeptgraph kann aufgrund von Introspektion als kanonisch festgelegt werden (beispielsweise während eines kreativen Augenblicks).

Die elementaren Bausteine eines Konzeptgraphen — Konzepte und Relationen — entspringen den Erfahrungen eines Anwenders. Die Konzepte, als Kategorien konkreter und abstrakter Dinge, sind in lexikographischen Datenbanken, wie das bereits erwähnte WordNet, zusammengefaßt (siehe [70]). In [54] wird ein formales Analyseverfahren vorgeschlagen, um Konzepte zu synthetisieren. Die Relationen setzen jeweils zwei oder mehrere Konzepte in einen Kontext. Der Relationskatalog in [98] umfaßt 37 Relationen, von denen 7 in Tabelle 7.1 angegeben sind. Die Relation `Location` verbindet beispielsweise zwei Konzepte, wobei das eine den Ort ausdrückt an dem das andere Konzept plaziert ist. Eine Relation hat somit eine feste Stelligkeit, die in der folgenden formalen Darstellung berücksichtigt werden wird.

7.2.4 Formale Definition eines Konzeptgraphen

Die formale Spezifikation der Konzeptgraphen ist Voraussetzung für präzise Aussagen über deren Eigenschaften. Die folgenden Definitionen stellen die Begriffe *Konzept*, *Relation*, *Instanzen* und *Konzeptgraph* formal dar. Die formale Spezifikation basiert auf den

Relationsname	Erläuterung
`Agent`	Verknüpft einen Akteur mit einer Handlung.
`Attribute`	Verbindet zwei Objekte, wobei das eine Objekt ein Attribut eines anderen ist.
`Duration`	Gibt für eine Aktion eine Zeitdauer an.
`Initiator`	Verbindet einen Akteur mit einer von ihm ausgelösten Handlung.
`Location`	Verbindet ein Objekt mit einer Lokation.
`Quantity`	Verknüpft ein Objekt mit einer Quantität.
`Recipient`	Ein Akteur ist der Empfänger eines Objekts.

Tabelle 7.1: Auszug des Relationskatalogs nach Sowa.

in den letzten Abschnitten vorgestellten Eigenschaften der Konzeptgraphen. Die disjunkte Vereinigung wird im folgenden durch das Symbol $\dot{\cup}$ repräsentiert. Weiterhin ist die Potenzmenge $Pot(M)$ einer Menge M die Gesamtheit aller Teilmengen von M.

Definition 7.2 (Konzeptgraph): Die Mengen C, R, IN, I, P, sowie die Funktion f_S seien wie folgt definiert:

- C Menge von *Konzeptnamen*, R Menge von *Relationsnamen*, IN Menge von *Instanznamen*,
- $f_S : R \to \mathbb{N}$ *Stelligkeitsfunktion* für Relationen,
- $I =_{df} Pot(IN) \dot{\cup} \mathbb{N}$ Menge von *Instanzen*,
- $P =_{df} (C \times I) \dot{\cup} R$ der Menge von *Eigenschaftstupeln*.

Das Tripel $G = (N, L, F)$ heißt *Konzeptgraph* mit

- $N \subseteq \mathbb{N}$ endliche Menge von *Knotennummern*,
- $L \subseteq N \times N$ endliche Menge von *Kanten*,
- $F \subseteq N \times P$ endliche Menge von *Knoten*,

wenn das nachfolgende Konstruktionsprinzip anwendbar ist:

1. Sei (N, L, F) ein Tripel mit

   (i) $N = \{n\}$, $n \in \mathbb{N}$, die Nummer des *Wurzelknotens*,

   (ii) $L = \emptyset$,

   (iii) $F = \{(n, p)\}$, $n \in N$, $p \in C \times I$,

   dann ist $G = (N, L, F)$ Konzeptgraph.

2. Falls (N, L, F) Konzeptgraph und

(i) $N' = N \dot\cup \{n', n''\}$, $n', n'' \in \mathbb{N}$, $n' \neq n''$ und

(ii) $L' = L \dot\cup \{(n, n'), (n', n'')\}$, $n \in N$, $(n, p) \in F$ und

(iii) $F' = F \dot\cup \{(n', p'), (n'', p'')\}$ mit $p' \in R$ und $p'' \in C \times I$

dann ist $G = (N', L', F')$ auch Konzeptgraph.

3. Falls (N, L, F) Konzeptgraph und

(i) $N' = N \dot\cup \{n'\}$, $n' \in \mathbb{N}$ und

(ii) $L' = L \dot\cup \{(n, n')\}$, $n \in N$, $(n, p) \in F$, $p \in R$ und

(iii) $F' = F \dot\cup \{(n', p')\}$ mit $p' \in C \times I$

dann ist $G = (N', L', F')$ auch Konzeptgraph.

und folgende Bedingung gilt: $\forall (n, (r)) \in F : f_S(r) =\mid \{(n, n') \in L\} \mid$ □

Ein Konzeptgraph ist ein endlicher, gerichteter Baum. Die Menge P umfaßt alle vorstellbaren Konzept– und Relationsknoten. Das Konstruktionsprinzip garantiert, daß die Wurzel und die Blätter eines Konzeptgraphen immer Konzeptknoten sind. Weiterhin stellt das Konstruktionsprinzip sicher, daß ein Konzeptgraph bipartit ist. Durch die Partitionierung des bipartiten Graphen ergeben sich die zwei disjunkte Mengen $C \times I$ und R. Die Knoten eines Konzeptgraphen sind numeriert, wobei jedem Knoten eine eindeutige Nummer zugeordnet ist. Der Grund dafür besteht in der Tatsache, daß ein Konzept bzw. eine Relation an unterschiedlichen Stellen des Konzeptgraphen auftreten kann. Die Numerierung sorgt dafür, daß der Kontext aller Knoten durch ihre Nummer eindeutig bestimmt ist. Auf der Basis der Definition 7.2 werden folgende Schreibweisen eingeführt:

Schreibweise 7.1:

- Die Menge der nach obigem Bildungsgesetz konstruierbaren Konzeptgraphen sei $CG =_{df} \{G = (N, L, F) \mid G \text{ ist Konzeptgraph}\}$.
- Sei $(n, p) \in F$ und $p = (r)$ (Relationsknoten) oder $p = (c, i)$ (Konzeptknoten), dann ist
  - n die Nummer,
  - p die Spezifikation des Knotens,
  - c ein Konzeptname,
  - r ein Relationsname,
  - i eine Menge von Instanznamen.

Die Menge CG umfaßt alle syntaktisch korrekten Konzeptgraphen. Nicht alle von diesen sind notwendigerweise kanonische Konzeptgraphen. Nach außen hin bestimmen Konzept– und Relationsnamen die Semantik der einzelnen Knoten eines Konzeptgraphen.

Der Kontext, über den die Knoten in Beziehung zueinander stehen, ist bestimmt durch die Kanten, die auf die Knotennumerierung Bezug nehmen. Der rekursive Aufbau eines Konzeptgraphen resultiert in einer induktiven Struktur, die Grundlage für die Definition eines Teilgraphen ist.

Definition 7.3 (Teilgraph): Sei $G = (N, L, F)$ ein Konzeptgraph und $n \in N$, $(n, (c, i)) \in F$, dann ist $G' = (N', L', F')$ der durch n aufgespannte *Teilgraph* von G genau dann, wenn

1. $L' = \{(n^{(1)}, n^{(2)}) \in L \mid \exists (n_0, n_1), (n_1, n_2), \ldots, (n_{i-1}, n_i), (n_i, n_{i+1}) \in L \text{ mit } n_j \in N,$
   $j \in \{0, \ldots, i+1\},\ i \in \mathbb{N},\ n_0 = n,\ n_i = n^{(1)},\ n_{i+1} = n^{(2)}\},$
2. $N' = \{n', n'' \in N \mid \exists (n', n'') \in L'\},$
3. $F' = \{(n', p) \in F \mid n' \in N'\}$

□

Ein Teilgraph ist selbst wiederum ein Konzeptgraph. Ein Teilgraph eines Konzeptgraphen hat seinen Ursprung bei einem beliebigen Konzeptknoten und erstreckt sich über dessen transitive Hülle. Die Relation „G_1 *ist ein Teilgraph von* G_2“ ist reflexiv und transitiv, jedoch nicht symmetrisch. Die Wurzel eines Teilgraphen stimmt mit einem inneren Knoten des Konzeptgraphen überein, aus dem der Teilgraph entstammt. Der Teilgraph repräsentiert eine verfeinerte Spezifikation des inneren Knotens. Der Konzeptgraph aus Abschnitt 7.2.2 besitzt folgenden Teilgraph:

```
[PAPER] -
    -> (DIMENSION) -> [SIZE:{A4}],
    -> (COLOR) -> [WHITE]
    .
```

Das Konzept `[PAPER]`, auf das als innerer Knoten des Konzeptgraphen in Abschnitt 7.2.2 bezug genommen wird, wird durch den obigen Teilgraph in seiner Spezifikation verfeinert. Die Verfeinerung von Beschreibungen erlaubt eine Präzisierung von Konzepten, die alleine für sich nicht ausreichende Bedeutung für eine eindeutige Identifizierung bieten würden. Beispielsweise kann anhand des Konzepts `[BOARD]` nicht bestimmt werden, ob es sich um eine *Tafel*, oder um eine *Behörde* handelt. Erst eine weitere Verfeinerung der Beschreibung durch Teilgraphen hilft zwischen den unterschiedlichen Bedeutungen zu differenzieren:

```
[BOARD] -> (LOCATION) -> [WALL]
[BOARD] -> (LOCATION) -> [OFFICE]
```

Die folgende Definition führt einige gebräuchliche Operatoren auf Konzeptgraphen ein:

Definition 7.4 (Operationen auf Konzeptgraphen): Folgende Operationen können auf Konzeptgraphen $G = (N, L, F) \in CG$ durchgeführt werden (Mengensymbole wie in Definition 7.2):

1. $root \in \mathrm{ABB}(CG, \mathbb{N})$, $root(G) =_{df} n'$ mit $(n' \in N)(\forall\, n'' \in N)((n'', n') \notin L)$, eindeutig nach Konstruktion, bestimmt den Wurzelknoten eines Konzeptgraphen,

2. $succ \in \mathrm{ABB}(CG \times \mathbb{N}, Pot(\mathbb{N}))$, $succ(G, n) =_{df} \{n' \in N \mid (n, n') \in L\}$, Menge von direkten Nachfolgeknotennummern eines Knotens n,

3. $nth\_succ \in \mathrm{ABB}(CG \times \mathbb{N} \times \mathbb{N}, \mathbb{N})$,
$$nth\_succ(G, n, i) =_{df} \begin{cases} n' & : \quad n' \in succ(G, n) \wedge \\ & \quad\; |\{n'' \in succ(G, n) \mid n'' \leq n'\}| = i \\ undef. & : \quad \text{sonst} \end{cases}$$
liefert den i–ten Nachfolger des Knotens n. Die Numerierung der Nachfolger beginnt bei 1.

4. $pred \in \mathrm{ABB}(CG \times \mathbb{N}, \mathbb{N})$, $pred(G, n) =_{df} \begin{cases} n' & : \quad (\exists (n', n) \in L) \\ undef. & : \quad \text{sonst} \end{cases}$
liefert den Vaterknoten eines Knotens.

□

Die formale Schreibweise eines Konzeptgraphen korrespondiert unmittelbar mit seiner linearen Notation. Der Zusammenhang in den beiden Repräsentationsformen ist durch folgende Schreibweise gegeben:

Schreibweise 7.2: Sei $G = (N, L, F)$ ein Konzeptgraph. Aus der formalen Definition von G kann wie folgt die lineare Notation des Konzeptgraphen erzeugt werden:

- Für $(n, p) \in F$ und
  - `[`c`]` mit $p = (c, \emptyset)$,
  - `[`c`:{`i`}]` mit $p = (c, i)$, $i \in Pot(IN)$,
  - `[`c`:`x`]` mit $p = (c, x)$, $x \in \mathbb{N}$,
  - `(`r`)` mit $p = (r)$.
- Für $(n', n) \in L \wedge (n, (c, i)) \in F \wedge (n', (r')) \in F$:
  - $i \in Pot(IN)$: `[`c`:{`i`} ] -> (`r'`)`. Jedes Element aus i werden voneinander durch Kommata getrennt, oder
  - $i = \{x\}, x \in \mathbb{N}$: `[`$c$`:`$x$` ] -> (`$r'$`)` oder
  - $i = \emptyset$: `[`c`] -> (`r'`)`.
- Für $(n, n') \in L \wedge (n, (c, i)) \in F \wedge (n', (r')) \in F$:
  - $i \in Pot(IN)$: `(`r'`) -> [`c`:{`i`}]`. Bzgl. i analog oben. Oder
  - $i = \{x\}, x \in \mathbb{N}$: `(`$r'$` ) -> [`$c$`:`$x$`]` oder
  - $i = \emptyset$: `(`r'`) -> [`c`]`.

- Für $\mid succ(k) \mid > 1$ mit $k \in F$ beginnt eine Aufzählung mit einem „-“ und endet mit einem „.“. Jeder nachfolgende Teilgraph wird rekursiv analog aufgeschrieben und mit einem „,“ von dem nächsten Teilgraph getrennt.

Die Abbildung von der formalen Beschreibung eines Konzeptgraphen zu seiner linearen Darstellung ist eindeutig. Der umgekehrte Fall gilt jedoch nicht. Aus einer linearen Notation lassen sich unterschiedliche formale Beschreibungen erzeugen. Der Unterschied liegt in der Numerierung der Knoten. Alle erzeugbaren formalen Darstellungen stimmen jedoch modulo der Knotennumerierung überein. Zum Abschluß der Syntaxbeschreibung der Konzeptgraphen zeigt folgendes Beispiel den Zusammenhang zwischen der linearen Notation und der formalen Beschreibung eines Konzeptgraphen.

Beispiel 7.1:

a) Sei $G = (N, L, F)$ ein Konzeptgraph mit

- $N = \{3, 6, 8\}$,
- $L = \{(6, 3), (3, 8)\}$,
- $F = \{(6, (\texttt{OO-LANGUAGE}, \{\texttt{C++}\})), (3, (\texttt{SUPPORTS})),$
  $(8, (\texttt{CLASSES}, \emptyset))\}$

dann kann G wie folgt geschrieben werden:

```
[OO-LANGUAGE:{C++}] -> (SUPPORTS) -> [CLASSES]
```

b) Der Konzeptgraph aus Abschnitt 7.2.2 kann mit $G = (N, L, F)$ wie folgt formal dargestellt werden:

- $N = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$,
- $L = \{(1, 2), (2, 3), (1, 4), (4, 5), (1, 6), (6, 7), (7, 8), (8, 9), (7, 10), (10, 11), (6, 12)\}$,
- $F = \{(1, (\texttt{PRINTER}, \{\texttt{HP-DESKJET}\})), (2, (\texttt{VISUALIZES})),$
  $(3, (\texttt{INFORMATION}, \emptyset)), (4, (\texttt{RESOLUTION})), (5, (\texttt{DPI}, \texttt{150})),$
  $(6, (\texttt{PRINTS-ON})), (7, (\texttt{PAPER}, \emptyset)), (8, (\texttt{DIMENSION})),$
  $(9, (\texttt{SIZE}, \{\texttt{ A4 }\})), (10, (\texttt{COLOR})), (11, (\texttt{WHITE})),$
  $(12, (\texttt{SLIDES}, \emptyset))\}$

7.2.5 Φ–Operator

Die Syntax der Konzeptgraphen legt eine *äußere Struktur* fest, die Voraussetzung für eine maschinelle Verarbeitung ist. Neben der äußeren Struktur besitzen die Konzeptgraphen eine *innere Struktur*, die durch ihre Semantik definiert ist. Ohne eine entsprechende Theorie über die Semantik ist ein Konzeptgraph ein bedeutungsleeres Gebilde, das nicht

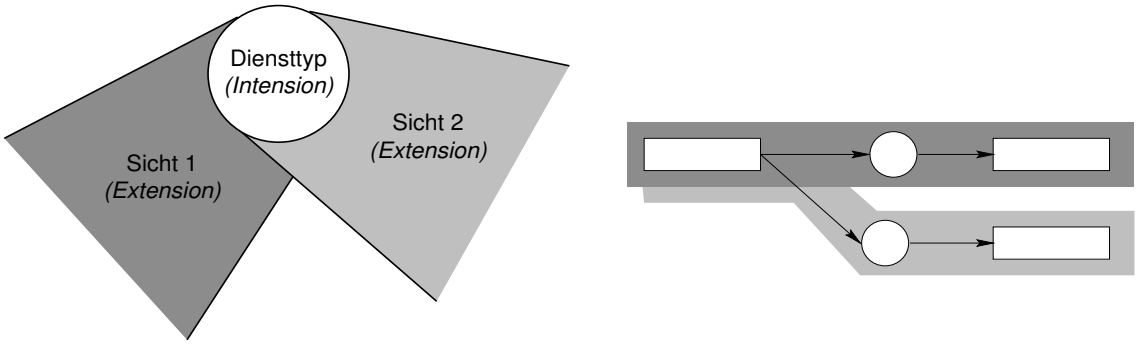


Abbildung 7.5: Konzeptgraph als Approximation der Intension.

als Gegenstand von inhaltsbezogenen Diskussionen dienen kann. Die zentrale Definition dieses Abschnitts ist eine Abbildung von Konzeptgraphen auf die Prädikatenlogik erster Stufe. Diese hier eingeführte Abbildung unterscheidet sich von der Sowa's (siehe [98]).

Der Unterschied ergibt sich aus der unterschiedlichen Betrachtungsweise über das, was ein Konzeptgraph repräsentiert. Die Theorie der Konzeptgraphen als Wissensrepräsentationstechnik dient im Rahmen dieser Arbeit als Typbeschreibungssprache für Diensttypen. Bei den Anforderungen an Typsysteme für offene verteilte Systeme wurde bereits die Notwendigkeit von semantischen Typbeschreibungssprachen erläutert, die Voraussetzung für die maschinelle Zuordnung von zwei Extensionen mit gleicher Intension ist. Im Unterschied zu der im letzten Kapitel vorgestellten deklarativen Typbeschreibungssprache ist ein Konzeptgraph als extensionale Spezifikation lediglich eine Approximation der Intension eines Diensttyps. Die unterschiedlichen Extensionen korrespondieren mit verschiedenen Sichten auf eine Intension (siehe Abbildung 7.5). Für das Konzept `[PRINTER]` sind beispielsweise zwei Sichten vorstellbar; eine funktionale (was macht ein Drucker) und eine strukturelle (woraus besteht ein Drucker). Beide Sichten entspringen legitimen Beschreibungen des Konzepts.

Ein Konzeptgraph vereinigt die einzelnen Sichten in Form unterschiedlicher Äste des Graphen. Je ein Ast korrespondiert mit einer Sicht auf den Diensttyp. Ein Konzeptgraph approximiert immer nur die Intension, repräsentiert diese aber im Unterschied zu einer deklarativen Typspezifikation nicht vollständig. An dieser Stelle lassen sich bereits Konsequenzen für die Dienstvermittlung auf Basis einer wissensbasierten Typbeschreibungssprache ableiten. Während die Typkonformität bei dem deklarativen Typsystem deterministisch ist, kann dies bei einem wissensbasierten Typsystem nicht mehr garantiert werden. Mit deterministisch ist gemeint, daß die Typkonformität das Verhältnis zwischen Extension und Intension korrekt widerspiegelt.

Die einzelnen Extensionen, die ein Konzeptgraph in sich vereinigt, werden im folgenden als *Alternativen* des Konzeptgraphen bezeichnet. Jede Alternative korrespondiert mit einer unterschiedlichen Sichtweise auf die Intension eines Diensttyps, den der Konzeptgraph spezifiziert.

Definition 7.5 (Alternative eines Konzeptgraphs): Sei $G = (N, L, F)$ ein Konzeptgraph. Dann ist der Konzeptgraph $G' = (N', L', F')$ eine *Alternative* von G genau dann,

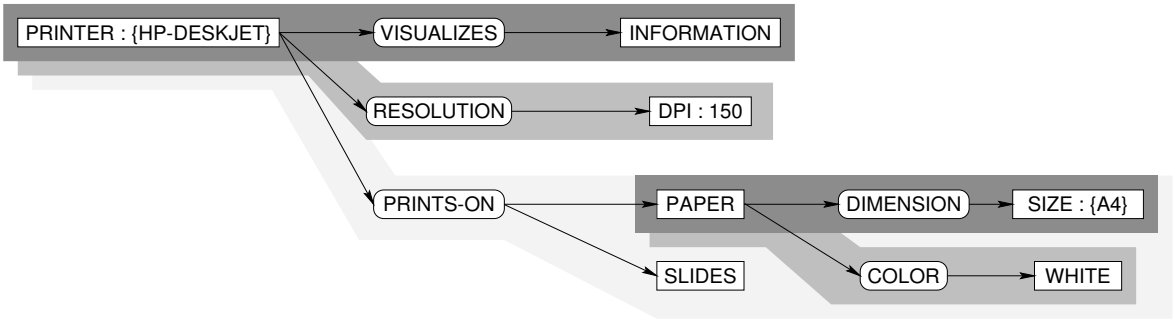


Abbildung 7.6: Teilgraphen als alternative Sichten.

wenn

1. $N' \subseteq N$, $root(G) \in N'$
2. $\forall\, n' \in N' : (pred(G, n') \in N')$
3. $\forall\, n' \in N' : ((node(G, n') = (n', (r'))) \Rightarrow (succ(G, n') \subseteq N'))$
4. $\forall\, n' \in N' : ((node(G, n') = (n', (c, i))\wedge \mid succ(G, n') \mid \neq 0) \Rightarrow$
   $\exists_1\, n'' \in N' : (n'' \in succ(G, n')))$
5. $\forall\, n', n'' \in N' : ((node(G, n') = (n', (r')) \wedge pred(G, n') = pred(G, n'')) \Rightarrow n' = n'')$
6. $L' =_{df} \{(n, n') \in L \mid n, n' \in N'\}$
7. $F' =_{df} \{(n, p) \in F \mid n \in N'\}$

□

Eine Alternative eines Konzeptgraphs ist selbst wieder ein Konzeptgraph. Die Kanten– und Knotenmenge einer Alternative ist eine Teilmenge des Konzeptgraphs, aus der sie hervorgeht. Zu einer Alternative gehört immer das Wurzelkonzept des Konzeptgraphen sowie ein vollständiger, zusammenhängender Pfad bis zu den Blättern. Von jedem Konzeptknoten gehört immer nur genau ein Teilgraph zu einer Alternative, während die Relationsknoten entlang des Pfads mit allen ihren nachfolgenden Teilgraphen zu der Alternative gehören. Der in Abbildung 7.5 dargestellte Konzeptgraph besitzt beispielsweise zwei Alternativen. Die vier Alternativen des Konzeptgraphen aus Abschnitt 7.2.2 sind in Abbildung 7.6 schattiert unterlegt.

Der nächste Schritt für eine Abbildungsvorschrift von Konzeptgraphen zu prädikatenlogischen Formeln erster Stufe besteht in der Definition der *korrespondierenden Prädikate*. Jedem Knoten eines Konzeptgraphen wird über folgende Definition ein Prädikat zugeordnet.

Definition 7.6 (Korrespondierende Prädikate): Sei $G = (N, L, F)$ ein Konzeptgraph. Zunächst seien Markierungen $m^{(j)}$ für $j \in N$ wie folgt definiert:

$$m^{(j)} =_{df} \begin{cases} x_j & : \quad (j, (c, \emptyset)) \in F \\ i & : \quad (j, (c, i)) \in F \\ undef. & : \quad \text{sonst} \end{cases}$$

Die x_j bezeichnen Variablen. Das *korrespondierende Prädikat* eines Knotens $n \in N$ von G ist über die Abbildung Ω definiert als:

$$\Omega(n) =_{df} \begin{cases} c(m^{(n)}) & : n \text{ ist Konzept} \\ r(m^{(pred(G,n))}, m^{(nth\_succ(G,n,1))}, \ldots, m^{(nth\_succ(G,n,|succ(G,n)|))}) & : n \text{ ist Relation} \end{cases}$$

□

Die Namen der Prädikate stimmen mit den Konzept– bzw. Relationsnamen der Knoten des Konzeptgraphen überein. Jeder Konzeptknoten wird auf ein einstelliges Prädikat abgebildet. Der Parameter ergibt sich aus den Instanzen, die dem Konzept zugeordnet sind. Besitzt ein Konzept keine Instanzen, so wird anstatt dessen eine eindeutige Variable eingesetzt. Die Anzahl der Parameter der korrespondierenden Prädikate für Relationsknoten ist abhängig von der Anzahl der Nachfolger, die die Relation besitzt. Der erste Parameter stimmt mit dem des korrespondierenden Prädikats des Vorgängerknotens überein und alle übrigen mit je einem Nachfolgeknoten der Relation. Die Kanten eines Konzeptgraphen, die einen Kontext zwischen den einzelnen Knoten herstellen, korrespondieren mit den Instanznamen bzw. freien Variablennamen. Das korrespondierende Prädikat eines Relationsknotens stellt mit seinen Argumenten seine benachbarten Konzepte in Bezug zueinander.

Die Definitionen der Alternativen eines Konzeptgraphen sowie seiner korrespondierenden Prädikate bilden die Basis des Φ–Operators, der einen Konzeptgraphen auf eine prädikatenlogische Formel erster Stufe abbildet.

Definition 7.7 (Φ–Operator): Sei $G = (N, L, F)$ ein Konzeptgraph. Seien $G_1, \ldots, G_n$ alle Alternativen von G mit $G_i = (N_i, L_i, F_i)$. Jeder Alternative G_i wird eine Teilformel zugeordnet mit $p_i =_{df} \bigwedge_{\forall j \in N_i} \Omega(j)$. Sei die Formel p die disjunktive Verknüpfung der Teilformeln: $p =_{df} \bigvee_{j \in \{1,\ldots,n\}} p_j$. Seien $x_1, \ldots, x_k$ alle in p frei vorkommenden Variablen. Der Φ–Operator ist definiert als: $\Phi(G) =_{df} \exists\, x_1 \exists\, x_2 \cdots \exists\, x_k(p)$. □

Innerhalb einer Alternative sind die korrespondierenden Prädikate konjunktiv verknüpft. Die sich daraus ergebenden Teilformeln sind miteinander disjunktiv verknüpft. Die auf Basis des Φ–Operators erzeugte prädikatenlogische Formel bringt hierdurch zum Ausdruck, daß die einzelnen Alternativen eines Konzeptgraphen unterschiedliche Beschreibungen des gleichen Konzepts liefern. Entlang eines Astes, der eine Alternative charakterisiert, erzeugen die Relationen einen Kontext, die zwei oder mehrere Konzepte in eine Beziehung zueinander stellen. Die Zuordnung der Konzepte durch Relationen innerhalb einer Alternative resultiert hierdurch in einer konjunktiven Verknüpfung der korrespondierenden Prädikate.

Die Disjunktion der Konjunktion von korrespondierenden Prädikaten ist existentiell abgeschlossen. Alle in der prädikatenlogischen Formel vorkommenden Variablen werden durch einen Existenzquantor gebunden. Der existentielle Abschluß bringt zum Ausdruck, daß ein Konzeptgraph eine Aussage repräsentiert, die für einen bestimmten Diensttyp zutrifft. Unterschiede in den Eigenschaften eines Diensttyps dürfen sich nicht notwendigerweise auf das Wurzelkonzept eines anderen Konzeptgraphen auswirken, wie es bei einem universellen Abschluß der Fall wäre. Beispielsweise werden mit den folgenden zwei

Konzeptgraphen zwei Diensttypen mit unterschiedlichen Eigenschaften spezifiziert:

```
[PRINTER] -> (RESOLUTION) -> [DPI:150]
[PRINTER] -> (RESOLUTION) -> [DPI:300]
```

Bei einem universellen Abschluß würde sich die informelle Semantik des ersten Konzept ändern in *„alle Drucker besitzen eine Auflösung von 150 DPI“*, so daß beide Graphen nicht gleichzeitig wahr sein könnten. Obwohl der Φ–Operator alle Konzeptgraphen auf eine prädikatenlogische Formel erster Stufe abbildet, gibt es keine allgemeine Umkehrabbildung. Es existieren prädikatenlogische Formeln, die nicht durch einen Konzeptgraphen darstellbar sind. Beispielsweise trifft dies auf jede Formel zu, die eine Negation enthält. Der Φ–Operator unterscheidet sich in der Art, wie Sowa seinen gleichnamigen Operator definiert hat. Sowa trifft keine Unterscheidung zwischen verschiedenen Alternativen eines Konzeptgraphs, sondern verknüpft alle korrespondierenden Prädikate konjunktiv. Die Konsequenz ist, daß ein Konzeptgraph nach Sowa immer genau eine Sicht eines Diensttyps spezifiziert.

Beispiel 7.2: Die Konzeptgraphen aus Beispiel 7.1 können mit Φ überführt werden in

a) $\exists\, x(\text{OO-LANGUAGE}(\text{C++}) \wedge \text{SUPPORTS}(\text{C++}, x) \wedge \text{CLASSES}(x))$

b) $\exists\, w \exists\, x \exists\, y \exists\, z((\text{PRINTER}(\text{HP-DESKJET}) \wedge \text{VISUALIZES}(\text{HP-DESKJET}, w) \wedge \text{INFORMATION}(w)) \vee (\text{PRINTER}(\text{HP-DESKJET}) \wedge \text{RESOLUTION}(\text{HP-DESKJET},150) \wedge \text{DPI}(150)) \vee (\text{PRINTER}(\text{HP-DESKJET}) \wedge \text{PRINTS-ON}(\text{HP-DESKJET}, x, y) \wedge \text{SLIDES}(y) \wedge \text{PAPER}(x) \wedge \text{DIMENSION}(x, \text{A4}) \wedge \text{SIZE}(\text{A4})) \vee (\text{PRINTER}(\text{HP-DESKJET}) \wedge \text{PRINTS-ON}(\text{HP-DESKJET}, x, y) \wedge \text{SLIDES}(y) \wedge \text{PAPER}(x) \wedge \text{COLOR}(x, z) \wedge \text{WHITE}(z)))$

7.3 Ontologie für Konzeptgraphen

Der im letzten Abschnitt eingeführte Φ–Operator bildet einen Konzeptgraphen auf eine prädikatenlogische Formel erster Stufe ab, wodurch die Semantik der inneren Struktur definiert ist. Ein Konzeptgraph als Extension besteht aus einer Menge von Alternativen, die die Intension eines Diensttyps approximieren. Die logische disjunktive Verknüpfung der aus den Alternativen zugeordneten Teilformeln spiegelt die unterschiedlichen Sichtweisen (d.h. Extensionen) auf einen Diensttyp wider, die abgesehen von ihrer Qualität gleichberechtigt sind.

Ein Konzeptgraph steht jedoch nicht für sich alleine, sondern stellt eine Typspezifikation dar, die im Kontext zu anderen Typspezifikationen existiert. Neben der Semantik, die sich aus der inneren Struktur eines Konzeptgraphen ableitet, stehen dessen Komponenten — die Konzept– und Relationsknoten — in einer Beziehung höherer Ordnung zueinander. Diese Beziehung, genannt *Ontologie* oder *semantisches Netzwerk*, ist ein Abbild der Realität und bettet einen Konzeptgraph in einen Kontext mit anderen Konzeptgraphen

ein. Während ein Konzeptgraph als Spezifikation eines Diensttyps einer Aussage gleicht, beinhaltet eine Ontologie allgemeines Umweltwissen. Die Semantik höherer Ordnung, die einen Konzeptgraph in Bezug zu anderen Konzeptgraphen stellt, ist Voraussetzung für die Definition der wissensbasierten Typkonformität. In Abschnitt 7.3.1 werden zunächst die Beziehungen zwischen Konzept– und Relationsnamen untereinander diskutiert und anschließend in Abschnitt 7.3.2 die Eigenschaften von Konzeptgraphen in bezug auf eine Ontologie.

7.3.1 Konzept– und Relationshierarchie

Die *Konzepthierarchie*, die einen Teil der Ontologie repräsentiert, setzt Konzepte in eine bestimmte Beziehung zueinander. Die Art der Beziehung basiert auf einer Spezialisierung, die ein wissensbasiertes Pendant zum Austauschbarkeitsprinzip des Inklusions–Polymorphismus repräsentiert. Die folgende Definition führt den Begriff der Konzepthierarchie formal ein.

Definition 7.8 (Konzepthierarchie): Das Tripel $T_C = (C, \leq_C, \texttt{SOMETHING})$ heißt *Konzepthierarchie* mit

1. C die Menge von Konzeptnamen.
2. $\leq_C \subseteq C \times C$ einer Halbordnung auf der Menge der Konzeptnamen C.
3. `SOMETHING` dem Wurzelelement mit $\texttt{SOMETHING} \in C$ und $\forall\, c \in C : (c \leq_C \texttt{SOMETHING})$.

□

Definition 7.9 (Operationen auf einer Konzepthierarchie): Sei $T_C = (C, \leq_C, \texttt{SOMETHING})$ ein Konzepthierarchie. Dann sind folgende Operationen definiert:

1. $Gen_C \in \text{ABB}(C, Pot(C))$, $Gen_C(c) =_{df} \{c' \mid c \leq_C c'\}$ ist die *Generalisierung*.
2. $Spec_C \in \text{ABB}(C, Pot(C))$, $Spec_C(c) =_{df} \{c' \mid c' \leq_C c\}$ ist die *Spezialisierung*.

□

Die Halbordnung $\leq_C$, die zwei Konzepte in Beziehung zueinander stellt, repräsentiert eine Spezialisierung. Gilt für zwei Konzepte $c_1 \leq_C c_2$, dann ist das Konzept c_1 spezieller als c_2. Die Spezialisierung ist aus dem Verhältnis von Konzepten zu den ihnen zugeordneten Instanzen abgeleitet. Zu der *Denotation* eines Konzepts gehören alle Instanzen, die dem Konzept zugehörig sind. Für die Beziehung $c_1 \leq_C c_2$ gilt, daß die Menge der Instanzen, die dem Konzept c_1 zugeordnet sind, eine Obermenge der dem Konzept c_2 zugeordneten Instanzen ist. Beispielsweise gilt für alle Instanzen des Konzepts `[PRINTER]`, daß sie auch Instanzen des Konzepts `[HARDWARE]` sind. Diese Eigenschaft erfüllt hierdurch das Austauschbarkeitsprinzips des Inklusions–Polymorphismus und ist als Relation über Konzepte reflexiv, anti–symmetrisch und transitiv. Das Konzept `[SOMETHING]` bildet die Wurzel der Konzepthierarchie, d.h. alle Individuen sind Instanzen dieses Konzepts.

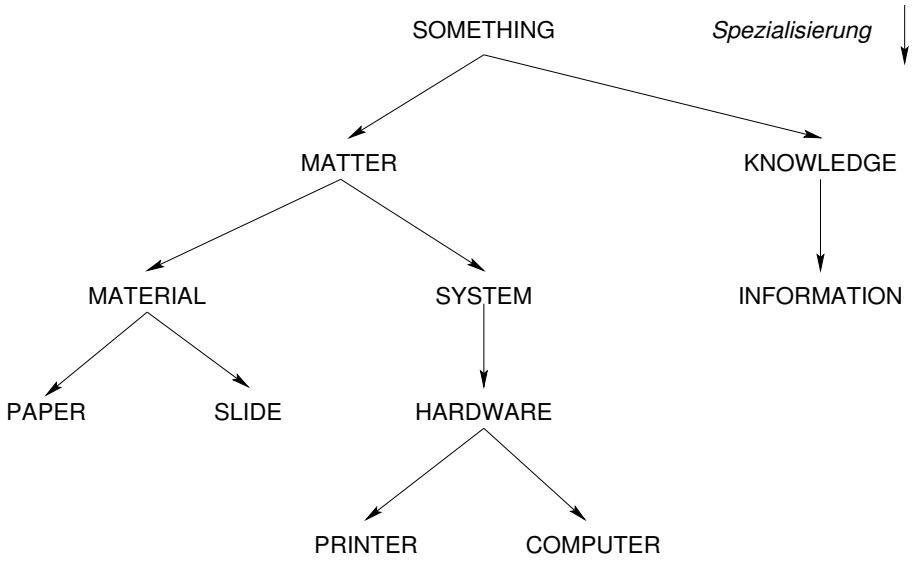


Abbildung 7.7: Konzepthierarchie.

Die Konzepthierarchie enthält allgemeingültige Aussagen über Beziehungen zwischen Konzepten (Abbildung 7.7 enthält einen Ausschnitt aus einer hypothetischen Konzepthierarchie). Die Konzepthierarchie als Teil einer Ontologie stellt hierdurch die Konzepte zueinander in einen Bezug. Die Bedeutung eines Konzepts ergibt sich aus seiner relativen Position innerhalb der Hierarchie. Für die über Konzepte definierte Halbordnung $\leq_C$ lassen sich Konformitätsbedingungen angeben, die eine Konzepthierarchie erfüllen muß. Der Bezug zur Realität wird durch folgende Konformitätsrelation hergestellt[4].

Definition 7.10 (Konformitätsrelation): Sei $T_C = (C, \leq_C, \texttt{SOMETHING})$ eine Konzepthierarchie und I eine Menge von Instanzen. Dann ist $Conf \in \mathrm{ABB}(C \times I, \{true, false\})$ die *Konformitätsrelation* der Konzepthierarchie T_C mit

1. $\forall\, i \in I : Conf(\texttt{SOMETHING}, i) = true$
2. $Conf(c, i) = true \wedge i' \subseteq i \Rightarrow Conf(c, i') = true$
3. $Conf(c, i) = true \Rightarrow (\forall\, c' \in Gen_C(c) \Rightarrow Conf(c', i) = true)$

□

Die Konformitätsrelation bewertet, ob eine Menge von Instanzen zur Denotation eines Konzepts gehört. Unabhängig von der Struktur einer Konzepthierarchie gelten für die Konformitätsrelation allgemeine Konsistenzbedingungen. Nach Eigenschaft (1) aus Definition 7.10 gehören zur Denotation des Wurzelkonzepts einer Konzepthierarchie alle Instanzen. Eigenschaft (2) besagt, daß jede Teilmenge der Denotation eines Konzepts auch

[4]In der Linguistik entspricht die Halbordnung $\leq_C$ gerade der in Abschnitt 7.1 eingeführten Hyponym–Beziehung. Die Bedingungen der Konformitätsrelation *Conf* spiegelt die linguistische Definition der Hyponym–Beziehung wider.

die Konformitätsrelation erfüllen muß. Daraus leitet sich der Spezialfall $Conf(c, \emptyset) = true$ ab. Mit der Konzepthierarchie aus Abbildung 7.7 gilt beispielsweise $Conf($`PRINTER`, {`HP-DESKJET`, `EPSON-FX80` }$) = true \Rightarrow Conf($`PRINTER`, { `HP-DESKJET`}$) = true$. Die Eigenschaft (3) besagt, daß Instanzen eines Konzepts auch zur Denotation von allgemeineren Konzepten gehören. Es gilt beispielsweise $Conf($`PRINTER`, {`HP-DESKJET`}$) = true \Rightarrow Conf($`HARDWARE`, {`HP-DESKJET`}$) = true$.

Die Menge der Relationen kann analog zu einer Relationshierarchie angeordnet werden. Da Relationen im Unterschied zu Konzepten keine Instanzen besitzen, kann für sie die Spezialisierung nicht über eine Konformitätsrelation formalisiert werden. Daß intuitiv Relationen dennoch zu einer Hierarchie angeordnet werden können, zeigt das Beispiel der Relation `(LOCATION)`, deren Spezialisierungen die Relationen `(UNDER)`, `(ABOVE)` oder `(IN)` sind. Die Rechtfertigung für die nachfolgende Definition einer Relationshierarchie basiert hierdurch ausschließlich auf der Hyponym–Beziehung aus der Linguistik.

Definition 7.11 (Relationshierarchie): Das Tupel $T_R = (R, \leq_R)$ heißt *Relationshierarchie* mit

1. R die Menge von Relationsnamen.
2. $\leq_R \subseteq R \times R$ eine Halbordnung definiert auf der Menge der Relationsnamen R.

□

Die Relation $\leq_R$ induziert eine Halbordnung auf der Menge der Relationen. Gilt $r \leq_R r'$, dann ist die Relation r eine Spezialisierung der Relation r'. Die Relationshierarchie besitzt im Unterschied zu der Konzepthierarchie nicht ein ausgezeichnetes Wurzelelement. Der Grund dafür besteht darin, daß zwei Relationen unterschiedliche Stelligkeiten besitzen können. Es sind nur solche Relationen bzgl. der Halbordnung $\leq_R$ vergleichbar, die die gleiche Stelligkeit haben. Eine Relationshierarchie kann deshalb nicht ein eindeutig bestimmtes Wurzelelement besitzen, sondern vielmehr eine Menge von Wurzelelementen. Jedes der Wurzelelemente ist Einstiegspunkt für einen Teil der Relationshierarchie. Für eine Relationshierarchie sind folgende Operationen definiert:

Definition 7.12 (Operationen auf einer Relationshierarchie): Sei $T_R = (R, \leq_R)$ eine Relationshierarchie. Dann sind folgende Operationen definiert:

1. $Gen_R \in \text{ABB}(R, Pot(R))$, $Gen_R(r) =_{df} \{r' \mid r \leq_R r'\}$ ist die *Generalisierung*.
2. $Spec_R \in \text{ABB}(R, Pot(R))$, $Spec_R(r) =_{df} \{r' \mid r' \leq_R r\}$ ist die *Spezialisierung*.

□

7.3.2 Eigenschaften von Konzeptgraphen unter einer Ontologie

Konzept– und Relationshierarchien setzen Konzepte und Relationen in Bezug zueinander. Durch sie steht ein Konzeptgraph in einem Kontext zu anderen Konzeptgraphen. Analog zu der Definition des Φ–Operators, der die Semantik eines Konzeptgraphen auf Basis der

Prädikatenlogik erster Stufe darstellt, soll die formale Definition der Konzept– und Relationshierarchie ebenfalls auf die Prädikatenlogik übersetzt werden. Eine Darstellung der Hierarchien durch prädikatenlogische Formeln erster Stufe soll im folgenden als Ontologie bezeichnet werden. Eine Ontologie erlaubt die formale Spezifikation von Eigenschaften eines Konzeptgraphen in Bezug zu anderen Konzeptgraphen, wie sie beispielsweise bei der im folgenden Abschnitt eingeführten semantischen Typkonformität benötigt wird.

Definition 7.13 (Ontologie): Sei T_C eine Konzepthierarchie und T_R eine Relationshierarchie. Eine *Ontologie* Γ auf Basis von T_C und T_R ist definiert als die disjunkte Vereinigung von $\Gamma =_{df} F_C \dot{\cup} F_R$, mit:

- $F_C =_{df} \{\forall\, x \in I : c(x) \Rightarrow c'(x) \mid c \leq_C c'\}$
- $F_R =_{df} \{\forall\, \tilde{x} \in I^n : r(\tilde{x}) \Rightarrow r'(\tilde{x}) \mid r \leq_R r' \wedge f_S(r) = n\}$

□

Eine Ontologie ist definiert über eine Konzepthierarchie T_C und eine Relationshierarchie T_R und enthält universell–abgeschlossene prädikatenlogische Formeln der Form $\forall\, x : (a(x) \Rightarrow b(x))$. Die Namen der Prädikate ergeben sich aus den Konzept– und Relationsnamen. Die Parameter der Prädikate entstammen der Menge von Instanzen I. Bei der Menge F_C aus Definition 7.13 sind die Parameter der Prädikate Elemente der Menge I, der Menge der Instanzen. Die Parameter der Prädikate der Menge F_R dagegen sind Tupel über ein kartesisches Produkt über der Menge der Instanzen. Die Dimension der Tupel hängt von der Stelligkeit der Relationen der Menge R ab, und ist festgelegt von der Stelligkeitsfunktion f_S.

Mit der Ontologie können weitere Eigenschaften eines Konzeptgraphen formuliert werden. Eine *subsumierte Alternative* eines Konzeptgraphen ist eine redundante Alternative, die von einer anderen Alternative logisch impliziert wird.

Definition 7.14 (Subsumierte Alternative): Sei G ein Konzeptgraph und G' eine Alternative von G. Der Konzeptgraph G' heißt eine *subsumierte Alternative* bzgl. einer Ontologie Γ genau dann, wenn eine Alternative G'' von G existiert, mit $G' \neq G''$ und $\Gamma \cup \Phi(G'') \models \Phi(G')$. □

Eine Alternative eines Konzeptgraphen ist per Definition wieder ein Konzeptgraph. Folglich kann der Φ–Operator auf eine Alternative angewandt werden. Der Begriff der subsumierten Alternative ist über den semantischen Folgerungsbegriff als eine Alternative G' definiert, die aus der Vereinigung der Ontologie Γ mit der sie subsumierenden Alternative G'' logisch impliziert wird. Jede Interpretation, die $\Gamma \cup \Phi(G'')$ zu *true* evaluiert, ist auch ein Modell für $\Phi(G')$. Eine subsumierte Alternative ist somit abhängig von einer Ontologie. Für den Spezialfall $\Gamma = \emptyset$ besitzt ein Konzeptgraph nur dann eine subsumierte Alternative, wenn dieser zwei syntaktisch äquivalente Alternativen besitzt. Der Begriff der subsumierten Alternative ist Grundlage für die Definition eines *minimalen Konzeptgraphen*.

Definition 7.15 (Minimaler Konzeptgraph): Ein *minimaler Konzeptgraph* bzgl. einer Ontologie ist definiert als ein Konzeptgraph, bei dem jede Alternative von keiner

anderen Alternative bzgl. der Ontologie subsumiert wird. □

Die Definition eines minimalen Konzeptgraphen nimmt ebenfalls Bezug auf eine Ontologie. Intuitiv ist ein Konzeptgraph minimal, wenn er keine redundanten Informationen enthält. Eine Information ist redundant, wenn eine Alternative eines Konzeptgraphen bzgl. einer anderen subsumiert wird. Die Konsequenz aus der obigen Definition ist, daß bei einem nicht minimalen Konzeptgraph eine oder mehrere seiner Alternativen entfernt werden können, ohne das sich die Semantik des Konzeptgraphen ändert. Das folgende Beispiel verdeutlicht die vorherigen Definitionen.

Beispiel 7.3: Gegeben sei der Konzeptgraph G:

```
[PRINTER:{HP-DESKJET}] -
            -> (VISUALIZES) -> [INFORMATION],
            -> (VISUALIZES) -> [KNOWLEDGE]
            .
```

Im folgenden wird gezeigt, daß eine der beiden Alternativen des Konzeptgraphen G bzgl. der Konzepthierarchie aus Abbildung 7.7 redundant ist. Die Ontologie Γ sei definiert gemäß der Konzepthierarchie in Abbildung 7.7, mit $F_R = \emptyset$. Demnach ist $\{\forall\, x \in I : (\text{INFORMATION}(x) \Rightarrow \text{KNOWLEDGE}(x))\} \in \Gamma$, da INFORMATION $\leq_C$ KNOWLEDGE.

Der Konzeptgraph G' sei gegeben als

```
[PRINTER:HP-DESKJET] -> (VISUALIZES) -> [KNOWLEDGE]
```

Es ist $\Phi(G') = \exists\, x(\text{PRINTER(HP–DESKJET)} \wedge \text{VISUALIZES(HP–DESKJET, } x) \wedge \text{KNOWLEDGE}(x))$

Der Konzeptgraph G'' sei gegeben als

```
[PRINTER:HP-DESKJET] -> (VISUALIZES) -> [INFORMATION]
```

Es ist $\Phi(G'') = \exists\, x(\text{PRINTER(HP–DESKJET)} \wedge \text{VISUALIZES(HP–DESKJET, } x) \wedge \text{INFORMATION}(x))$

Die Konzeptgraphen G' und G'' sind die beiden einzigen Alternativen, die G besitzt. Bzgl. der Ontologie Γ gilt, daß jede Interpretation, die Γ und $\Phi(G'')$ erfüllt, auch $\Phi(G')$ erfüllt, d.h. $\Gamma \cup \Phi(G'') \models \Phi(G')$. Der Graph G' ist hierdurch eine subsumierte Alternative von G. Der Graph G' ist minimal, da dieser Konzeptgraph nur eine Alternative besitzt.

7.4 Wissensbasierte Definition der Typkonformität

Gegenstand dieses Abschnitts ist die Diskussion der wissensbasierten Typkonformität für Konzeptgraphen. In Abschnitt 7.4.2 wird zunächst eine syntaktische und eine semantische Definition der Typkonformität vorgestellt. Diese beiden Definitionen sind generisch in dem Sinne, daß sie auf beliebige Konzeptgraphen anwendbar sind. Es zeigt sich jedoch, daß die Typkonformität in Abhängigkeit von der Spezifikation, die ein Konzeptgraph repräsentiert, liberaler interpretiert werden kann. Das resultiert in Abschnitt 7.4.1 in der Definition von Vergleichsregeln, die eine Parametrisierung bei der Überprüfung der Typkonformität erlauben.

7.4.1 Syntaktische und semantische Definition der Typkonformität

Die in diesem Abschnitt vorgestellten Definitionen der syntaktischen und semantischen Typkonformität sind generisch in dem Sinne, daß sie unabhängig von der Art der Diensttypen sind, die durch Konzeptgraphen spezifiziert werden. Für eine konstruktive Herleitung der syntaktischen Typkonformität sei zunächst folgender Konzeptgraph G gegeben:

```
[PRINTER:{HP-DESKJET}] -> (PRINTS-ON) -
                                -> [PAPER],
                                -> [SLIDES]
                                .
```

Durch Hinzufügen von weiteren Teilästen kann G dergestalt erweitert werden, daß sich der Konzeptgraph aus Abbildung 7.4 ergibt. Der Konzeptgraph G wird dabei um zusätzliche Alternativen erweitert. Außerdem werden bereits bestehende Alternativen durch Teilgraphen erweitert. Der obige Konzeptgraph gleicht einer „verkürzten Alternative“. Er bildet eine Teilmenge des Konzeptgraphen aus Abbildung 7.4, die sowohl weniger Alternativen, als auch verkürzte Alternativen besitzt. Der Konzeptgraph aus Abbildung 7.4 ist hierdurch eine verfeinerte Variante des Konzeptgraphen G. G besitzt weniger Extensionen und die übereinstimmenden Extensionen sind allgemeiner, da sie auf eine Verfeinerung der tiefer im Graphen vorkommenden Konzepte verzichtet.

Aufgrund der Definition des Φ–Operators, der einen Konzeptgraphen als Approximation einer Intension definiert, ist der Konzeptgraph aus Abbildung 7.4 konform zu dem oben angegebenen Konzeptgraphen G. Diese Form der Typkonformität läßt sich anhand der syntaktischen Struktur zweier Konzeptgraphen entscheiden. Die folgende Definition formalisiert die syntaktische Typkonformität von zwei Konzeptgraphen.

Definition 7.16 (Syntaktische Typkonformität): Seien $G = (N, L, F)$ und $G' = (N', L', F')$ Konzeptgraphen. G' ist *syntaktisch typkonform* zu G, geschrieben $G' \leq_{SYN} G$, wenn eine injektive Abbildung $f : N \rightarrow N'$ existiert, für die gilt:

1. $f(root(G)) = root(G')$
2. $\forall (n, p) \in F : (f(n), p) \in F'$
3. $\forall (n_1, n_2) \in L : (f(n_1), f(n_2)) \in L'$

□

Ein Konzeptgraph G' ist syntaktisch typkonform zu einem Konzeptgraph G, wenn die Knotenmenge von G injektiv auf die von G' abgebildet werden kann, so daß die Abbildung strukturerhaltend ist. Die Abbildung ist strukturerhaltend, wenn der Wurzelknoten von G auf den von G' abgebildet wird, und alle weiteren Knoten von G gleiche Bilder in G' sowie übereinstimmende Kanten besitzen. Das Bild, das durch die Abbildung f aus Definition 7.16 erzeugt wird, entspricht nicht notwendigerweise einer Alternative. Zu einer Alternative gehört ausgehend von dem Wurzelknoten des Konzeptgraphen immer die transitive Hülle entlang *eines* Teilgraphen bis zu den Blättern des Graphen. Abbildung

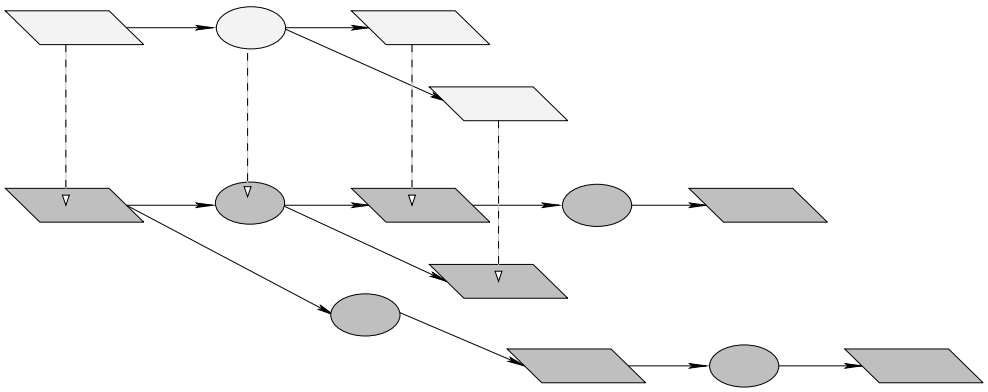


Abbildung 7.8: Vergleich von zwei Konzeptgraphen.

7.8 gibt ein Beispiel von zwei typkonformen Konzeptgraphen. Der untere Konzeptgraph ist syntaktisch typkonform zu dem oberen. Die gestrichelten Pfeile spiegeln gerade die injektive Abbildung f aus Definition 7.16 wider.

Die syntaktische Typkonformität, die durch den $\leq_{SYN}$–Operator definiert ist, induziert eine Halbordnung auf der Menge CG der Konzeptgraphen. Jedes Konzept der Menge C als Wurzelknoten eines Konzeptgraphen definiert eine eigene Hierarchie. Da die syntaktische Typkonformität lediglich Bezug auf die Struktur eines Konzeptgraphen nimmt, existiert kein eindeutig bestimmtes größtes Element der durch den $\leq_{SYN}$–Operator aufgespannten Halbordnung.

Der folgende Satz stellt einen Zusammenhang zwischen der syntaktischen Typkonformität und dem Φ–Operator her. Ist ein Konzeptgraph G' syntaktisch konform zu einem Konzeptgraphen G, dann ist die Formel $\Phi(G)$ logische Konsequenz einer Formel, die sich aus bestimmten Alternativen des Graphen G' ergibt.

Satz 7.1 (Interpretation der syntaktischen Typkonformität): Seien G und G' Konzeptgraphen. Ist $G' \leq_{SYN} G$, dann existiert eine nicht leere Menge von Alternativen $G'_1, \ldots, G'_n$ von G' mit $\{\Phi(G'_1), \ldots, \Phi(G'_n)\} \models \Phi(G)$.

Beweis (informell): Wenn $G' \leq_{SYN} G$ gilt, existiert nach Definition 7.16 eine injektive Abbildung f von G nach G'. Wähle die Alternativen $G'_1, \ldots, G'_n$ gerade so, daß jede Alternative mindestens einen Knoten aus dem Bild von f besitzt. Wenn eine Interpretation $\{\Phi(G'_1), \ldots, \Phi(G'_n)\}$ erfüllt, so muß diese aber auch $\Phi(G)$ erfüllen, da $\Phi(G)$ weniger Konjunktionen als $\{\Phi(G'_1), \ldots, \Phi(G'_n)\}$ enthält. ■

Die syntaktische Typkonformität vergleicht zwei Konzeptgraphen, indem versucht wird, die Knotenmenge sowie die Struktur des einen Graphen injektiv auf den anderen abzubilden. Die Definition 7.16 setzt gleiche Konzeptnamen C, Instanzennamen IN, Relationsnamen R und Stelligkeitsfunktion f_S voraus. Bei der Einbettung eines Konzeptgraphen in einen anderen mittles der Funktion f werden syntaktisch gleiche Konzept– und Relationsknoten aufeinander abgebildet. Diese ausschließlich syntaktische Definition der Typkonformität kann mit einer Ontologie weiter flexibilisiert werden. Beispielsweise ist der Konzeptgraph aus Abbildung 7.4 typkonform zu

```
[SOMETHING] -> (VISUALIZES) -> [KNOWLEDGE]
```

wenn eine Ontologie wie in Abbildung 7.7 zugrunde gelegt wird. Intuitiv ergibt sich die Aussage aus der Tatsache, daß ein Drucker Wissen (*knowledge*) visualisiert, da er auch Informationen (*information*) visualisiert. Es ist zu beachten, daß der Konzeptgraph aus Abbildung 7.4 nicht syntaktisch typkonform zu obigem Konzeptgraph ist. Der Konzeptgraph in Abbildung 7.4 geht aus dem obigen Graphen durch wohldefinierte Operationen hervor. Analog zu der syntaktischen Typkonformität, kann der obige Graph zunächst struktuell erweitert werden. Darüberhinaus können seine Knoten bzgl. einer Ontologie spezialisiert werden. Das Konzept `[PRINTER:{HP-DESKJET}]` ist eine Spezialisierung des Konzepts `[SOMETHING]`. Das gleiche gilt für die Konzeptknoten `[INFORMATION]` und `[KNOWLEDGE]`.

Aus diesem Beispiel läßt sich eine semantische Typkonformität für Konzeptgraphen ableiten, die die syntaktische Variante strikt erweitert. Zwei Konzeptgraphen, die syntaktisch typkonform sind, sind ebenfalls semantisch typkonform. Die Umkehrung dieser Aussage gilt nicht. Die formale Definition der semantischen Typkonformität nimmt Bezug auf eine Ontologie.

Definition 7.17 (Semantische Typkonformität): Seien $G = (N, L, F)$ und $G' = (N', L', F')$ Konzeptgraphen. Ferner sei Γ eine Ontologie. G' ist *semantisch typkonform* zu G bzgl. Γ, geschrieben $G' \leq_{SEM} G$, wenn eine nicht leere Menge von Alternativen $G'_1, \ldots, G'_n$ von G' existiert, für die gilt $\Gamma \cup \{\Phi(G'_1), \ldots, \Phi(G'_n)\} \models \Phi(G)$. □

Die Definition der semantischen Typkonformität zwischen zwei Konzeptgraphen ähnelt der Interpretation der syntaktischen Typkonformität aus Satz 7.1. Dort wird die Typkonformität ebenfalls auf den semantischen Folgerungsbegriff zurückgeführt. Der Unterschied zwischen der syntaktischen und der semantischen Typkonformität ist, daß bei der letzteren eine Ontologie berücksichtigt wird. Mit der Definition für die semantische Typkonformität kann die Abbildung 7.8 so interpretiert werden, daß die Knoten des oben dargestellten Konzeptgraphen Verallgemeinerungen bzgl. einer Ontologie der übereinstimmenden Knoten des unteren Konzeptgraphen sind.

Beispiel 7.4: Gegeben sei der Konzeptgraph G:

```
[SOMETHING] -> (VISUALIZES) -> [KNOWLEDGE]
```

G kann mit dem Φ–Operator abgebildet werden auf:

$$\exists\, x\, \exists\, y(\text{SOMETHING}(x) \wedge \text{VISUALIZES}(x, y) \wedge \text{KNOWLEDGE}(y))$$

Die Ontologie Γ sei definiert durch die Konzepthierarchie aus Abbildung 7.7 mit $F_R = \emptyset$. Folglich gilt:

$$\{\forall\, x \in I : (\text{PRINTER}(x) \Rightarrow \text{SOMETHING}(x)),$$
$$\forall\, x \in I : (\text{INFORMATION}(x) \Rightarrow \text{KNOWLEDGE}(x))\} \in \Gamma$$

Sei G' die folgende Alternative des Konzeptgraphen aus Abbildung 7.4:

```
[PRINTER:{HP-DESKJET}] -> (VISUALIZES) -> [INFORMATION]
```

mit $\Phi(G')$:

$$\exists\, x(\text{PRINTER}(\text{HP–DESKJET}) \wedge \text{VISUALIZES}(\text{HP–DESKJET}, y) \wedge \text{INFORMATION}(x))$$

Da $\Gamma \cup \{\Phi(G')\} \models \Phi(G)$ gilt, folgt $G' \leq_{SEM} G$.

Die semantische Typkonformität induziert eine Halbordnung auf der Menge der Konzeptgraphen. Im Unterschied zu der Halbordnung, die durch den $\leq_{SYN}$–Operator induziert wird, definiert die durch $\leq_{SEM}$ aufgespannte Halbordnung *eine* Hierarchie. Das eindeutig bestimmte größte Element der Hierarchie ist der Konzeptgraph `[SOMETHING]`. Unabhängig von der Wahl der Ontologie ist jeder Konzeptgraph semantisch konform zu `[SOMETHING]`. Der Grund dafür ergibt sich aus der Tatsache, daß `SOMETHING` das Wurzelelement der Konzepthierarchie T_C ist und hierdurch von jedem Konzeptnamen bzgl. einer Ontologie Γ logisch impliziert wird. D.h., für jede Alternative G' eines beliebigen Konzeptgraphen G gilt, daß $\Gamma \cup \{\Phi(G')\} \models \Phi($`[SOMETHING]`$)$ ist.

7.4.2 Typkonformität auf Basis von Vergleichsregeln

Mit der Theorie der Konzeptgraphen lassen sich beliebige Aussagen formal darstellen. Eine Typspezifikation ist dabei nur eine mögliche Ausprägung einer Aussage. Im folgenden Kapitel wird diese Tatsache dazu ausgenutzt, operationale Schnittstellenspezifikationen in Form von Konzeptgraphen darzustellen. Ein wissensbasierter Dienstvermittler bedient sich ausschließlich der Konzeptgraphen für die Zuordnung von Angebot und Nachfrage. Gerade das Beispiel der Kodierung von operationalen Schnittstellenspezifikationen in Form von Konzeptgraphen legt den Schluß nahe, daß die syntaktische und semantische Typkonformität, wie sie hier definiert wurde, nicht ausreichend für den Vergleich von Konzeptgraphen ist. Bereits in Kapitel 4 wurde bei der Diskussion der Typsysteme von DCE, CORBA und ODP hervorgehoben, daß diese sich u.a. in ihrer Definition der Typkonformität unterscheiden. Die Konsequenz daraus ist, daß unterschiedliche Vergleichsregeln für Konzeptgraphen gefordert werden müssen. Der Dienstnutzer muß für eine Import–Operation angeben, bzgl. welcher Vergleichsregeln der Vermittler einen typkonformen Diensttyp ermitteln soll.

Definition 7.18 (Vergleichsregel für Konzeptgraphen): Eine Vergleichsregel φ ist definiert als eine Abbildung mit $\varphi \in$ ABB$(CG \times CG \times \Gamma, \{$`accept`, `reject`, `unknown` $\})$. Die Vergleichsregel φ vergleicht zwei Konzeptgraphen bzgl. einer Ontologie Γ und entscheidet, ob die beiden Konzeptgraphen typkonform sind (`accept`), nicht typkonform sind (`reject`) oder die Typkonformität nicht entschieden werden kann (`unknown`). □

Eine Vergleichsregel entscheidet aufgrund zweier Konzeptgraphen und einer Ontologie, ob beide Konzeptgraphen typkonform sind. Auf Basis der in Definition 7.17 vorgestellten semantischen Typkonformität läßt sich eine entsprechende Vergleichsregel definieren:

$$\varphi_{SEM}(G_1, G_2, \Gamma) =_{df} \begin{cases} \texttt{accept} & : \quad G_1 \leq_{SEM} G_2 \text{ bzgl. } \Gamma \\ \texttt{unknown} & : \quad \text{sonst} \end{cases}$$

Es soll die Möglichkeit nicht ausgeschlossen werden, daß die Typkonformität aufgrund mehrerer Vergleichsregeln festgestellt wird. Beispiel dafür ist die Kombination der Vergleichsregel für semantische Typkonformität und einer noch vorzustellenden Vergleichsregel für Negation. Die Vergleichsregel für Negation untersucht widersprüchliche Extensionen in Konzeptgraphen und kann für diesen Fall die Typkonformität explizit

ausschließen. Ist die Typkonformität zwischen Konzeptgraphen über mehrere Vergleichsregeln definiert, so wird jede Vergleichsregel einzeln ausgewertet. Die Typkonformität ist genau dann hergestellt, wenn mindestens eine Vergleichsregel `accept` liefert und keine Vergleichsregel `reject` liefert. Die Typkonformität ist genau dann nicht hergestellt, wenn eine Vergleichsregel `reject` liefert oder alle Vergleichsregeln `unknown` liefern. Bei dieser Definition dominiert `reject` gegenüber `accept` und `unknown`. Im folgenden Kapitel werden unterschiedliche Vergleichsregeln vorgestellt und formal spezifiziert.

7.5 Zusammenfassung

In offenen verteilten Systemen treten Anwender in den Rollen der Dienstnutzer und –anbieter auf. Sie nehmen aktiv an der Vermittlung von Diensten teil. Die Voraussetzung dafür ist ein Typsystem, das Typspezifikationen auf der Abstraktionsebene der Anwender unterstützt. Die in früheren Kapiteln diskutierten Schnittstellenbeschreibungssprachen sind dafür ungeeignet. Für den Rahmen dieser Arbeit kommen die Konzeptgraphen als Typbeschreibungssprache zum Einsatz. Das Konstruktionsprinzip eines Konzeptgraphen leitet sich aus der Wahrnehmungspsychologie ab. Ihre Ausdruckskraft entspricht einer Teilmenge der Prädikatenlogik erster Stufe. Dadurch, daß Konzeptgraphen eine graphische Darstellung besitzen, erleichtert das die Visualisierung von wissensbasierten Typspezifikationen.

Im Unterschied zu einem deklarativen Typsystem stellt ein Konzeptgraph lediglich eine Approximation der Intension eines Diensttyps dar. Die Alternativen eines Konzeptgraphen repräsentierten unterschiedliche Extensionen, die jedem Anwender ihre eigene Sicht auf einen Diensttyp erlauben. Die Semantik eines Konzeptgraphen ist definiert durch den Φ–Operator, der einen Konzeptgraphen auf eine prädikatenlogische Formel erster Stufe abbildet. Ein Konzeptgraph steht bzgl. einer Ontologie in Beziehung zu anderen Konzeptgraphen. Eine Ontologie enthält allgemeingültige Aussagen über Konzept– und Relationsnamen. Die Ontologie ist Voraussetzung für die Definition der semantischen Typkonformität.

Kapitel 8

Wissensbasierte Dienstvermittlung

Ein wissensbasiertes Typsystem ermöglicht die Spezifikation von Diensttypen, die auf der Abstraktionsebene der Anwender angesiedelt sind. Das in dieser Arbeit vorgestellte wissensbasierte Typsystem basiert auf der Theorie der Konzeptgraphen. Ein Konzeptgraph als Diensttypspezifikation enthält mehrere Extensionen die die Intension des Diensttyps approximieren. Ein Vermittler soll als *wissensbasierter Dienstvermittler* bezeichnet werden, wenn er Diensttypspezifikationen auf Basis eines wissensbasierten Typsystems verarbeitet. Die wissensbasierte Dienstvermittlung, die neben der Vermittlung von Instanzen auch die Vermittlung von Typen zum Ziel hat, ist Gegenstand dieses Kapitels. Wissensbasierte Typspezifikationen entspringen vagen Vorstellungen der an einem Vermittlungsvorgang beteiligten Anwender. Da ein wissensbasierter Vermittler nie vollständige Informationen über einen Diensttyp besitzt, muß für das inkrementelle Hinzufügen von weiteren Extensionen ein Lernvorgang vorausgesetzt werden, welcher auch Auswirkungen auf die Interaktionen zwischen Dienstnutzern und –anbieter mit dem Vermittler hat.

Die unterschiedlichen Abläufe, die sich unter der *Closed World Assumption* und der *Open World Assumption* bei der Vermittlung von Typen und Instanzen ergeben, sind Gegenstand des Abschnitts 8.1. In Abschnitt 8.2 wird ein abstraktes Modell für die Typvermittlung eingeführt. Anschließend wird die Architektur eines wissensbasierten Dienstvermittlers in Abschnitt 8.3 vorgestellt. Die dynamischen Aspekte der wissensbasierten Dienstvermittlung, die in Form von Import– und Export–Protokollen beschrieben werden, folgen in Abschnitt 8.4. Die Abschnitte 8.5 und 8.6 demonstrieren Anwendungen der wissensbasierten Dienstvermittlung in den Bereichen der natürlichen Sprachen und der Schnittstellenbeschreibungssprachen. Eine Zusammenfassung in Abschnitt 8.7 beschließt das Kapitel.

8.1 Abläufe bei der wissensbasierten Dienstvermittlung

Während eines Vermittlungsvorgangs kooperieren drei Objekte in den Rollen eines Dienstnutzers, eines Dienstanbieters und eines Vermittlers. Ziel der Vermittlung ist nach Abschnitt 3.3.2 auf Seite 26 die Herstellung der Sichtbarkeit und der Erreichbarkeit. Je nachdem, ob für ein offenes verteiltes System die *Closed World Assumption* oder die

Open World Assumption zugrundegelegt wird, ergeben sich unterschiedliche Abläufe zwischen den an einem Vermittlungsvorgang beteiligten Objekten. Insbesondere im Hinblick auf die *Open World Assumption* leiten sich aus den im folgenden diskutierten Abläufen Anforderungen an einen wissensbasierten Dienstvermittler ab. Die Kooperation zwischen Dienstnutzer, Dienstanbieter und Vermittler wird in den Abschnitten 8.1.1 und 8.1.2 getrennt für die *Closed World Assumption* und die *Open World Assumption* analysiert.

8.1.1 Kooperation unter der *Closed World Assumption*

Unter der *Closed World Assumption* wird die Typvermittlung implizit vorausgesetzt. Ein Diensttyp nimmt hierbei den Charakter eines Standards an, auf den sich alle an einem offenen verteilten System partizipierenden Parteien einigen müssen. Die erfolgreiche Vermittlung von Instanzen setzt Absprachen bei deren Diensttypspezifikationen voraus. Um eine *a–priori*–Kenntnis der Diensttypspezifikationen zwischen Dienstanbietern und –nutzern zu erreichen, sind zwei Schritte notwendig:

1. Festlegung eines Standards für einen Diensttyp.
2. Bekanntmachen des Standards für den Diensttyp.

Die Standardisierung von Diensttypen kann zentral oder dezentral erfolgen, wobei bei letzterem darauf geachtet werden muß, daß die Eindeutigkeit der Typspezifikationen gewahrt bleibt. Nach der Standardisierung müssen die Typspezifikationen interessierten Dienstnutzern bekannt gemacht werden. Ist ein Dienstnutzer in Besitz einer Typspezifikation, so kann er diese als Parameter einer Import–Operation an den Vermittler verwenden. Da aufgrund der Standardisierung globaler Konsens über Typspezifikationen herrscht, kann der Dienstvermittler einen typkonformen Dienst dem Dienstnutzer zuordnen.

Vorteil der *Closed World Assumption* ist, daß aufgrund der Standardisierung von Diensttypspezifikationen die Zuordnung zwischen Angebot und Nachfrage zweifelsfrei ist. Es ist nicht möglich, daß wegen unvollständiger Information über die Intension eines Diensttyps ein Dienst zugeordnet wird, der die Anforderungen des Dienstnutzers nicht erfüllt. Ein Kooperationsvorgang zwischen einem Dienstnutzer und –anbieter durchläuft bei *a–priori*–Absprachen bzgl. der Typspezifikationen die Aktionsfolge Vermittlung, Bindung und Interaktion nur einmal (siehe obere Hälfte der Abbildung 8.1). Wegen der globalen Eindeutigkeit von Typspezifikationen kann die Vermittlung und Bindung von Dienstanbieter und –nutzer transparent aus Sicht der Anwender geschehen, die die Dienstnutzer steuern (siehe [52]). Ein Anwender nimmt in diesem Fall nicht aktiv an einem Vermittlungsvorgang teil.

8.1.2 Kooperation unter der *Open World Assumption*

Annahme der *Open World Assumption* ist, daß die Standardisierung von Diensttypen in offenen verteilten Systemen nur eingeschränkt möglich ist. Eine implizite Typvermittlung durch Standardisierung der Typspezifikationen bedingt einen erheblichen Aufwand bei der Bekanntmachung dieser Standards an interessierte Dienstnutzer. Aufgabe der Vermittlung von Typen ist es, diesen Aufwand mit Hilfe geeigneter Mechanismen zu automatisieren

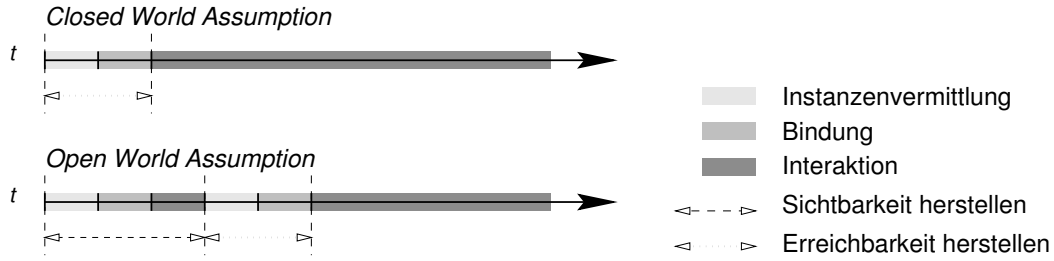


Abbildung 8.1: Unterschiedliche Aktionsabfolge unter der *Closed World* und *Open World Assumption*.

und hierdurch die Notwendigkeit von *a–priori*–Absprachen zu reduzieren. Erreichen läßt sich dies mit einem wissensbasierten Typsystem, wie es im letzten Kapitel beschrieben wurde.

Die Typvermittlung ist eine der Instanzenvermittlung vorgelagerte Phase, die einem Anwender zunächst die Kenntnis (d.h. Sichtbarkeit) eines Diensttyps aufgrund seiner vagen Vorstellungen vermittelt. Wegen der zunächst unklaren Vorstellungen eines Anwenders kann nicht garantiert werden, daß die Typvermittlung bereits im ersten Schritt zu dem gewünschten Typ führt. Es muß vielmehr davon ausgegangen werden, daß der Anwender erst bei der Interaktion mit einem Dienstanbieter merkt, daß dieser seine Anforderungen bzgl. einer Funktionalität nicht erfüllt. Die Typvermittlung muß hierdurch wiederholt werden, um die Spezifikation des Dienstwunsches des Anwenders zu verfeinern.

Die einmalige Aktionsabfolge Vermittlung, Bindung und Interaktion bei der Instanzenvermittlung muß bei der Typvermittlung u.U. mehrfach wiederholt werden (siehe untere Hälfte der Abbildung 8.1). Der Anwender erschließt die Semantik eines Dienstes während er mit einem Dienstanbieter interagiert. Bemerkt er, daß die von dem Dienstanbieter angebotene Funktionalität nicht seinen Vorstellungen entspricht, muß die Typvermittlung wiederholt werden. Das setzt voraus, daß die bisherigen Interaktionen mit dem ungeeigneten Dienstanbieter keine Auswirkungen haben. Dies läßt sich beispielsweise mit Hilfe eines Transaktionssystems bewerkstelligen, mit dem mögliche Auswirkungen bei der Interaktion eines Anwenders mit einem Dienstanbieter rückgängig gemacht werden können.

Die Typvermittlung schließt den Anwender in den Vermittlungsvorgang aktiv mit ein. Die vagen Vorstellungen eines Anwenders bzgl. der von ihm gewünschten Funktionalität erlauben keine vollständige Automatisierung der Typvermittlung. Letztendlich kann nur der Anwender durch seine Interaktion mit einem Dienstanbieter entscheiden, ob dieser seinen Dienstwunsch erfüllt oder nicht. Ein Dienstvermittler, der die Typvermittlung unterstützt, muß den Anwendern die Verfeinerung von Typspezifikationen erlauben.

8.2 Abstraktes Modell der Typvermittlung

Die Typvermittlung resultiert in mehreren Interaktionen zwischen dem Dienstanbieter und dem Dienstnutzer mit einem Vermittler. Ausgehend von der Annahme, daß Konzeptgraphen für die Diensttypspezifikation verwendet werden, beschreibt dieser Abschnitt ein abstraktes Modell der Typvermittlung. Diensttypspezifikationen auf Basis der Konzept-

graphen stellen eine Approximation der Intension des Diensttyps dar. Ein Konzeptgraph vereinigt unterschiedliche Sichten, die Anwender auf einen Dienst haben. Unter der *Open World Assumption* hängt der Detaillierungsgrad in der Beschreibung einer Sicht vom subjektiven Standpunkt eines Betrachters ab.

Definition 8.1 (Sicht): Eine *Sicht* entspricht einer semantischen Diensttypspezifikation, die die Präferenzen desjenigen Anwenders berücksichtigt, der den Dienst beschreibt. Diese Präferenzen basieren auf seinem Wissen über den Dienst und den Anforderungen, die er im Rahmen seiner Anwendung an diesen stellt. □

Qualitative Unterschiede in der Spezifikation eines Diensttyps leiten sich aus den Rollen der Teilnehmer an einem Vermittlungsvorgang ab. Der Dienstanbieter besitzt vollständige Kenntnis über seinen Dienst und wird ihn daher präziser und detaillierter beschreiben als ein Dienstnutzer, der nicht über alle Informationen verfügt und deshalb eine unvollständige Vorstellung über den gesuchten Dienst hat (siehe [33]). Er hat jedoch eine vage Vorstellung davon, was der gesuchte Dienst leisten soll, und beschreibt daher den Dienst aus seiner subjektiven Sicht. Eine Aufgabe der Typvermittlung ist es, aus den unterschiedlichen individuellen Sichten eine Spezifikation eines Diensttyps in Form eines Konzeptgraphen zu konstruieren.

Da ein Anbieter nicht alle Sichten der Dienstnutzer vorhersehen kann, muß die Typspezifikation eines Dienstes in einem mehrstufigen Prozeß aus diesen verschiedenen Sichten konstruiert werden. An diesem Prozeß müssen die Dienstnutzer beteiligt werden, damit ihre Sicht, falls diese von der Approximation der Intension eines Diensttyps noch nicht berücksichtigt ist, zur Erweiterung dieser verwendet wird. Der Aufwand zur Konstruktion der generellen Sicht nimmt im Laufe der Zeit ab, da immer mehr Sichten aufgenommen werden. Ein Ende dieses Erweiterungsprozesses kann jedoch nicht angegeben werden, da es i.d.R. unendlich viele Sichten auf einen Dienst gibt. Dies hat für die Konzeption eines wissensbasierten Vermittlungsprotokolls zur Folge, daß der Dienstexport nicht ein auf einen Zeitpunkt beschränkter Vorgang ist. Er umfaßt mehrere Phasen, in denen die verschiedenen Sichten berücksichtigt werden, so daß das Protokoll zugleich Phasen zur Änderung und Erweiterung einer exportierten Diensttypspezifikation enthalten muß.

Die Typvermittlung wäre nicht möglich, wenn sich alle Sichten eines Dienstes vollkommen voneinander unterscheiden würden. Eine wissensbasierte Dienstvermittlung, die die Automatisierung der Typvermittlung zum Ziel hat, kann nur möglich sein, wenn sich bei einer signifikanten Anzahl von individuellen Sichten eine gemeinsame Schnittmenge ausbildet. Einen Hinweis für diese Eigenschaft entstammt der Linguistik. Unter dem Begriff *semantische Merkmale* werden dort jene Komponenten bezeichnet, aus denen sich eine Wortbedeutung elementar zusammensetzt (siehe [106], [61]). Die semantischen Merkmale bilden die Schnittmenge der elementaren Komponenten, die eine Person mit hoher Wahrscheinlichkeit einer Wortbedeutung zuweist. Die Existenz der semantischen Merkmale dient als Motivation für die Definition eines Kernbereichs:

Definition 8.2 (Kernbereich): Ein *Kernbereich* definiert die grundlegenden funktionalen und technischen Eigenschaften eines Dienstes. Er ergibt sich aus der Schnittmenge „sinnvoller“ Sichten eines Dienstes. Es ist die Aufgabe des Dienstanbieters zu entscheiden, wann eine Sicht sinnvoll ist. □

Ein Dienstanbieter muß bei der initialen Spezifikation seines Diensttyps den Kernbereich identifizieren. Die initiale Typspezifikation bleibt jedoch nur eine Approximation der Intension, die zu einem späteren Zeitpunkt um individuelle Sichten eines Dienstnutzers erweitert werden muß. Untersuchungen von Dienstnutzern bei der Informationssuche haben gezeigt, daß diese beim Beginn der Suche nur eine unvollständige Vorstellung über den gewünschten Diensttyp haben und die Ergebnisse ihrer Anfrage verwenden, um diese zu reformulieren (siehe [33]). Auf Seiten des Dienstnutzers findet hierdurch ebenfalls eine Verfeinerung der Diensttypspezifikation statt. Der Grad der Verfeinerung orientiert sich an dem quantitativen Angebot bei den Diensttypen. Um einzelne Diensttypspezifikationen voneinander abzugrenzen, wird das Konzept von zutreffenden und nicht zutreffenden Eigenschaften der semantischen Merkmale für Sichten eingeführt.

Definition 8.3 (Widersprüche zwischen Sichten): Ein *Widerspruch* zwischen zwei Sichten liegt dann vor, wenn eine Sicht eine Eigenschaft als zutreffend kennzeichnet, die in einer anderen Sicht als nicht zutreffend angegeben ist. □

Anhand der im letzten Kapitel eingeführten semantischen Typkonformität für Konzeptgraphen kann ein Widerspruch zwischen Sichten nicht hergeleitet werden. Dies ist erst mit einer speziellen Vergleichsregel möglich, die in Abschnitt 8.5.3 vorgestellt werden wird. Widersprüche zwischen Sichten können erkannt, aber nicht aufgelöst werden, da das Problem, ob ein Konzeptgraph eine korrekte Spezifikation eines Dienstes ist, nicht maschinell überprüfbar ist. Diese Aufgabe obliegt dem Dienstanbieter, der Experte bzgl. seines Dienstes ist. Tritt ein Widerspruch zwischen zwei Sichten auf, die ein Dienstanbieter dem wissensbasierten Dienstvermittler während eines Exports übergibt, so handelt es sich um eine fehlerhafte Spezifikation, die der Dienstanbieter auflösen muß. Treten Widersprüche zwischen den Sichten eines Dienstanbieters und eines Dienstnutzers während eines Imports auf, so wird davon ausgegangen, daß es sich um unterschiedliche Diensttypspezifikationen handelt. Die Erkennung von Widersprüchen reduziert die Menge der typkonformen Diensttypen und hierdurch den Suchraum aus Dienstnutzersicht.

Da i.d.R. unendlich viele Extensionen eines Diensttyps existieren, ist es möglich, daß der wissensbasierte Dienstvermittler keinen typkonformen Diensttyp als Antwort auf eine Import–Operation ermitteln kann, obwohl ein passender Dienst exportiert wurde. Es muß in diesem Fall möglich sein, daß vollständige Dienstangebot manuell zu durchsuchen. Voraussetzung für diesen Vorgang ist die Diensttyphierarchie.

Definition 8.4 (Diensttyphierarchie): Die *Diensttyphierarchie* besteht aus einer Menge von Typspezifikationen, die in einer Hierarchie angeordnet sind. Die Anordnung der Typspezifikationen basiert auf einem Klassifikationsschema, die den Anwendern die systematische Traversierung des Typraums erlaubt. Alle exportierten Typen sind in der Diensttyphierarchie enthalten. □

Jeder Import kann als ein Test angesehen werden, bei dem geprüft wird, ob die Sicht eines Dienstnutzers mit der Sicht eines Dienstanbieters übereinstimmt. Ist dies nicht der Fall, erlaubt die manuelle Navigation der Diensttyphierarchie dem Dienstnutzer alle exportierten Diensttypen durchzusehen und zu entscheiden, welcher Dienst seinen Vorstellungen entspricht (siehe [33]). Zu den Aufgaben eines Dienstanbieters gehört das Einfügen seines Diensttyps in die Diensttyphierarchie.

Die manuelle Navigation kann auch dann notwendig sein, wenn sich aufgrund der Dynamik des offenen verteilten Systems das Dienstangebot spezialisiert. In diesem Fall wird der Dienstnutzer gezwungen, seine Vorstellungen in gleichem Maße zu spezialisieren. Eine Typspezifikation, mit der ein Dienstnutzer einen Dienst eindeutig identifizieren konnte, kann sich zu einem späteren Zeitpunkt als zu allgemein erweisen. Als Ergebnis erhält der Dienstnutzer eine Liste von typkonformen Diensttypen. Je nach Umfang der Liste muß der Dienstnutzer sich der manuellen Navigation bedienen, um seine Anfrage zu verfeinern.

8.3 Architektur eines wissensbasierten Dienstvermittlers

Im folgenden wird die Architektur eines wissensbasierten Dienstvermittlers vorgestellt, der, neben der Vermittlung von Instanzen, die Typvermittlung unterstützt. Die während eines Vermittlungsvorgangs notwendigen Typspezifikationen, die als Parameter in Import– und Export–Operationen enthalten sind, basieren auf den im letzten Kapitel vorgestellten Konzeptgraphen. Während in diesem Abschnitt die Komponenten des wissensbasierten Dienstvermittlers erläutert werden, folgt die Diskussion der dynamischen Aspekte in Form eines Vermittlungsprotokolls im nächsten Abschnitt.

Es wird von einer Integration des wissensbasierten Dienstvermittlers auf Anwendungsebene ausgegangen. Die anwendungsbezogene Modellierung weist einem Objekt die Rolle eines wissensbasierten Dienstvermittlers zu. Objekte, die in den Rollen von Dienstnutzern und –anbietern auftreten, besitzen implizit eine Referenz auf den Vermittler, was ihn innerhalb einer Typdomäne zu einem Objekt mit einer wohlbekannten Identität macht. Der Vermittler bietet an seiner Schnittstelle Operationen für den Import und Export von Diensten an. Da jedes Objekt in einem offenen verteilten System eine Referenz auf den Vermittler besitzt, können Import– und Export–Aufforderungen an diesen gerichtet werden.

Der wissensbasierte Dienstvermittler besteht aus vier Komponenten, der *Dienstdatenbank*, den *Vergleichsregeln*, der *lexikographischen Datenbank* und dem *Dispatcher* (siehe 8.2). In der Dienstdatenbank speichert der Dienstvermittler alle Dienstangebote, die Anbieter über eine Export–Operation zuvor bekannt gemacht haben. Ein Eintrag in der Dienstdatenbank entspricht einem Tupel bestehend aus einer Typspezifikation und einer Menge von Referenzen. Die Referenzen verweisen auf Dienstanbieter, deren Typ durch den ersten Eintrag des Tupels gegeben ist. Eine Export–Operation erzeugt entweder einen neuen Eintrag in der Dienstdatenbank in Form eines Tupels oder erweitert die Menge der Referenzen eines Tupels, wenn der Diensttyp dem Vermittler bereits bekannt ist.

Sowohl der Import als auch der Export von Diensten setzt den Vergleich zweier Konzeptgraphen gemäß der Definition der Typkonformität eines wissensbasierten Typsystems voraus. Während des Imports ermittelt der Vermittler einen Dienst, der zu der Anfrage typkonform ist, wohingegen bei einem Export überprüft wird, ob der Diensttyp bereits in der Dienstdatenbank vorhanden ist. Bereits im letzten Kapitel wurde angedeutet, daß neben der semantischen Typkonformität für Konzeptgraphen weitere Vergleichsregeln existieren. Eine Vergleichsregel entscheidet anhand zweier Konzeptgraphen, ob diese typkonform zueinander sind oder nicht. Da in Abhängigkeit vom Anwendungskontext

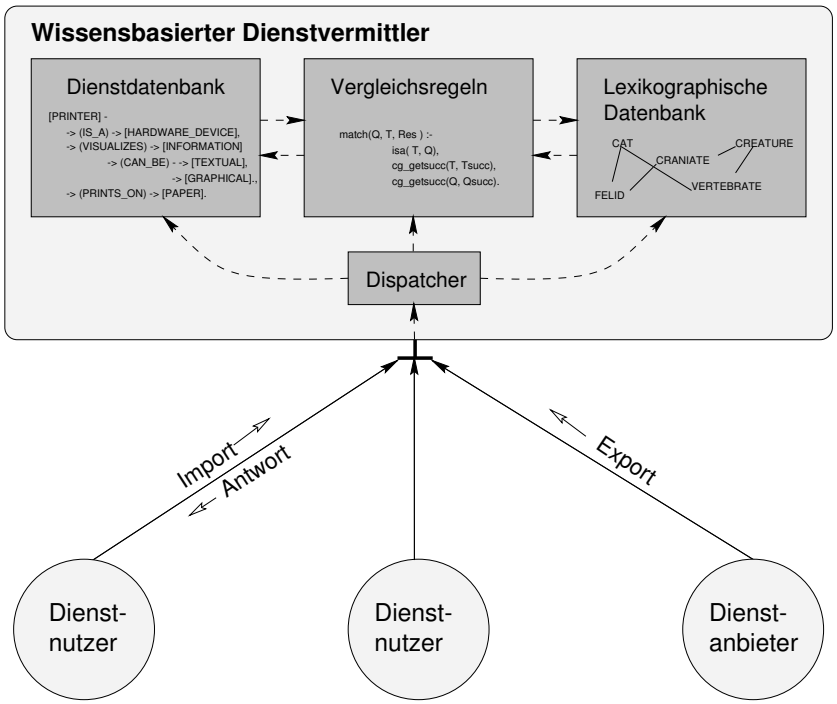


Abbildung 8.2: Architektur eines wissensbasierten Dienstvermittlers.

unterschiedliche Vergleichsregeln existieren, von denen einige in den Abschnitten 8.5 und 8.6 noch vorgestellt werden, erlaubt der Vermittler das Hinzufügen von Vergleichsregeln zur Laufzeit. Eine Implementation eines wissensbasierten Dienstvermittlers bedingt einen Interpreter für die Ausführung von Vergleichsregeln zur Laufzeit.

Bei der Vorstellung des wissensbasierten Typsystems wurde in Abschnitt 7.4.1 zwischen der syntaktischen und semantischen Typkonformität für Konzeptgraphen unterschieden. Während die syntaktische Typkonformität anhand der Struktur von Konzeptgraphen entschieden wird, berücksichtigt die semantische Typkonformität Informationen einer Ontologie. Die Ontologie enthält allgemeingültige Aussagen, die bei einem Vergleich zweier Konzeptgraphen herangezogen werden können. Die in Abschnitt 7.3 definierte Ontologie basierte ausschließlich auf der Hyponym–Beziehung. Arbeiten aus dem Bereich der Linguistik zeigen, daß daneben andere Beziehungen sinnvoll sein können, wie beispielsweise die Synonym– und Antonym–Beziehung (siehe [71]).

In Anlehnung an diese Arbeiten enthält der wissensbasierte Dienstvermittler eine weitere Komponente, die lexikographische Datenbank. Diese Komponente stellt eine Verallgemeinerung der in Abschnitt 7.3 eingeführten Ontologie dar. Die in der lexikographischen Datenbank enthaltenen Informationen können zu einer Menge von Tripeln verallgemeinert werden, die zwei lexikographische Einheiten mit einer Relation verknüpft. Bei den Konzeptgraphen entspricht eine lexikographische Einheit einem Konzept–, Relations– oder Instanznamen. Die lexikographische Datenbank verwaltet frei definierbare Beziehungen zwischen lexikographischen Einheiten. Die Diensttyphierarchie, die eine manuelle Navigation des Dienstangebots ermöglicht, ist ebenfalls ein Teil der lexikographischen Datenbank.

Die Nachrichten, die der wissensbasierte Dienstvermittler an seiner operationalen

Schnittstelle erhält, werden von einem *Dispatcher* an eine der drei Komponenten delegiert. Jede der drei Komponenten, die Dienstdatenbank, die Vergleichsregeln und die lexikographische Datenbank, können durch administrative Operationen verwaltet werden. Import– und Export–Operationen basieren auf den Vergleichsregeln. Während der Ausführung dieser Operationen wird auf Informationen aus der Dienstdatenbank und der lexikographischen Datenbank Bezug genommen.

8.4 Phasen der Diensttypvermittlung

Im folgenden werden die dynamischen Aspekte der wissensbasierten Dienstvermittlung in Form eines Vermittlungsprotokolls diskutiert. In Abschnitt 8.4.1 werden zunächst Anforderungen an ein Vermittlungsprotokoll formuliert. Die Vermittlung von Typen wird getrennt für den Export und den Import in den Abschnitten 8.4.2 und 8.4.3 durch Protokollautomaten vorgestellt. Eine formale Spezifikation dieser Automaten in der Spezifikationssprache SDL ist in [35] nachzulesen.

8.4.1 Anforderung an ein Vermittlungsprotokoll

Änderungen in einem wissensbasierten System können auf einen Lernvorgang zurückgeführt werden. *Lernen* soll als jede Veränderung eines Systems verstanden werden, die es diesem erlaubt, eine Aufgabe bei ihrer Wiederholung oder einer artverwandten Aufgabe besser zu lösen (siehe [97], [72]). Eine Übertragung des Lernvorgangs auf die wissensbasierte Dienstvermittlung führt zu einer Definition des Lernens von Diensttypen:

Definition 8.5 (Lernen von Diensttypen): Lernen ist die Konstruktion einer Approximation eines Diensttyps an Hand von Inferenzmechanismen, die diese aus den verschiedenen individuellen Sichten (d.h. Extensionen) erzeugen, so daß der Vermittler bei einer erneuten Anfrage den gesuchten Dienst identifizieren kann. □

In der künstlichen Intelligenz existieren verschiedene Modelle für Lernvorgänge, die stark von den verwendeten Lernmechanismen abhängen. Eine Übersicht der Verfahren findet sich in [69]. Für die Realisierung der Typvermittlung wird ein Lernverfahren verwendet, das auf der Verknüpfung von *Beispielen* und *Gegenbeispielen* beruht. Beispiele und Gegenbeispiele korrespondieren mit Sichten, die die Intension eines Diensttyps approximieren.

Bei einem *Lernvorgang* erlernt das System aus einer Menge von Beispielen und Gegenbeispielen eine generelle Beschreibung eines Konzepts (siehe Abbildung 8.3). Dabei hat das System vor dem Lernschritt zunächst keine Informationen über das Konzept. Die Beispiele werden inkrementell in mehreren Schritten präsentiert. Mit jeder Iteration des Lernvorgangs generiert das System eine Beschreibung, die mit jedem weiteren Beispiel verfeinert wird. Den Teilnehmern eines Lernvorgangs können unterschiedliche Rollen zugeordnet werden:

Lehrer: Ein Lehrer verfügt über spezielle Informationen auf dem Gebiet, in dem der Lernende sich Wissen aneignen soll. Der Lernvorgang wird vom Lehrer gesteuert, der gegebenenfalls Korrekturen vornimmt, wenn Fehler auftreten oder der Lernende

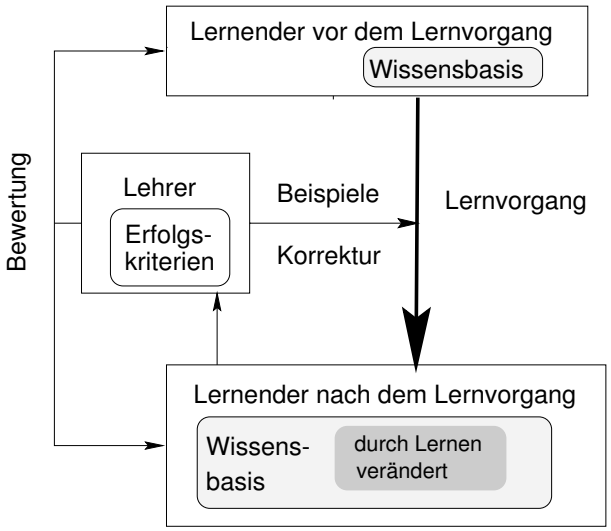


Abbildung 8.3: Ablauf eines Lernvorgangs.

Widersprüche innerhalb der vorgegebenen Beispiele erkennt. Außerdem stellt er die Erfolgskriterien auf, mit denen er feststellt, wann der Lernvorgang basierend auf den Beispielen abgeschlossen ist.

Lernender: Ein Lernender besitzt eine Wissensbasis, die durch den Lernvorgang modifiziert wird. Ziel des Lernvorgangs ist die Vermittlung von Informationen an den Lernenden, die dieser vor dem Lernschritt noch nicht besaß. Die Modifikationen an der Wissensbasis des Lernenden dürfen nicht unbeaufsichtigt geschehen, sondern müssen über eine Lernkontrolle bewertet werden.

Da die Menge der möglichen Beispiele, die der Lerner in seine Wissensbasis aufnehmen kann, *a priori* nicht beschränkt ist, kann nur ein vorläufiges Ende des Lernvorgangs festgestellt werden. Die Erfolgskriterien beinhalten ein Lernziel sowie Gütekriterien, die eine Bewertung des Lernergebnisses erlauben. Das Lernziel hängt davon ab, was gelernt werden soll. Bei der Konstruktion einer Approximation einer Typbeschreibung in Form eines Konzeptgraphen anhand von Beispielen, könnte das Lernziel beispielsweise fordern, daß sich der neue Konzeptgraph gegenüber anderen Konzeptgraphen genügend abgrenzt. Dies führt u.U. zu einer mehrfachen Interaktion zwischen Lehrer und Lernenden.

8.4.2 Protokoll für den Export von Typen

Das Protokoll für den Export von Typen teilt sich in mehrere Phasen auf, die durch die Anwendung des Modells des Lernvorgangs bestimmt werden. Der Export eines Diensttyps durchläuft vier verschiedene Phasen (siehe Abbildung 8.4): die *Lernphase*, *Abgrenzungsphase I*, *Abgrenzungsphase II* sowie die *Erweiterungsphase*. Jeder Export eines Diensttyps beginnt mit der *Lernphase*. Der Exporteur übermittelt dem Dienstvermittler eine initiale Beschreibung seines Diensttyps in Form eines Konzeptgraphen. Ist der Diensttyp noch nicht in der Diensttyphierarchie enthalten, muß dieser an einer geeigneten Stelle eingefügt

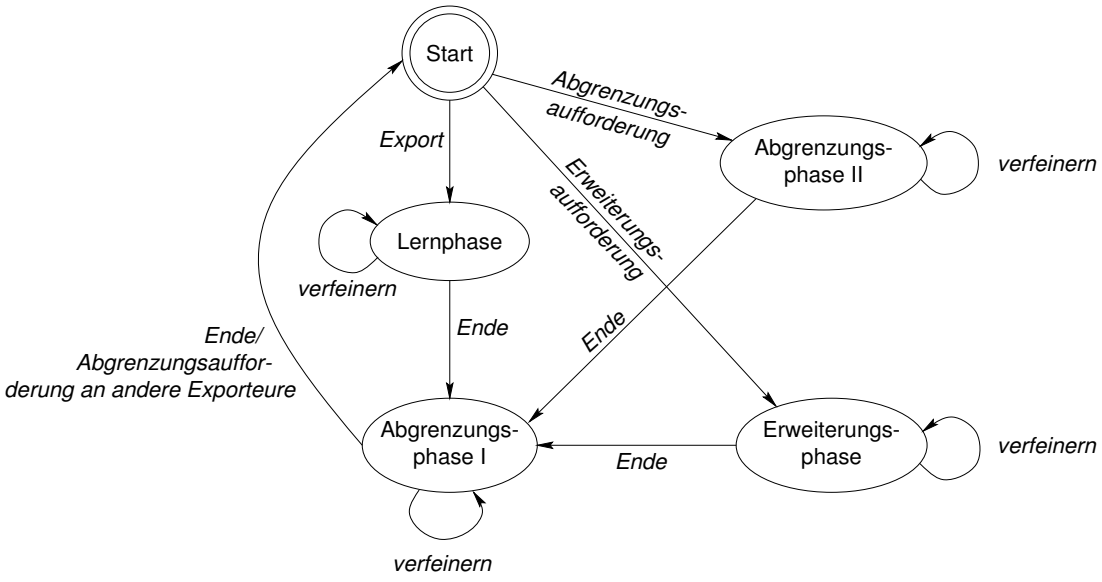


Abbildung 8.4: Export–Protokoll.

werden. Diese Aufgabe kann entweder zentral von einem Administrator oder von dem Exporteur selbst übernommen werden.

Im weiteren Verlauf der Lernphase präsentiert der Exporteur dem Vermittler sukzessive eine Menge von Typspezifikationen, die verschiedene Sichten des Dienstes repräsentieren und mit deren Hilfe der Kernbereich gebildet wird. In der Lernphase nimmt der Exporteur die Rolle des Lehrers ein und der Vermittler die Rolle des Lernenden. Beendet der Dienstanbieter die Lernphase, dann ermittelt der Vermittler in der Abgrenzungsphase I alle typkonformen Diensttypspezifikationen der ihm bekannten Diensttypen. Der Dienstanbieter muß dabei entscheiden, ob sein Diensttyp semantisch äquivalent mit einem der typkonformen Diensttypen ist. In diesem Fall wird seine Beschreibung in die des semantisch äquivalenten Diensttyps als weitere Extension aufgenommen. Ansonsten muß der Dienstanbieter seine Beschreibung dergestalt verfeinern, daß diese sich gegenüber den Diensttypen abgrenzt, die vom Vermittler aufgrund fehlender Informationen als typkonform erkannt wurden.

Diejenigen Anbieter, deren Diensttypspezifikationen fälschlicherweise als typkonform während der Abgrenzungsphase I eingestuft wurden, werden vom Vermittler benachrichtigt. Die Nachricht beinhaltet eine Abgrenzungsaufforderung. Die Empfänger dieser Nachricht wechseln in die Abgrenzungsphase II und verfeinern ihrerseits ihre Diensttypspezifikationen auf Basis des global veränderten Dienstangebots. Deren Verfeinerungen können rekursiv auf andere Exporteure Auswirkungen haben, so daß diese ebenfalls eine Abgrenzungsaufforderung des Vermittlers erhalten. In der Abgrenzungsphase II kehrt sich der Lernprozeß um, so daß der Vermittler als Lehrer den anderen Anbietern ein neues Beispiel präsentiert. Die abschließende Bewertung der Verfeinerung findet wieder in der Abgrenzungsphase I statt, bei dem sich die Rollenverteilung von Lehrer und Lernenden erneut umkehrt.

Ein Export kann auch von einem Import indirekt beeinflußt werden (siehe nächster

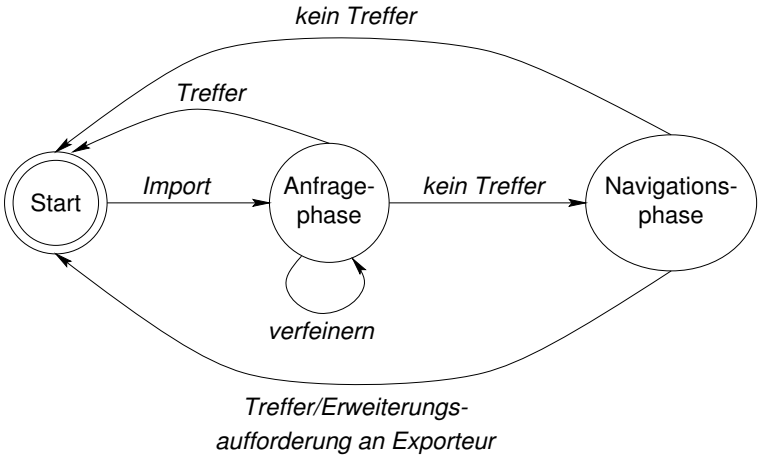


Abbildung 8.5: Import–Protokoll.

Abschnitt). Der Vermittler schickt in diesem Fall eine Erweiterungsaufforderung an den Exporteur, die eine Sicht eines Dienstnutzers enthält. Der Exporteur muß in der Erweiterungsphase überprüfen, ob die Sicht des Dienstnutzers einer gültigen Spezifikation seines Diensttyps entspricht um ggf. seine eigene Diensttypspezifikation zu verfeinern. Da ein Konzeptgraph die Intension eines Diensttyps nur approximiert, ist der Export eines Dienstes kein abgeschlossener Vorgang. Der Dienstanbieter muß immer damit rechnen, daß ein Dienstnutzer eine Sicht zur Beschreibung seines Dienstes gewählt hat, die nicht in seiner Typspezifikation enthalten ist. Der Exporteur ist hierdurch auch der Verwalter seiner Typspezifikation, da nur er entscheidet, welche Sichten aufgenommen werden.

8.4.3 Protokoll für den Import von Typen

Das Import–Protokoll durchläuft zwei Phasen (siehe Abbildung 8.5): die *Anfragephase* und die *Navigationsphase*. Der Dienstnutzer stellt in der Anfragephase eine Suchanfrage nach einem Diensttyp an den Vermittler. Dieser versucht, einen typkonformen Diensttyp zu ermitteln. Der Vermittler bedient sich dabei der Vergleichsregeln, die er für die Überprüfung der Typkonformität heranzieht. Als Ergebnis der Anfrage erhält der Importeur eine Liste von potentiellen Diensttypen, die seiner Anfrage genügen (d.h. typkonform zu der Typspezifikation der Anfrage sind). Der Importeur kann nun einen Diensttyp auswählen oder seine Anfrage verfeinern. Dazu kann er sich die Typspezifikationen der Ergebnisse übermitteln lassen und diese in seiner verfeinerten Anfrage verwenden.

Anfragen der Dienstnutzer können als ein Test des bisher durch den Vermittler gelernten Wissens angesehen werden. Ist eine Anfrage zu allgemein, dann werden viele Typspezifikationen vom Vermittler als typkonform identifiziert. Eine Reformulierung der Anfrage ist dann notwendig. Durch die dynamische Entwicklung des Systems werden die in der Dienstdatenbank gespeicherten Typspezifikationen weiterentwickelt (beispielsweise in der Abgrenzungsphase II des Export–Protokolls). Eine Anfrage, die zu einem früheren Zeitpunkt gute Ergebnisse geliefert hat, kann zu einem späteren Zeitpunkt wegen dieser Weiterentwicklung zu allgemein sein, so daß auch auf der Seite des Importeurs ein

Lernschritt notwendig wird. Er muß dann seine Anfrage verfeinern. Die Qualität einer Typspezifikation hängt hierdurch unmittelbar von dem Umfang der Angebote ab.

Liefert die Anfragephase nicht den gewünschten Diensttyp, so wechselt der Importeur in die Navigationsphase. In dieser Phase durchsucht der Importeur über eine manuelle Navigation alle exportierten Diensttypen. Anhand der Diensttyphierarchie kann der Importeur die Menge der Typspezifikationen der Dienstdatenbank systematisch erschließen. Dabei kann er sich die entsprechenden Typspezifikationen anzeigen lassen, um zu entscheiden, ob eine von diesen seinen Vorstellungen entspricht. Hat er die gewünschte Typspezifikation gefunden, dann kann er den Vermittler veranlassen, seine ursprüngliche Typspezifikation, die während der Anfragephase kein Ergebnis erzielt hat, dem Exporteur des Diensttyps zu übermitteln. Hier tritt der Vermittler in der Rolle des Lehrers gegenüber dem Exporteur auf. Der Exporteur wechselt in die oben beschriebene Erweiterungsphase.

8.5 Anwendung bei den natürlichen Sprachen

In diesem und dem folgenden Abschnitt werden zwei Anwendungen der wissensbasierten Dienstvermittlung vorgestellt. Beiden ist gemein, daß die Konzeptgraphen als *Meta–Grammatik* verwendet werden, die eine Anpassung an unterschiedliche Notationen erlaubt. Die Konzeptgraphen als Wissensrepräsentationstechnik ermöglichen die Kodierung beliebiger Aussagen. Je nach Anwendung existieren spezialisierte Notationen, um die Aussagen darzustellen. Zwei Beispiele für spezialisierte Notationen sind die *natürliche Sprache* und die im nächsten Abschnitt behandelten *Schnittstellenbeschreibungssprachen*. Eine Sicht eines Anwenders umfaßt hierdurch nicht nur eine Art, einen Dienst zu beschreiben, sondern auch die Notation, in der die Darstellung vorgenommen wird. Der wissensbasierte Dienstvermittler arbeitet ausschließlich auf Basis der Konzeptgraphen. Vorgeschaltete *Übersetzer* erlauben Anwendern die Formulierung von Typspezifikationen in ihrer bevorzugten Sprache.

Ein Übersetzer transformiert Worte einer Sprache in Worte einer anderen Sprache (siehe Abbildung 8.6). Mehrere Übersetzer können kaskadiert angeordnet werden, so daß sich neue Übersetzer ergeben. Da die wissensbasierte Dienstvermittlung auf den Konzeptgraphen basiert, sind insbesondere die Übersetzung, die als Ausgangs– oder Zielsprache die Konzeptgraphen haben, von Interesse. Es wird sich zeigen, daß die Übersetzung zwischen Schnittstellenbeschreibungssprachen und Konzeptgraphen automatisierbar ist, während für die natürliche Sprache nachwievor keine allgemeingültigen Lösungen existieren. Eine geeignete Anordnung der Übersetzer würde theoretisch eine Übersetzung zwischen natürlicher Sprache und Schnittstellenbeschreibungssprachen erlauben. Zumindest eine Richtung, die Übersetzung von Schnittstellenbeschreibungssprachen in die natürliche Sprache erscheint sinnvoll, um aus operationalen Schnittstellenspezifikationen automatisch Dokumentationen zu erzeugen.

Die natürliche Sprache ist für Anwender geeignet, um ihre Vorstellungen über einen Diensttyp zu formulieren. Die natürliche Sprache zeichnet sich durch einen hohen Grad an Zweideutigkeiten aus, die eine automatische Übersetzung erschweren. Die dabei auftretenden Probleme sind noch nicht zufriedenstellend gelöst. Für Teilbereiche existieren Ansätze, die die Übersetzung in einem eingeschränkten Kontext ermöglichen. In [63]

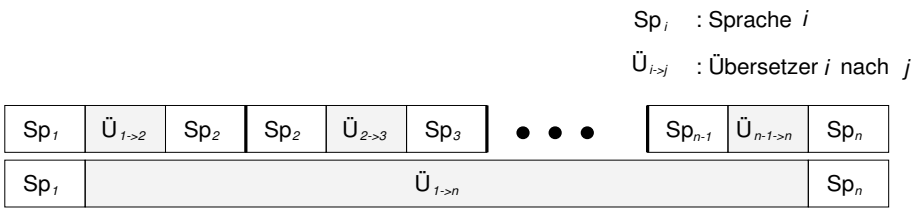


Abbildung 8.6: Übersetzung zwischen Sprachen.

wird beispielsweise ein automatischer Übersetzer vorgestellt, der Wegbeschreibungen in der englischen Sprache in Konzeptgraphen transformiert.

Die Übersetzung von Konzeptgraphen in die natürliche Sprache ist einfacher, da ein Konzeptgraph Syntax und Semantik einer Spezifikation in sich vereint. In Abschnitt 7.2.2 wurde bereits eine intuitive Umsetzung eines Konzeptgraphen der Form `[`K_1`] -> (`R`) -> [`K_2`]` mit entsprechenden Füllwörtern in einen natürlichsprachlichen Satz nach der Schablone *„Die R eines* K_1 *ist ein* K_2*“* vorgestellt. Es existieren einige Arbeiten, die diese einfache Übersetzung erweitern, so daß die Schablone grammatikalisch korrekt umgesetzt wird (siehe beispielsweise [75]). Unterschiedliche Übersetzer erlauben die Konvertierung eines Konzeptgraphen in unterschiedliche natürliche Sprachen. In [110] werden beispielsweise medizinische Diagnosen, die in Form von Konzeptgraphen vorliegen, in die Sprachen Deutsch, Englisch und Französisch übersetzt.

Wegen der Komplexität bei der Übersetzung zwischen natürlicher Sprache und Konzeptgraphen wird im folgenden auf diesen Komplex nicht weiter eingegangen. Da Konzeptgraphen eine intuitive Abstraktion der natürlichen Sprache darstellen, wird davon ausgegangen, daß die Notation der Konzeptgraphen für Anwender geeignet ist. Der Umgang mit Konzeptgraphen kann durch graphisch–orientierte, interaktive Werkzeuge vereinfacht werden. Die Implementationsbeschreibung im nächsten Kapitel wird ein entsprechendes Werkzeug vorstellen.

Neben der maschinellen Übersetzung sind die Vergleichsregeln von Bedeutung, die die Typkonformität zwischen Konzeptgraphen überprüfen. Im letzten Kapitel wurde bereits gezeigt, daß die dort vorgestellte semantische Typkonformität nicht ausreichend ist. Der Vergleich von Konzeptgraphen ist abhängig von dem in den Konzeptgraphen kodierten Spezifikationen. Im folgenden werden drei Vergleichsregeln erläutert, die Konzeptgraphen mit natürlichsprachlichem Inhalt vergleichen. Der Fokus liegt dabei auf einer Allgemeingültigkeit, die die Vergleichsregeln nicht nur für genau eine Spezifikation anwendbar machen. Auf eine formale Spezifikation der Vergleichsregeln wird an dieser Stelle verzichtet. Anhang A enthält eine formale Spezifikation der drei folgenden Vergleichsregeln auf Basis der Programmiersprache Prolog.

8.5.1 Teilweise übereinstimmende Sichten

Die Definition der semantischen Typkonformität aus Abschnitt 7.4.1 garantiert, daß sämtliche Alternativen eines Anfragegraphen eine Verallgemeinerung von Alternativen eines Typgraphen sind. Die semantische Typkonformität schlägt fehl, wenn der Anfragegraph Alternativen besitzt, die nicht im Typgraphen vorkommen. Unterschiedliche Alternativen

entsprechen unterschiedlichen Extensionen einer Intension eines Diensttyps. Es erscheint sinnvoll, die Typkonformität für diesen Fall zu erweitern.

Beispiel 8.1: Gegeben sei folgender Konzeptgraph, der von einem Dienstanbieter exportiert wurde:

```
[OO-LANGUAGE : {C++}] -
  -> (SUPERSET-OF) -> [PROGRAMMING-LANGUAGE : {C}],
  -> (DOES-NOT-SUPPORT) -> [CLASS-OBJECTS] .
```

Die informelle Semantik des Konzeptgraphen ist: *„C++ ist eine objektorientierte Programmiersprache, die eine Obermenge der Programmiersprache C ist und keine Klassenobjekte unterstützt.“* Ferner repräsentiere der folgende Konzeptgraph eine Anfrage eines Dienstnutzers:

```
[OO-LANGUAGE] -
  -> (SUPERSET-OF) -> [PROGRAMMING-LANGUAGE : {C}],
  -> (SUPPORTS) -> [CLASSES] .
```

Die informelle Semantik der Anfrage lautet: *„Gesucht ist eine objektorientierte Programmiersprache, die eine Obermenge der Programmiersprache C ist und Klassen unterstützt.“* Obwohl eine Alternative des Anfragegraphen mit dem Typgraphen übereinstimmt, würde die semantische Typkonformität aus Abschnitt 7.4.1 keine Typkonformität feststellen.

Eine Vergleichsregel, die teilweise übereinstimmende Sichten berücksichtigt, erlaubt, daß ein Anfragegraph Extensionen besitzt, die im Typgraphen nicht vorhanden sind. Voraussetzung ist, daß mindestens eine Alternative des Anfragegraphen mit einer des Typgraphen übereinstimmt. Die Übereinstimmung der überlappenden Alternativen verläuft analog wie bei der semantischen Typkonformität unter Bezugnahme auf eine Ontologie. Die formale Spezifikation einer Vergleichsregel für teilweise übereinstimmende Sichten ist unter dem Namen `Specialization` in Anhang A widergegeben.

8.5.2 Quantität–Vergleichsregel

Die semantische Typkonformität und die Vergleichsregel für teilweise überlappende Sichten versuchen die Alternativen des Anfragegraphen über eine Ontologie in Beziehung zu Alternativen eines Typgraphen zu stellen. Die Übereinstimmung kann unabhängig von dem Inhalt, d.h. der konkreten Spezifikation die der Konzeptgraph repräsentiert, festgestellt werden. Es existieren Spezifikationen, die über die bisher vorgestellten Vergleichsregeln nicht als typkonform erkannt werden, die jedoch intuitiv als konform eingestuft würden.

Beispiel 8.2: Der Typgraph `[PRINTER:{HP-LASERJET}] -> (RESOLUTION) -> [DPI:600]` besitzt die informelle Semantik: *„Ein HP–Laserjet ist ein Drucker mit einer Auflösung von 600 DPI.“* Der Konzeptgraph `[PRINTER] -> (GREATER_OR_EQUAL) -> [DPI:300]` besitzt als Anfragegraph die informelle Semantik: *„Gesucht wird ein Drucker, der mindestens 300 DPI besitzt.“* Diese Anfrage könnte dem vorherigen Dienstangebot zugeordnet werden, da ein HP–Laserjet die Mindestanforderung von 300 DPI erfüllt.

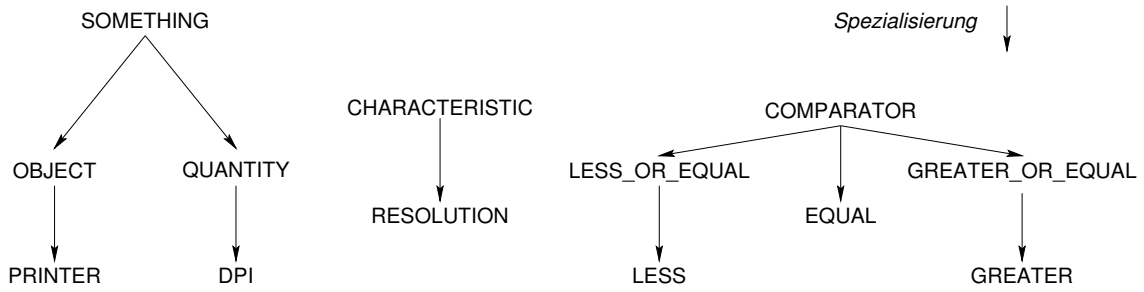


Abbildung 8.7: Ontologie für Quantitäts–Vergleichsregel.

Die in dem letzten Beispiel vorgestellte Zuordnung von Angebot und Nachfrage, die auf einem quantitativen Vergleich von Instanzen zweier Konzepte beruht, kann über eine weitere Vergleichsregel realisiert werden. Das allgemeine Schema der Quantitäts–Vergleichsregel ist durch die beiden folgenden Konzeptgraphen gegeben:

Typgraph: `[OBJECT] -> (CHARACTERISTIC) -> [QUANTITY:`i_1`]`

Anfragegraph: `[OBJECT] -> (COMPARATOR) -> [QUANTITY:`i_2`]`

Sind Anfrage– und Typgraph Spezialisierungen dieser allgemeinen Konzeptgraphen, kann die Quantitäts–Vergleichsregel angewandt werden. Die Konzeptgraphen aus Beispiel 8.2 sind bzgl. der in Abbildung 8.7 dargestellten Ontologie Spezialisierungen der allgemeinen Konzeptgraphen, so daß hier die Quantitäts–Vergleichsregel anwendbar ist. Die numerischen Werte i_1 und i_2 entscheiden in Abhängigkeit von dem Vergleichsoperator `(COMPARATOR)` über die Typkonformität. Die formale Spezifikation der Vergleichsregel in der Sprache Prolog ist dem Anhang A unter der Bezeichnung `Quantity` zu entnehmen.

8.5.3 Negation–Vergleichsregel

Die bisher vorgestellten Vergleichsregeln liberalisieren die Definition der Typkonformität im Vergleich zu der semantischen Typkonformität. D.h., die Menge der typkonformen Diensttypen ist unter Anwendung der in den letzten beiden Abschnitten vorgestellten Vergleichsregel im Mittel größer, als bei alleiniger Anwendung der semantischen Typkonformität. Liefert eine Vergleichsregel das Ergebnis `reject` während des Vergleichs zweier Konzeptgraphen, so schränkt diese die Menge der typkonformen Diensttyp ein. Dies kann beispielsweise von einer speziellen Vergleichsregel geleistet werden, die zwei Konzeptgraphen auf sich widersprechende Sichten hin untersucht. Das folgende Beispiel zeigt ein mögliches Szenario:

Beispiel 8.3: Gegeben sein der in Beispiel 8.1 angegebene Typgraph, sowie der folgende Anfragegraph:

```
[OO-LANGUAGE] -
  -> (SUPERSET-OF) -> [PROGRAMMING-LANGUAGE : {C}],
  -> (SUPPORTS) -> [CLASS-OBJECTS] .
```

Die informelle Semantik des Anfragegraphen lautet: *„Gesucht wird eine objektorientierte Programmiersprache, die eine Obermenge der Programmiersrpache C ist und Klassenobjekte unterstützt.“* Die Vergleichsregel für teilweise übereinstimmende Sichten würde den Typgraphen als typkonform zu dem Anfragegraphen erkennen, da diese eine gemeinsame Alternative besitzen. Die Alternative `[OO-LANGUAGE] -> (SUPPORTS) -> [CLASS-OBJECTS]` des Anfragegraphen widerspricht jedoch der Alternative `[OO-LANGUAGE:{C++}] -> (DOES-NOT-SUPPORT) -> [CLASS-OBJECTS]` des Typgraphen.

Das Beispiel zeigt, daß trotz übereinstimmender Sichten zwischen Anfrage– und Typgraph die Typkonformität explizit verworfen werden sollte. Dieser Sachverhalt kann durch eine weitere Vergleichsregel geleistet werden. Die Anwendbarkeit der Vergleichsregel für Negation ist gegeben, wenn zwei Alternativen des Typ– und Anfragegraphen nach folgendem Schema aufgebaut sind:

Typgraph: `[`K_1`] -> (`R`) -`
`-> [`K_2`],`
$\ldots$
`-> [`K_n`].`

Anfragegraph: `[`K'_1`] -> (`R'`) -`
`-> [`K'_2`],`
$\ldots$
`-> [`K'_n`].`

Die Vergleichsregel für Negation ist genau dann anwendbar, wenn für die Konzepte K_i und K'_i mit $i = 1, \ldots, n$ gilt $\Gamma \cup \{\Phi(\texttt{[}K_i\texttt{]})\} \models \Phi(\texttt{[}K'_i\texttt{]})$ und die Semantik von R und R' sich widersprechen. Als Grundlage für eine widersprüchliche Semantik bei Relationen dient die aus der Linguistik bekannte Antonym–Beziehung (siehe [71]). Die Anwendung der Negations–Vergleichsregel bedingt hierdurch eine neue Beziehungsart in der lexikographischen Datenbank. Auf Basis der dort gespeicherten Informationen kann die Vergleichsregel festellen, ob R und R' in einer Antonym–Beziehung stehen. Die formale Spezifikation dieser Vergleichsregel findet sich in Anhang A unter der Bezeichnung `Negation`.

8.6 Anwendung bei den Schnittstellenbeschreibungssprachen

Syntaktische Typbeschreibungssprachen erlauben die Spezifikation operationaler Objektschnittstellen. Im Vergleich zu den im letzten Abschnitt behandelten natürlichen Sprachen besitzen sie eine eingeschränkte Ausdruckskraft und sind für die Spezifikation von Schnittstellen spezialisiert. Die bereits in Abschnitt 3.4.2 diskutierte Unterscheidung zwischen Systemtypen und Benutzertypen zeigte, daß Schnittstellenbeschreibungssprachen sich nicht für die Spezifikation von anwendernahen Diensttypen eignen. Obwohl die Konzeptgraphen bisher als Typbeschreibungssprache für Anwender vorgestellt wurden, können diese darüber hinaus für die Spezifikation operationaler Schnittstellen herangezogen werden.

Die Syntax der Konzeptgraphen gleicht einer Meta–Grammatik, mit der bei geeigneter Wahl von Konzept– und Relationsnamen beliebige Aussagen spezifiziert werden können. Diese Interpretation ist auf jede beliebige Anwendungsdomäne übertragbar. Neben Spezifikationen in Form von natürlichsprachlichen Aussagen können beispielsweise auch Schnittstellenspezifikationen mit Hilfe von Konzeptgraphen dargestellt werden. Da ein Konzeptgraph meherere Sichten auf einen Diensttyp in sich vereinigt, kann dieser mehrere Extensionen eines Diensttyps auf unterschiedlichen Abstraktionsebenen enthalten. Die beiden in Abbildung 7.5 auf Seite 108 dargestellten Sichten könnten beispielsweise einer Spezifikation einer operationalen Schnittstelle und einer informellen natürlichsprachlichen Beschreibung der Funktionalität des Diensttyps entsprechen.

Ein Dienstanbieter erhält hierdurch die Möglichkeit, seinen Dienst auf unterschiedlichen Abstraktionsebenen zu spezifizieren. Während der Typvermittlung ist die Sicht relevant, die die natürlichsprachliche Spezifikation des Diensttyps enthält. Ist ein typkonformer Diensttyp gefunden, enthält dieser zusätzlich die Spezifikation der operationalen Schnittstelle in einer weiteren Extension. Aus Sicht des wissensbasierten Dienstvermittlers ist die Unterscheidung in den Abstraktionsebenen der Extensionen transparent, da dieser ausschließlich Konzeptgraphen verarbeitet.

Die folgenden Abschnitte beschreiben die Anwendung der wissensbasierten Dienstvermittlung bei den Schnittstellenbeschreibungssprachen von DCE und CORBA. Ebenso wie bei den natürlichen Sprachen, sind hier zwei Aspekte von Interesse. Einerseits muß zwischen den beiden Schnittstellenbeschreibungssprachen und den Konzeptgraphen ein Übersetzer konstruiert werden. Andererseits sind für die unterschiedlichen Definitionen der syntaktischen Typkonformität von DCE, CORBA und ODP entsprechende Vergleichsregeln anzugeben. Abschnitt 8.6.1 zeigt ein Szenario, welches einen Vermittlungsvorgang zwischen unterschiedlichen Typdomänen beschreibt. Daraus leiten sich Anforderungen an die Übersetzer zwischen Schnittstellenbeschreibungssprachen und Konzeptgraphen ab. Da die Übersetzer bereits in [48] ausführlich behandelt wurden, wird im folgenden auf die notwendigen Vergleichsregeln eingegangen. Dazu wird in Abschnitt 8.6.2 zunächst eine abstrakte Schnittstellenbeschreibungssprache vorgestellt, mit der dann in Abschnitt 8.6.3 die Vergleichsregeln formal spezifiziert werden können.

8.6.1 Wissensbasierte Vermittlung zwischen unterschiedlichen Typdomänen

Auf Basis der wissensbasierten Dienstvermittlung kann ein Vermittlungsvorgang über unterschiedliche Typdomänen ausgedehnt werden. Schnittstellenspezifikationen in unterschiedlichen Typbeschreibungssprachen werden in Konzeptgraphen übersetzt und an den wissensbasierten Dienstvermittler weitergeleitet. Im folgenden wird auf die Vermittlung zwischen der DCE– und der CORBA–Typdomäne eingegangen. Der wissensbasierte Vermittler befindet sich in der Konzeptgraphen–Typdomäne und die Dienstnutzer und –anbieter in beliebigen anderen Typdomänen. Abbildung 8.8 zeigt den Fall, daß ein Dienstnutzer sich in einer DCE–Typdomäne befindet, während der Dienstanbieter in einer CORBA–Typdomäne angesiedelt ist.

Dienstnutzer N und –anbieter A besitzen jeweils eine Referenz auf den wissensbasierten Vermittler V. Da Anfangs– und Endpunkte der Referenzen in unterschiedlichen

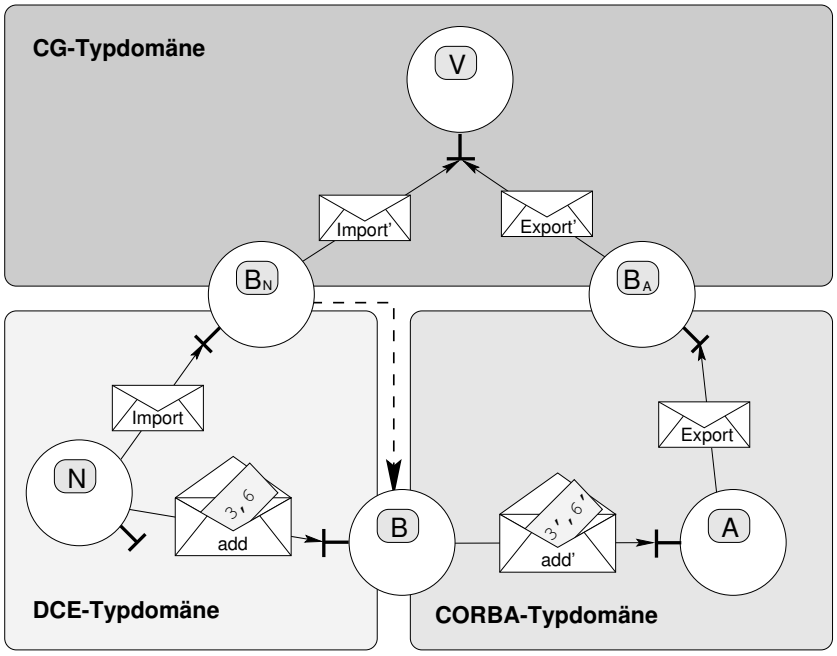


Abbildung 8.8: Dienstvermittlung zwischen DCE– und CORBA–Typdomänen.

Typdomänen liegen, müssen Binder die Domänen überbrücken (Objekte B_N und B_A in Abbildung 8.8). Die in den Import– und Export–Operationen enthaltenen Typspezifikationen basieren auf den Typbeschreibungssprachen der jeweiligen Typdomäne. Die Binder B_N und B_A übersetzen diese in Konzeptgraphen. Beantwortet der Vermittler V die Import–Operation des Dienstnutzers N mit einer Referenz auf A, so muß Binder B_N einen neuen Binder B instantiieren (gestrichelter Pfeil in Abbildung 8.8). In Abhängigkeit der Möglichkeiten des Binders B bei der Anpassung der über ihn laufenden Nachrichten, kann der Binder B_N beim Import eine geeignete Vergleichsregel auswählen. Ist Binder B beispielsweise in der Lage, Parametertypen gemäß der Kontra– und Kovarianzregeln zu konvertieren, kann für den Import die ODP Vergleichsregel für syntaktische Typkonformität herangezogen werden.

Die oben beschriebene Anordnung der drei Typdomänen zeigt, daß zumindest Übersetzer von den Schnittstellenbeschreibungssprachen CORBA's und DCE's nach Konzeptgraphen existieren müssen. Bedient sich ein Programmierer des wissensbasierten Dienstvermittlers, um zuerst einen geeigneten Typ zu ermitteln, so muß darüber hinaus ein Übersetzer existieren, der einen Konzeptgraphen auf die vom Programmierer gewünschte Typbeschreibungssprache übersetzt. Die Konzeptgraphen, die aus einer Übersetzung einer operationalen Schnittstellenspezifikationen hervorgegangen sind, bestehen aus vorgegebenen Konzept– und Relationsnamen. Die einzelnen Bestandteile einer Schnittstellenspezifikation werden auf Teilgraphen eines Konzeptgraphen abgebildet:

Schnittstellenspezifikation:

`interface` *name*
{
    Konstanten–Deklarationen
    Typ–Deklarationen
    Operationen
}

Konzeptgraph:

`[INTERFACE:{„`*name*`“}] -`
`-> (CONSTANTS) ->` ···
`-> (TYPES) ->` ···
`-> (OPERATIONS) ->` ···
`-> (ANNOTATIONS) ->` ···
.

Der Aufbau der Konzeptgraphen–Schablone orientiert sich an den Anforderungen, die bei der Vermittlung von operationalen Schnittstellenspezifikationen gestellt werden. Bei der Diskussion verschiedener syntaktischer Typsysteme in Kapitel 4 zeigte sich, daß einige Eigenschaften einer Typbeschreibungssprache für die Vermittlung operationaler Schnittstellenspezifikationen nicht relevant sind (wie beispielsweise die Vererbung der CORBA–IDL). Unabhängig davon sind diese Eigenschaften für eine Übersetzung zwischen verschiedenen Schnittstellenbeschreibungssprachen notwendig. Die Schablone definiert deswegen einen Teilgraphen, der durch `ANNOTATIONS` eingeleitet wird und alle Spezifika einer operationalen Schnittstellenspezifikation enthält, die für den Vermittlungsvorgang nicht notwendig sind. Dort werden die Informationen abgelegt, die für eine Rückübersetzung in eine Schnittstellenbeschreibungssprache benötigt werden.

Das Anfügen von Annotationen ermöglicht eine präzise Übersetzung zwischen einer Schnittstellenbeschreibungssprache und den Konzeptgraphen in beide Richtungen. Probleme bereiten jedoch Übersetzungen zwischen unterschiedlichen Schnittstellenbeschreibungssprachen (siehe [108], [109]). Einige Eigenschaften der jeweiligen Schnittstellenbeschreibungssprache können nicht, oder nur indirekt, in eine andere Notation überführt werden. Die Eigenschaften, die die Schnittstellenbeschreibungssprachen von DCE und CORBA bei der Spezifikation operationaler Schnittstellen erlauben, können als *direkt*, *indirekt* oder *nicht übersetzbar* klassifiziert werden:

Direkt Übersetzbar: Bestimmte Eigenschaften einer Schnittstellenbeschreibungssprache können direkt übersetzt werden. Dazu gehören beispielsweise Basistypen, `struct`– und `union`–Konstrukte sowie die Spezifikation von Operationen.

Indirekt Übersetzbar: Einige Eigenschaften können nur indirekt in der jeweils anderen Typbeschreibungssprache dargestellt werden. Dabei kann bei der Übersetzung nur auf Sprachelemente der Zielsprache zurückgegriffen werden, um die fehlenden Eigenschaften nachzubilden. Dazu gehören beispielsweise die Schnittstellenvererbung in CORBA und die Zeigerstrukturen von DCE.

Nicht Übersetzbar: Für einen Teil der Eigenschaften einer Schnittstellenbeschreibungssprache existiert in der Zielsprache weder ein Äquivalent, noch lassen sich diese indirekt übersetzen. Dazu gehören beispielsweise Klienten–Kontexte in CORBA und das Pipe–Konstrukt in DCE.

Direkt übersetzbare Eigenschaften einer Schnittstellenbeschreibungssprache lassen sich bijektiv in beide Richtungen übersetzen. Bei der indirekten Übersetzung gehen hingegen

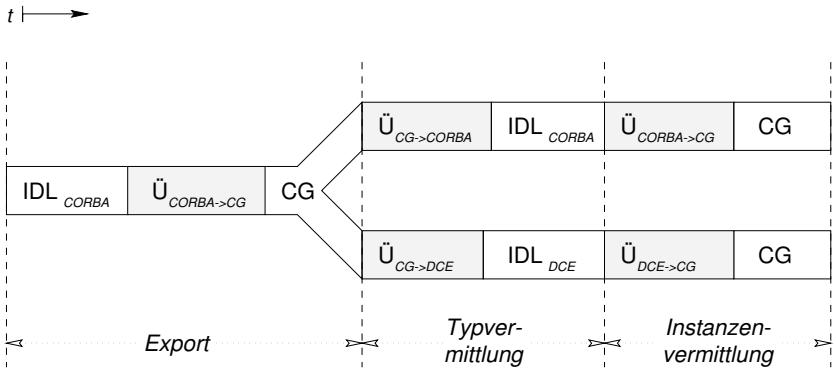


Abbildung 8.9: Mehrfachübersetzungen zwischen Schnittstellenspezifikationen.

Informationen verloren, die eine symmetrische Abbildung unmöglich machen. Die Vererbung von Schnittstellenspezifikationen in CORBA kann beispielsweise über eine Abflachung der Vererbungshierarchie nach DCE umgewandelt werden. Bei einer Rückübersetzung geht die Information über die Vererbungsbeziehung jedoch verloren. Enthält eine Schnittstellenspezifikation nicht übersetzbare Konstrukte, so ist diese von einer Übersetzung in eine andere Schnittstellenspezifikation ausgeschlossen.

Die Typ– und Instanzenvermittlung zwischen unterschiedlichen Typdomänen resultiert in einer Folge von Übersetzungen der betreffenden Schnittstellenspezifikationen. Zunächst exportiert ein Dienstanbieter seine operationale Schnittstellenspezifikation zusammen mit deren informeller Beschreibung in Form eines Konzeptgraphen. Nach einer Übersetzung vermerkt der Vermittler die Spezifikation der operationalen Schnittstelle und der informellen Beschreibung in seiner Dienstdatenbank. Während der Typvermittlung sucht ein Programmierer in der Rolle eines Importeurs zunächst einen geeigneten Diensttyp anhand einer informellen natürlichsprachlichen Beschreibung. Ist einer gefunden, extrahiert er die Spezifikation der operationalen Schnittstelle und übersetzt diese in die von ihm eingesetzte Schnittstellenbeschreibungssprache. Während der Instanzenvermittlung wird die extrahierte Schnittstellenspezifikation wiederum in einen Konzeptgraphen übersetzt. Abbildung 8.9 stellt diesen Vorgang dar.

Ein vollständiger Vermittlungsvorgang (d.h. Typ– und Instanzenvermittlung) schließt drei Übersetzungen zwischen Schnittstellenspezifikationen und Konzeptgraphen ein. Die Übersetzer, die dies leisten, müssen garantieren, daß die Konzeptgraphen nach der ersten und dritten Übersetzung trotz eines evtl. Informationsverlustes von dem Dienstvermittler bzgl. der Vergleichsregeln als konform erkannt werden. Diese Anforderung kann für direkt und indirekt übersetzbare Eigenschaften von Schnittstellenbeschreibungssprachen garantiert werden. Dazu sind Konventionen festzulegen, wie indirekt übersetzbare Eigenschaften von allen Übersetzern gleich verarbeitet werden.

Die Übersetzung zwischen den Schnittstellenbeschreibungssprachen von DCE bzw. CORBA und den Konzeptgraphen ist in [48] ausführlich diskutiert. Der große Detaillierungsgrad einer Schnittstellenspezifikation resultiert in umfangreichen Konzeptgraphen mit einer Vielzahl von unterschiedlichen Konzeptnamen. Zentrales Ergebnis der oben zitierten Arbeit ist ein Konzeptkatalog, der parallel zu dem Konzeptkatalog für natürliche

Sprachen in [98] die Spezifikation von Schnittstellenspezifikationen in Form von Konzeptgraphen ermöglicht. Alle notwendigen Konzeptnamen für die Spezifikation operationaler Schnittstellen sind in Anhang C tabellarisch mit ihrer Semantik aufgeführt.

8.6.2 Abstrakte Schnittstellenbeschreibungssprache

Da die Übersetzung zwischen den Schnittstellenbeschreibungssprachen und den Konzeptgraphen an anderer Stelle ausführlich diskutiert wird, soll im folgenden auf die Vergleichsregeln eingegangen werden. Die Übersetzung von Schnittstellenspezifikationen resultiert in komplexen Konzeptgraphen. Die unterschiedlichen Konzeptnamen aus Anhang C reflektieren die unterschiedlichen Eigenschaften der CORBA– und DCE–IDL. Eine formale Spezifikation der Vergleichsregeln auf Basis der Konzeptgraphen wäre hierdurch unübersichtlich. Aus diesem Grund wird in diesem Abschnitt eine *abstrakte Schnittstellenbeschreibungssprache* eingeführt.

Die abstrakte Schnittstellenbeschreibungssprache entspricht einer spezialisierten Notation, mit der sich unterschiedliche Definitionen von Typkonformitäten formal spezifizieren lassen. Mit ihr lassen sich folglich nur diejenigen Eigenschaften einer Schnittstellenspezifikation formulieren, die für die unterschiedlichen Typkonformitäten relevant sind. Sie sind nicht für eine Übersetzung zwischen Schnittstellenbeschreibungssprachen geeignet. Die hier definierte abstrakte Schnittstellenbeschreibungssprache besitzt eine ausreichende Ausdruckskraft, um die in Kapitel 4 vorgestellten Typkonformitäten der syntaktischen Typsysteme überprüfen zu können. Die abstrakte Notation wird durch die Grammatik G_{ASBS} wie folgt definiert:

Definition 8.6 (Grammatik der abstrakten Schnittstellenbeschreibungssprache): Die Syntax der abstrakten Schnittstellenbeschreibungssprache ist definiert über eine Grammatik $G_{ASBS} = (N, \Sigma, P, S)$, mit $N = \{$ `asbs`, `operation`, `id`, `sig`, `args`, `type`, `type_list` $\}$, Σ dem ISO8859–1 Zeichensatz und $S =$ `asbs`. Sei $\Upsilon =_{df} \Sigma \setminus = \{$ "μ", "$\rightarrow$", "$\times$", "[", "]", "<", ">", ",", ".", ":", ";" $\}$. Die Menge der Produktionen P ist gegeben durch:

```
asbs      ← operation
asbs      ← asbs ";" operation
operation ← id ":" sig
sig       ← args "→" args
args      ← type
args      ← args "×" type
type      ← id
type      ← "void"
type      ← type "→" type
type      ← type "×" type
type      ← "⟨" type_list "⟩"
type      ← "[" type_list "]"
type      ← "μ" id "." type
type_list ← id ":" type
```

```
type_list ← type_list "," id ":" type
id        ← ϒ+
```

□

Die Definition 8.6 ist an die der abstrakten Notation für Schnittstellentypen aus dem RM–ODP X.903 angelehnt (siehe [44] und [74]). Eine abstrakte Schnittstellenspezifikation besteht aus einer Menge von Operationen (`operation`), die einen Namen und eine Signatur (`sig`) besitzen. Eine Signatur definiert eine Abbildung, die eine Menge von Eingabeparametertypen auf eine Menge von Ausgabeparametertypen (beide durch das nicht–terminale Symbol `args` symbolisiert) abbildet. Ein elementarer Typ ist entweder *void* oder ein Basistyp, dargestellt durch einen Bezeichner (`id`). Ein zusammengesetzter Typ ist entweder ein kartesisches Produkt über Typen, ein `struct` (symbolisiert durch eckige Klammern) oder eine `union` (symbolisiert durch spitze Klammern).

Die abstrakte Schnittstellenbeschreibungssprache erlaubt die Spezifikation von rekursiven Typen, da dies für eine Definition der Typkonformität nach dem RM–ODP notwendig ist. Die Notation $\mu\, t.\alpha$ besagt, daß wenn in der Typdefinition von α ein Typ t auftritt, an dieser Stelle die Definition von α rekursiv eingesetzt wird. Der Typ $\mu\, t.\alpha$ entspricht der unendlichen Expansion der Rekursion. Das folgende Beispiel verdeutlicht den Gebrauch der abstrakten Schnittstellenbeschreibungssprache.

Beispiel 8.4: Gegeben seien die zwei Spezifikationen operationaler Schnittstellen `foo` und `bar`. `foo` ist in der CORBA–IDL und `bar` in der DCE–IDL spezifiziert.

CORBA–IDL:

```
interface foo {

  typedef struct ShortList {
    short               elem;
    sequence<ShortList> next;
  } ShortList;

  void m( out ShortList x );

};
```

DCE–IDL:

```
[local] interface bar
{

  typedef struct ShortLongList {
     short         elem;
     LongShortList next;
  } ShortLongList;

  typedef struct LongShortList {
    long          elem;
    ShortLongList next;
  } LongShortList;

  void m( [out] ShortLongList x );
}
```

Die Schnittstellenspezifikation von `foo` läßt sich in der abstrakten Schnittstellenbeschreibungssprache als $m : \mathit{void} \rightarrow \mu\, s.[\mathit{elem} : \mathit{short}, \mathit{next} : s]$ formulieren. Analog läßt sich die Spezifikation `bar` in $m : \mathit{void} \rightarrow \mu\, t.[\mathit{elem} : \mathit{long}, \mathit{next} : [\mathit{elem} : \mathit{short}, \mathit{next} : t]]$ überführen.

Definition 8.7 (Operationen auf abstrakten Schnittstellenbeschreibungen): Sei $I = O_1;\ O_2;\ \ldots;\ O_p$ mit $I \in L(G_{ASBS})$. Ein O_i entspricht der Spezifikation einer Operation der Schnittstellenspezifikation I. Insbesondere tritt das terminale Symbol „;“ in O_i nicht auf. Die Funktionen f_a und f_n seien wie folgt definiert:

1. $f_a \in \mathrm{ABB}(L(G_{ASBS}), \mathbb{N})$, $f_a(I) =_{df} n$ mit n gleich der um eins erhöhten Anzahl des terminalen Symbols „;“ in I.
2. $f_n \in \mathrm{ABB}(L(G_{ASBS}) \times \mathbb{N}, L(G_{ASBS}))$, $f_n(I, i) =_{df} \begin{cases} O_i & : \ 0 < i \leq f_a(I) \\ undef. & : \ \text{sonst} \end{cases}$

□

Neben der Definition der abstrakten Schnittstellenbeschreibungssprache werden für die formale Spezifikation zwei Operationen auf abstrakten Schnittstellenbeschreibungen benötigt. Die Funktion f_a ermittelt die Anzahl der in einer Schnittstellenbeschreibung enthaltenen Operationen. Da zwei aufeinanderfolgende Operationen in einer Schnittstellenbeschreibung immer durch das terminale Symbol „;“ voneinander getrennt sind, ist diese Funktion über deren Anzahl definiert. Die Funktion f_n ermittelt bei einer gegebenen Schnittstellenbeschreibung und Positionsangabe die entsprechende Operation innerhalb der Spezifikation.

8.6.3 Vergleichsregeln für die Typkonformität von syntaktischen Typsystemen

Auf Grundlage der abstrakten Schnittstellenbeschreibungssprache werden im folgenden unterschiedliche Vergleichsregeln für Konzeptgraphen formal spezifiziert. Für jedes syntaktische Typsystem läßt sich dabei eine entsprechende Vergleichsregel angeben. Die Möglichkeiten bei der Zuordnung sind dabei nicht notwendigerweise auf die in Kapitel 4 vorgestellten Definition von Typkonformitäten syntaktischer Typsysteme beschränkt (weiterführende Ansätze finden sich beispielsweise in [77], [76] oder [50]). Im folgenden werden die Definitionen der Typkonformitäten für die Typsysteme von DCE, CORBA und dem RM–ODP als Vergleichsregeln formal spezifiziert. Deren formale Spezifikation bedient sich der abstrakten Schnittstellenbeschreibungssprache.

Typkonformität nach DCE

Die Typkonformität von DCE erkennt zwei Schnittstellenspezifikationen genau dann als typkonform, wenn die eine eine strikte Erweiterung der anderen ist. Die Reihenfolge der in einer Schnittstellenspezifikation aufgeführten Operationen darf nicht verändert werden. Die folgende Definition spezifiziert formal eine Vergleichsregel, die der Typkonformität von DCE entspricht.

Definition 8.8 (Vergleichsregel für die Typkonformität nach DCE): Seien I_1 und I_2 zwei abstrakte Schnittstellenspezifikationen. I_1 ist typkonform zu I_2 gemäß der Regeln von DCE, geschrieben $I_1 \leq_{DCE} I_2$, genau dann, wenn ein $I' \in L(G_{ASBS})$ existiert, mit $I_1 = I_2;\ I'$. □

Die Definition beruht auf der Konkatenation von Wörtern einer Sprache. Die Schnittstellenspezifikation I' wird zusammen mit dem terminalen Trennzeichen „;“ an I_2 konkateniert. Die obige Definition nimmt keinen Bezug auf die UUID und Versionsnummer einer DCE–Schnittstellenspezifikation, da in der abstrakten Schnittstellenspezifikation diese Information nicht vorhanden ist. Die Definition 8.8 beschreibt die Typkonformität zweier Schnittstellenspezifikationen auf Grund der in ihnen definierten Operationen und nicht bzgl. der von einem Programmierer zugeordneten UUID und Versionsnummer.

Typkonformität nach CORBA

Die Definition der Typkonformität nach CORBA erweitert die von DCE in bezug auf die Reihenfolge, in der die Operationen in einer Schnittstellenspezifikation auftreten. Die formale Spezifikation der Vergleichsregel für CORBA's Typkonformität basiert auf einer Teilmengenbeziehung der Operationen beider Schnittstellenspezifikationen.

Definition 8.9 (Vergleichsregel für die Typkonformität nach CORBA): Seien I_1 und I_2 zwei abstrakte Schnittstellenspezifikationen mit $I_1 = O_1^1;\ O_2^1;\ \ldots;\ O_p^1$ und $I_2 = O_1^2;\ O_2^2;\ \ldots;\ O_q^2$. I_1 ist typkonform zu I_2 gemäß der Regeln von CORBA, geschrieben $I_1 \leq_{CORBA} I_2$, genau dann, wenn

1. $p \geq q$
2. eine injektive und totale Funktion $g \in \mathrm{ABB}([1 \ldots q], [1 \ldots p])$ existiert mit $\forall\, 1 \leq i \leq q : O_i^2 = O_{g(i)}^1$.

□

Typkonformität nach ODP

Das RM–ODP erweitert die Definition der Typkonformität nach CORBA, indem es neben der Unabhängigkeit der Reihenfolge der Operationen noch den Inklusions–Polymorphismus von Parametertypen berücksichtigt. Was die Typkonformität nach dem RM–ODP von den bisher vorgestellten Typkonformitäten für syntaktische Typsysteme unterscheidet, ist, daß die Struktur der Parametertypen für den Typvergleich herangezogen wird.

Ein *Basistyp* definiert eine Menge von gültigen Werten, die dem Basistyp zugeordnet sind. Der Basistyp stellt hierdurch eine Abstraktion einer Menge von Werten dar. Der Basistyp `long` definiert beispielsweise den Wertebereich der ganzen Zahlen im Interval $[-2^{31} \ldots 2^{31} - 1]$. Die *Subtyprelation* zwischen Basistypen ist eine Form von Typkonformität. Ein Basistyp T_1 ist ein Subtyp eines Basistyps T_2 genau dann, wenn der Wertebereich von T_1 eine Teilmenge des Wertebereichs von T_2 ist. So ist beispielsweise der Basistyp `short` mit dem Wertebereich $[-2^{15} \ldots 2^{15} - 1]$ ein Subtyp von `long`. Die Subtyprelation ist hierdurch ein Beispiel für den Inklusions–Polymorphismus.

Die Teilmengenrelation induziert einen Verband auf der Potenzmenge aller Werte, die den Basistypen zugeordnet sind. Das größte Element ist der Basistyp $\top$, dem alle Werte zugeordnet sind. Das kleinste Element ist der Basistyp $\bot$, dem kein Wert zugeordnet ist.

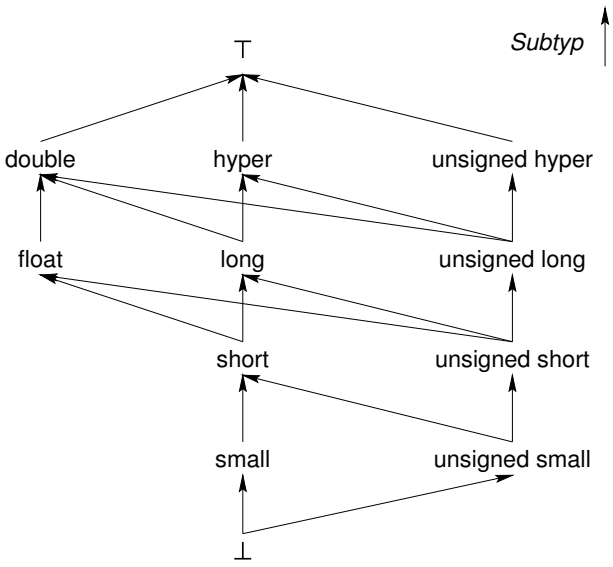


Abbildung 8.10: Anordnung der Basistypen von DCE und CORBA in einer Subtyphierarchie.

Abbildung 8.10 zeigt die Anordnung aller Basistypen, die in DCE und CORBA definiert sind, gemäß der Subtyprelation.

Die Typkonformität von operationalen Schnittstellenspezifikationen nach dem RM–ODP basiert auf der strukturellen Gleichheit rekursiver Typen. In der nachfolgenden Definition, die dem RM–ODP X.903 entnommen ist, beschreibt die Notation $\alpha[\beta/t]$, daß in α alle Vorkommen von t durch β ausgetauscht werden. Die Eigenschaft $\alpha \downarrow t$ bedeutet, daß entweder t in der Definition von α durch den Rekursionsoperator μ gebunden ist, oder α bzgl. der Konstruktionsregel für zusammengesetzte Typen der abstrakten Schnittstellenbeschreibungssprache expandiert werden kann.

Definition 8.10 (Strukturelle Gleichheit rekursiver Typen): Die Relation $=$ definiert eine strukturelle Gleichheit auf der Menge der abstrakten Schnittstellenspezifikationen.

$$\alpha = \alpha \tag{E.1}$$

$$\alpha = \beta \Rightarrow \beta = \alpha \tag{E.2}$$

$$\alpha = \beta, \beta = \gamma \Rightarrow \alpha = \gamma \tag{E.3}$$

$$\alpha_1 = \alpha_2, \beta_1 = \beta_2 \Rightarrow \alpha_1 \rightarrow \beta_1 = \alpha_2 \rightarrow \beta_2 \tag{E.4}$$

$$\alpha = \beta \Rightarrow \mu\, t.\alpha = \mu\, t.\beta \tag{E.5}$$

$$\forall\, i \in \{1, \ldots, n\}, \alpha_i = \beta_i \Rightarrow \alpha_1 \times \ldots \times \alpha_n = \beta_1 \times \ldots \times \beta_n \tag{E.6}$$

$$\forall\, i \in \{1, \ldots, n\}, \alpha_i = \beta_i \Rightarrow \langle a_1 : \alpha_1, \ldots, a_n : \alpha_n \rangle = \langle a_1 : \beta_1, \ldots, a_n : \beta_n \rangle \tag{E.7}$$

$$\forall\, i \in \{1, \ldots, n\}, \alpha_i = \beta_i \Rightarrow [a_1 : \alpha_1, \ldots, a_n : \alpha_n] = [a_1 : \beta_1, \ldots, a_n : \beta_n] \tag{E.8}$$

$$\mu\, t.t = \bot \tag{E.9}$$

$$\alpha[\mu\, t.\alpha/t] = \mu\, t.\alpha \tag{E.10}$$

$$\gamma[\alpha/t] = \alpha, \gamma[\beta/t] = \beta, \gamma \downarrow t \Rightarrow \alpha = \beta \tag{E.11}$$

□

Die Eigenschaften E.1, E.2 und E.3 machen die strukturelle Gleichheit von abstrakten Schnittstellenspezifikationen zu einer Äquivalenzrelation. Die Eigenschaften E.4, E.6, E.7 und E.8 besagen, daß zusammengesetzte Typen genau dann gleich sind, wenn ihre Komponenten gleich sind. E.5 sagt aus, daß zwei rekursive Typen gleich sind, wenn die Rekursion über der gleichen Variable definiert ist und die Rümpfe der beiden rekursiven Typen gleich ist. E.9 definiert die Bedeutung des Ausdrucks $\mu\, t.t$. Die Regeln E.10 und E.11 induzieren, daß die strukturelle Gleichheit über die reine syntaktische Gleichheit hinausgeht. Die Regel E.10 besagt, daß ein rekursiver Typ, der einmal expandiert wird, wieder zu sich selbst gleich ist. Die Regel E.11 hingegen besagt, daß zwei Typen, die beide Fixpunkte eines dritten Typs sind, gleich sein müssen. Die Bedingung $\gamma \downarrow t$ garantiert die Eindeutigkeit der Fixpunkte (siehe [5]).

Das RM–ODP definiert die Typkonformität von rekursiven Typen basierend auf dem syntaktischen Herleitungsbegriff. Der Ausdruck $\Gamma \vdash \alpha \leq \beta$ besagt, daß aus Γ hergeleitet werden kann, daß α ein Subtyp von β ist. Γ repräsentiert eine Menge von Hypothesen, die *a priori* angenommenen Subtypbeziehungen zwischen Basistypen enthält. Die Menge Γ könnte beispielsweise die in Abbildung 8.10 dargestellte Subtyphierarchie enthalten. Der syntaktische Herleitungsbegriff $\vdash$ ist durch die folgenden Regeln S.1 bis S.10 definiert. Die Regeln basieren auf Inferenzregeln der Form $\frac{\alpha}{\beta}$ (um β zu zeigen, muß α nachgewiesen werden). Die nachfolgende Definition ist dem RM–ODP X.903 entnommen.

Definition 8.11 (Strukturelle Typkonformität rekursiver Typen): Die Relation $\leq$ ist über die Inferenzregeln S.1 bis S.10 wie folgt definiert:

$$\frac{\alpha = \beta}{\Gamma \vdash \alpha \leq \beta} \tag{S.1}$$

$$\frac{\Gamma \vdash \alpha \leq \beta, \Gamma \vdash \beta \leq \gamma}{\Gamma \vdash \alpha \leq \gamma} \tag{S.2}$$

$$\frac{t \leq s \in \Gamma}{\Gamma \vdash t \leq s} \tag{S.3}$$

$$\frac{}{\Gamma \vdash \bot \leq \alpha} \tag{S.4}$$

$$\frac{}{\Gamma \vdash \alpha \leq \top} \tag{S.5}$$

$$\frac{\Gamma \vdash \alpha_2 \leq \alpha_1, \Gamma \vdash \beta_1 \leq \beta_2}{\Gamma \vdash \alpha_1 \to \beta_1 \leq \alpha_2 \to \beta_2} \tag{S.6}$$

$$\frac{\forall\, i \in \{1, \ldots, n\}, \Gamma \vdash \alpha_i \leq \beta_i}{\Gamma \vdash \langle a_1 : \alpha_1, \ldots, a_m : \alpha_m \rangle \leq \langle a_1 : \beta_1, \ldots, a_n : \beta_n \rangle} \text{ mit } n \leq m \tag{S.7}$$

$$\frac{\forall\, i \in \{1, \ldots, n\}, \Gamma \vdash \alpha_i \leq \beta_i}{\Gamma \vdash [a_1 : \alpha_1, \ldots, a_m : \alpha_m] \leq [a_1 : \beta_1, \ldots, a_n : \beta_n]} \text{ mit } n \leq m \tag{S.8}$$

$$\frac{\forall\, i \in \{1, \ldots, n\}, \Gamma \vdash \alpha_i \leq \beta_i}{\Gamma \vdash \alpha_1 \times \ldots \times \alpha_n \leq \beta_1 \times \ldots \times \beta_n} \tag{S.9}$$

$$\frac{\Gamma \cup \{t \leq s\} \vdash \alpha \leq \beta}{\Gamma \vdash \mu\, t.\alpha \leq \mu\, s.\beta}\ t \text{ nur in } \alpha;\ s \text{ nur in } \beta;\ s, t \text{ nicht in } \Gamma \tag{S.10}$$

□

Die Regeln S.1 bis S.10 definieren eine Typkonformität für abstrakte Schnittstellenspezifikationen, die auf einem Daten–Polymorphismus beruhen (siehe [105]). Die Regel S.1 definiert, daß strukturell gleiche Typen auch typkonform sind. Die Regel S.2 macht die Typkonformität nach dem RM–ODP transitiv. Nach Regel S.3 sind Hypothesen über die Typkonformität aus Γ auch syntaktisch herleitbar. Die Regeln S.4 und S.5 definieren die Rollen der ausgezeichneten Basistypen $\top$ und $\bot$. Regel S.6 definiert die Kontra– und Kovarianz. Die Regeln S.7 und S.8 besagen, daß zusammengesetzte Typen mit `struct` und `union` Komponentenweise erweitert werden können. Die übereinstimmenden Komponenten müssen in einer vorgegebenen Richtung typkonform zueinander sein. Das gilt ebenso für das kartesische Produkt in Regel S.9. Regel S.10 definiert letztendlich die Typkonformität für rekursive Typen.

Die Hypothesenmenge Γ kann auf die lexikographische Datenbank des wissensbasierten Dienstvermittlers abgebildet werden. Sowohl in der Hypothesenmenge als auch in der lexikographischen Datenbank ist allgemeingültiges Wissen vermerkt. Das allgemeingültige Wissen der Hypothesenmenge bezieht sich auf die Typkonformität von Basistypen. Im folgenden bezeichnet Γ_L die lexikographische Datenbank, die eine Hypothesenmenge enthält. Auf Basis der Definition 8.10 und 8.11 kann die Typkonformität von abstrakten Schnittstellenspezifikationen gemäß dem RM–ODP wie folgt definiert werden:

Definition 8.12 (Vergleichsregel für die Typkonformität nach dem RM–ODP): Seien I_1 und I_2 zwei abstrakte Schnittstellenspezifikationen mit $I_1 = o_1^1 : sig_1^1;\ o_2^1 : sig_2^1;\ \ldots;\ o_p^1 : sig_p^1$ und $I_2 = o_1^2 : sig_1^2;\ o_2^2 : sig_2^2;\ \ldots;\ o_q^2 : sig_q^2$. I_1 ist typkonform zu I_2 bzgl. Γ_L gemäß der Regeln des RM–ODP, geschrieben $\Gamma_L \vdash I_1 \leq_{ODP} I_2$, genau dann, wenn

1. $p \geq q$
2. eine injektive und totale Funktion $g \in \mathrm{ABB}([1 \ldots q], [1 \ldots p])$ existiert mit $\forall\, 1 \leq i \leq q : o_i^2 = o_{g(i)}^1$ und $\Gamma_L \vdash sig_i^2 \leq sig_{g(i)}^1$.

□

Nach dem RM–ODP ist ein Typ T_1 typkonform zu einem Typ T_2 genau dann, wenn für jede Operation aus T_2 eine Operation aus T_1 existiert, für die die Operationsnamen gleich sind und die Signaturen in einer strukturellen Typkonformität für rekursive Typen nach den Regeln S.1 bis S.10 stehen. Das folgende Beispiel verdeutlicht diese Definition der Typkonformität.

Beispiel 8.5: Es sei $\{\texttt{short} \leq \texttt{long}\} \in \Gamma_L$. Für die Schnittstellenspezifikationen *foo* und *bar* aus Beispiel 8.4 gilt $\Gamma_L \vdash foo \leq_{ODP} bar$. Ein formaler Beweis dieser Aussage ist in Anhang B zu finden. Informal gilt die Typkonformität wegen der Kovarianzregel und der Struktur der rekursiven Typen `ShortList` und `LongShortList`. Der Typ den `ShortList` repräsentiert, entspricht einer unendlichen Folge des Basistyps `short` (siehe Abbildung 8.11). Analog gilt für den Typ `LongShortList`, daß er einer unendlichen Folge der Basistypen `long` und `short` entspricht. Da alle Elemente der beiden Folgen paarweise zueinander typkonform sind, trifft dies auch für die zusammengesetzten rekursiven Typen `ShortList` und `LongShortList` zu.

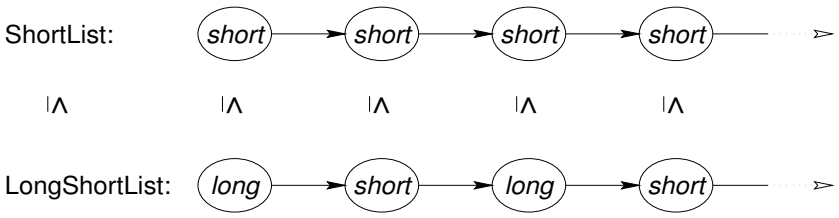


Abbildung 8.11: Unendliche Expansion der Parametertypen.

8.7 Zusammenfassung

Die Notation der Konzeptgraphen als Typbeschreibungssprache eines wissensbasierten Typsystems erlaubt die Approximation der Intension eines Diensttyps. Daraus leiten sich Anforderungen an einen wissensbasierten Dienstvermittler ab, der den sukzessiven Aufbau von Typspezifikationen unterstützen muß. Die Vermittlung von Typen resultiert in mehrfachen Interaktionen mit dem Dienstvermittler und seinen Klienten. Ist dem wissensbasierten Diestvermittler eine Extension eines Typs nicht bekannt, so erlaubt die Diensttyphierarchie die systematische Traversierung aller exportierten Typen. Durch das Lernen von weiteren Extensionen verfeinert der Dienstvermittler sein Wissen über die Typen, die er verwaltet. Die Qualität der Typvermittlung steigt hierdurch im Laufe der Zeit, je mehr Extensionen der Dienstvermittler erlernt.

Die wissensbasierte Dienstvermittlung ist in unterschiedlichen Bereichen anwendbar. Ein Konzeptgraph kann über einen entsprechenden Konzept– und Relationskatalog Typspezifikationen auf unterschiedlichen Abstraktionsebenen kodieren. In diesem Kapitel wurde die Kodierung natürlichsprachlicher Spezifikationen und operationaler Schnittstellenspezifikationen in Form von Konzeptgraphen vorgeführt. Aus den unterschiedlichen Abstraktionsebenen, auf denen die Spezifikation von Konzeptgraphen angesiedelt sein kann, ergeben sich verschiedene Vergleichsregeln. Eine Vergleichsregel dient der Überprüfung der Typkonformität und ist abhängig von den Anforderungen an die Zuordnung von Angebot und Nachfrage.

Kapitel 9

Prototyp eines wissensbasierten Dienstvermittlers

Die in dieser Arbeit vorgestellten Konzepte der wissensbasierten Dienstvermittlung sind über einen Zeitraum von drei Jahren an der Professur für verteilte Systeme und Betriebssysteme im Fachbereich Informatik der Universität Frankfurt implementiert worden. Der *AI–Trader* ist ein Prototyp eines wissensbasierten Dienstvermittlers und das Ergebnis der Implementierungsarbeiten. Dieses Kapitel liefert einen Überblick über den Prototypen, ohne auf Implementationsdetails einzugehen. Zunächst werden in Abschnitt 9.1 die Richtlinien für die Implementierung, sowie die Architektur des Prototypen mit seinen Komponenten, vorgestellt. Den zentralen Kern bildet eine Kommandosprache, die sowohl die operationale Schnittstelle des wissensbasierten Dienstvermittlers spezifiziert, als auch zugleich als textorientierte Benutzerschnittstelle dient. Da die Kommandosprache die vollständige Funktionalität des *AI–Traders* widerspiegelt, ist diese in Abschnitt 9.2 ausführlich dokumentiert.

9.1 Architektur des Prototypen

Für eine konsequente Umsetzung der Konzepte der wissensbasierten Dienstvermittlung sind Vorgaben für die Entwicklung eines Prototypen notwendig. Die Vorgaben garantieren eine funktionstüchtige Implementation, die nicht eine Produktreife zum Ziel hat, sondern die Konzepte durch einen Prototypen demonstriert. Folgende Faktoren haben Einfluß auf das Design des Prototypen:

- Portierbarkeit
- proprietäre Werkzeuge
- Effizienz
- verteilte Implementierung
- Benutzerschnittstelle

Um die Implementation portabel zu gestalten, muß die Verwendung plattformspezifischer Eigenschaften vermieden werden. Aus diesem Grund kamen proprietäre Werkzeuge und Klassenbibliotheken nicht zum Einsatz. Es wurden nur frei verfügbare und allgemein zugängliche Werkzeuge und Bibliotheken verwendet. Die Komplexität bei der Verwaltung und Bearbeitung von Konzeptgraphen verlangt eine effiziente Implementierung auf Basis einer systemnahen Programmiersprache. Ferner sollte der Prototyp in einer verteilten heterogenen Umgebung laufen, um die Dienstvermittlung in offenen verteiten Systemen zu demonstrieren. Die Wahl von Konzeptgraphen für Diensttypspezifikationen legt zudem eine graphische Benutzerschnittstelle nahe.

Um eine portable und effiziente Implementierung des Prototypen zu gewährleisten, fiel die Wahl auf die Programmiersprache C++ und die *Standard Template Library* als Bibliothek für Standard–Datenstrukturen (siehe [101] und [7]). Die Programmiersprache C++ erlaubt eine objektorientierte Strukturierung des Prototypen, die die methodische Implementierung in einer Arbeitsgruppe unterstützt. Für die Kommunikation zwischen Rechnern kommen wahlweise der ONC–RPC der Firma Sun oder Orbix 2.0 der Firma Iona zum Einsatz (siehe [93] und [41]). Der ONC–RPC, der auf allen Unix–Plattformen frei verfügbar ist, bietet eine Möglichkeit der Kommunikation über Rechnergrenzen hinweg. Der Prototyp wurde ferner im Rahmen eines Kooperationsprojekts auf Orbix 2.0 portiert. Orbix 2.0 ist eine CORBA–Implementierung und erlaubt eine Interoperabilität nach den Standards der OMG.

Die graphische Benutzerschnittstelle, die eine Maus–gesteuerte Eingabe von Konzeptgraphen erlaubt, liegt ebenfalls in zwei Varianten vor. Eine Implementierung basiert auf der Skriptsprache Tcl mit dem Tk–Toolkit für graphische Oberflächenprogrammierung (siehe [80]). Das Werkzeug Tcl/Tk ist frei verfügbar und bietet auf allen gängigen Systemen ein einheitliches *Look–and–Feel*. Eine andere Implementierung setzt auf der Programmiersprache Java der Firma Sun auf (siehe [102], [23]). Java bietet den Vorteil, daß es von einem Compiler in einen Bytecode übersetzt wird, der architekturunabhängig ist. In Java geschriebene Programme sind hierdurch auf allen Plattformen ausführbar, für die ein Interpreter für den Bytecode vorliegt. Beide Varianten, die Implementierung der Oberfläche in Tcl/Tk und in Java, besitzen einen identischen Funktionsumfang. In beiden Implementierungen sind Benutzerschnittstellen für den Import und Export von Diensttypen realisiert.

Der *AI–Trader* als Prototyp eines wissensbasierten Dienstvermittlers wurde im Rahmen von drei Diplomarbeiten realisiert (siehe [65], [35] und [48]). Der *AI–Trader* läuft als eigenständiger Unix–Prozeß. Dienstanbieter und –nutzer, die ebenfalls als separate Unix–Prozesse laufen, kommunizieren mittels des ONC–RPC oder des Orbix Methodenaufrufs mit dem Dienstvermittler. Die operationale Schnittstelle des *AI–Traders* ist als eine Menge von Befehlen in Form von ASCII–Zeichenketten definiert. Ein im *AI–Trader* vorgeschalteter Parser, implementiert mit Hilfe der Parser–Generatoren Lex und Yacc (siehe [56]), analysiert die Befehle, übersetzt Parameter in interne Datenstrukturen und reicht sie dann an den Dispatcher weiter. Die Antworten des *AI–Traders* bestehen wiederum aus ASCII–Zeichenketten, deren Analyse und Auswertung in der Verantwortung des Aufrufers liegen. Die Struktur der vom *AI–Trader* akzeptierten Befehle ist Gegenstand des nächsten Abschnitts.

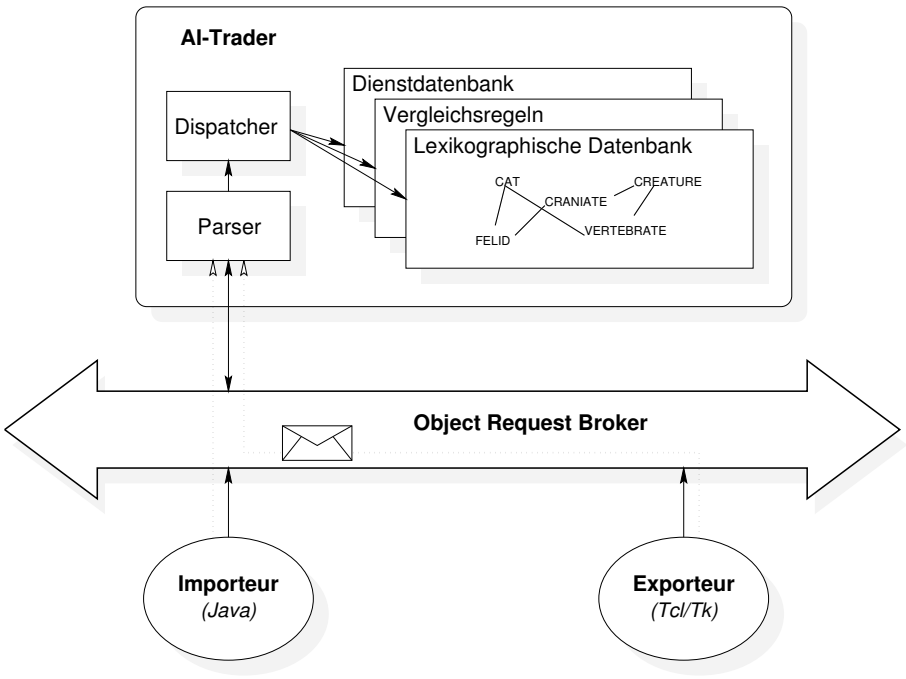


Abbildung 9.1: Implementierung des *AI–Traders*.

Der *AI–Trader* ist intern in drei Komponenten aufgeteilt, die unterschiedliche Aufgaben der wissensbasierten Dienstvermittlung übernehmen (siehe Abbildung 9.1). Die *Dienstdatenbank* enthält alle Angebote, die Dienstanbieter über eine Export–Operation zuvor registriert haben. Die Dienstdatenbank bildet Konzeptgraphen auf Referenzen ab. Ein Konzeptgraph entspricht einer Dienstbeschreibung und die zugeordnete Referenz einem Dienstanbieter, der an seiner Schnittstelle den Dienst anbietet. Die Dienstdatenbank bietet Operationen zur Manipulation von Konzeptgraphen und zur Verwaltung von Dienstanbietern an.

Der *AI–Trader* verwaltet durch eine weitere Komponente die *Vergleichsregeln*, die für Import und Export–Vorgänge benötigt werden. Der Vergleich zweier Konzeptgraphen ist parameterisiert, d.h. der Algorithmus für die Vergleichsregeln ist nicht fest programmiert. Die Regeln können vielmehr zur Laufzeit definiert und modifiziert werden. Als Paradigma für die Spezifikation von Vergleichsregeln wurde die logische Programmierung unter Prolog gewählt, da sie ausführbare Spezifikationen ermöglicht. Das frei verfügbare SWI–Prolog (siehe [113]), eine Prolog Implementierung gemäß dem *Edinburgh Standard*, wurde in die Komponente für Vergleichsregeln integriert. Im Rahmen des *AI–Trader*–Projekts wurden verschiedene linguistische– sowie Subtyp–Vergleichsregeln spezifiziert (siehe Anhang A).

Als eine weitere Komponente enthält der *AI–Trader* eine *lexikographische Datenbank*, die Wörter und Relationen zwischen Wörtern verwaltet. Sowohl Wörter als auch Relationen können frei definiert werden. Den Relationen werden bei ihrer Definition mathematische Eigenschaften zugeordnet, die später Anfragen an die lexikographische Datenbank beeinflußen. Es wurden Teile der Daten des semantischen Wörterbuchs WordNet importiert (siehe [71]). Die in WordNet enthaltenen Substantive, sowie Synonym–, Antonym–

und Hyponym–Beziehungen wurden in die Kommandosprache des *AI–Traders* übersetzt. Insgesamt enthält die lexikographische Datenbank auf Basis von WordNet ca. 87.000 Substantive sowie ca. 95.000 Relationen.

9.2 Kommandosprache

Der *AI–Trader* bietet seine Funktionalität an einer operationalen Schnittstelle an. Die operationale Schnittstelle ist definiert durch eine Menge von Nachrichten mit Ein– und Ausgabeparametern, die an der Schnittstelle akzeptiert werden. Anstatt die Definition der *AI–Trader*–Schnittstelle auf proprietären Standards — wie beispielsweise die IDL eines Herstellers — beruhen zu lassen, wurde der Weg über eine *Kommandosprache* als Schnittstelle gewählt. Ein Befehl der Kommandosprache besteht aus einer Zeichenkette, die aus mehreren lexikalischen Einheiten besteht, die durch Leerzeichen voneinander getrennt sind. Die erste lexikalische Einheit der Zeichenkette bezeichnet ein Kommando, während die übrigen lexikalischen Einheiten Parameter sind[1]. Die Rückgabeparameter des *AI–Traders* sind ebenfalls in Form von Zeichenketten definiert.

Die Verwendung einer Kommandosprache bietet neben der Unabhängigkeit von proprietären Standards (und damit höherer Portabilität) eine Reihe weiterer Vorteile:

1. Spezifikation der Schnittstelle
2. textuelle Benutzerschnittstelle
3. Persistenz der internen Datenstrukturen
4. Kommunikation in einer heterogenen Umgebung

Zwischen dem wissensbasierten Dienstvermittler und seinen Klienten werden ausschließlich Zeichenketten auf Basis der Syntax der Kommandosprache ausgetauscht. Die Interaktionen des Vermittlers mit seiner Umgebung können in einer Datei in lesbarer Form mitprotokolliert werden. Die Kommandosprache dient weiterhin als textuelle Benutzerschnittstelle, die eine Interaktion mit dem *AI–Trader* erlaubt. Da auf den *AI–Trader* ausschließlich über die Kommandosprache Einfluß genommen werden kann, können seine internen Datenstrukturen als Folge von Befehlen dauerhaft auf einem externen Medium gesichert werden. Mit dem nächsten Starten des *AI–Traders* werden die zuvor gespeicherten Befehle wieder eingelesen und die internen Datenstrukturen sukzessive aufgebaut. Zuletzt überwindet die Definition der Kommandosprache auf Basis des ASCII–Zeichensatzes die Heterogenität unterschiedlicher Datendarstellungen in einem offenen verteilten System.

Eine zentrale Komponente des *AI–Traders* ist ein Parser, der einen eingehenden Befehl aus der Kommandosprache in interne Aktionsfolgen umsetzt (siehe auch Abbildung 9.1). Die Ausgabeparameter eines Befehls werden wiederum in Form einer Zeichenkette dargestellt, die der Aufrufer weiter verarbeiten kann. Die einzelnen Befehle, die der *AI–Trader* akzeptiert, sind gemäß der Funktionalität seiner Komponenten in solche für die lexikographische Datenbank, den Vergleichsregeln, der Konzeptgraphenverwaltung und

[1] Die unter dem Unix–Betriebssystem für die Befehlseingabe üblichen Shells arbeiten nach dem gleichen Prinzip. Ein Kommando ist definiert als Folge von Zeichen des ASCII–Zeichensatzes.

der Dienstanbieterverwaltung sowie sonstigen Befehle unterteilt und in den folgenden Abschnitten detailliert beschrieben. Die Darstellung der Syntax der Befehle beruht auf der *Extended Backus–Naur Form* (EBNF) (siehe [114]).

9.2.1 Antworten des *AI–Traders*

Jedes Kommando, das an den *AI–Trader* gerichtet wird, erzeugt genau eine Antwort. Alle Antworten außer **@error** sind *positive Antworten*, daß heißt, der vorhergehende Befehl wurde korrekt abgearbeitet. Ein Fehler, welcher während der Abarbeitung eines beliebigen Kommandos auftrifft, wird einheitlich über die **@error**–Antwort gemeldet.

@ok
Die Antwort signalisiert die korrekte Abarbeitung eines Kommandos, die jedoch keine weiteren Rückgabeparameter besitzt.

@error <num> : <string>
Die Bearbeitung eines Kommandos erzeugte einen Fehler, der über diese Antwort dem Aufrufer mitgeteilt wird. Die Antwort enthält eine eindeutige Fehlernummer sowie eine Klartext–Fehlermeldung.

@answer-graph = [<cg>]
Die Antwort auf ein Kommando ist ein Konzeptgraph aus der Dienstdatenbank.

@answer-flag = (**@yes** | **@no**)
Die Antwort ist ein Bool'scher Rückgabewert.

@answer-string = <string>
Die Antwort besteht aus einer Zeichenkette, deren Inhalt von dem zuvor abgesetzten Kommando abhängt.

@answer-matches = [<key> { : <key> }]
Mit dieser Antwort werden eine Menge von Indizes aus der Dienstdatenbank an den Aufrufer zurückgegeben. Die Konzeptgraphen, die durch die Indizes referenziert werden, können mit dem Kommando **@get-graph** abgerufen werden (siehe unten).

@answer-providers = [<provider> { : <provider> }]
<provider> ::= <maintainer> , <oid>
Die Antwort besteht aus einer Menge von Dienstanbietern. Ein Dienstanbieter ist charakterisiert über eine Beschreibung (<maintainer>) und einer Referenz auf seine Schnittstelle (<oid>).

@answer-words = [<word> { : <word> }]
Die Antwort enthält eine Menge von Wörtern aus der lexikographischen Datenbank als Rückgabewert.

@answer-key = <key>
Als Antwort auf ein Kommando wird ein einzelner Index auf die Dienstdatenbank zurückgegeben.

9.2.2 Lexikographische Datenbank

Die lexikographische Datenbank verwaltet Wörter und Relationen zwischen zwei beliebigen Wörtern. Wörter und Relationen können dynamisch zur Laufzeit hinzugefügt und wieder gelöscht werden. Die Typen von Relationen sind nicht fest kodiert, sondern können ebenfalls zur Laufzeit über ihre mathematische Eigenschaften registriert werden.

@register-relation <rel> [: { **@symmetric** | **@reflexive** | **@transitive** | **@acyclic** }]
Definiert in der lexikographischen Datenbank eine neue Relation. Deren mathematische Eigenschaften werden über eine Kombination der Schlüsselwörter **@symmetric** (symmetrisch), **@reflexive** (reflexiv), **@transitive** (transitiv) und **@acyclic** (azyklisch) spezifiziert.

@show-all-relations
Alle in der lexikographischen Datenbank registierten Relationen werden über **@answer-string** ausgegeben.

@add-word <word>
Ein neues Wort wird in die lexikographische Datenbank aufgenommen.

@contains-word <word>
Überprüft, ob ein Wort in der lexikographischen Datenbank enthalten ist. Das Ergebnis wird über die Antwort **@answer-flag** mitgeteilt.

@delete-word <word>
Ein zuvor eingetragenes Wort wird wieder aus der lexikographischen Datenbank gelöscht.

@add-relation <word> : <rel> : <word>
Zwischen zwei Wörtern, die zuvor mittels **@add-word** in die lexikographische Datenbank eingetragen worden sein müssen, werden mit einer Relation verbunden, die ebenfalls zuvor mit dem Kommando **@add-relation** registriert worden sein muß.

@get-relation-pred <word> : <rel>
Bestimmt alle Vorgänger eines Wortes bezüglich einer Relation. Die Menge der Wörter, die mit dem Wort in Relation stehen, werden über die Antwort **@answer-words** ausgegeben.

@get-relation-succ <word> : <rel>
Bestimmt alle Nachfolger eines Wortes bezüglich einer Relation. Die Menge der Wörter, die mit dem Wort in Relation stehen, werden über die Antwort **@answer-words** ausgegeben.

@are-related <word> : { <rel> }+ : <word>
Überprüft, ob zwei in der lexikographischen Datenbank enthaltene Wörter über eine Wortkette bezüglich einer Menge von Relationen in Beziehung zueinander stehen. Das Ergebnis wird über **@answer-flag** dem Aufrufer mitgeteilt.

9.2.3 Verwaltung von Vergleichsregeln

Die Verwaltung von Vergleichsregeln bezieht sich auf die Verwaltung der Prolog Quelltexte, die während eines Vergleichs zweier Konzeptgraphen aktiviert werden. Die dafür vorgesehenen Kommandos sehen die üblichen Befehle, wie beispielsweise Anlegen und Löschen von Vergleichsregeln, vor. Der Befehl **@match** nimmt bezug auf die zuvor gespeicherten Vergleichsregeln.

@define-matching-rule <name> <rule> **@end**
: Definiert eine neue Vergleichsregel <rule> unter dem Namen <name>. Existiert die Regel bereits, wird die alte Definition überschrieben.

@delete-matching-rule <name>
: Die Vergleichsregel mit dem Namen <name> wird gelöscht.

@show-matching-rules
: Die Namen sämtlicher Vergleichsregeln werden über eine **@answer-string** Antwort ausgegeben.

@show-matching-rule <name>
: Der Quelltext einer Vergleichsregel mit dem Namen <name> wird über eine **@answer-string** Antwort ausgegeben.

@match { <name> }+ **:** <cg>
: Durchsucht die Dienstdatenbank nach einem Graphen, der bezüglich der vorgegebenen Vergleichsregeln mit dem Graphen <cg> übereinstimmt. Das Ergebnis ist eine Menge von Datenbankindizes über die Antwort **@answer-matches**.

9.2.4 Konzeptgraphenverwaltung

Die in diesem Abschnitt beschriebenen Befehle beziehen sich auf die Verwaltung von Konzeptgraphen innerhalb der Dienstdatenbank. Alle Konzeptgraphen sind über einen Datenbankindex eindeutig benannt. Alle Befehle nehmen ausschließlich über einen Index auf die in der Dienstdatenbank gespeicherten Konzeptgraphen bezug.

@add <cg>
: Fügt einen neuen Konzeptgraphen in die Dienstdatenbank ein. Als Ergebnis erhält der Aufrufer über eine **@answer-key** Antwort einen Datenbankindex, mit dem der Konzeptgraph wieder referenziert werden kann.

@add <key> **:** <cg>
: Fügt unter einem gegebenen Index einen Konzeptgraphen ein. Existiert unter dem Index bereits ein Konzeptgraph, so wird dieser zuvor gelöscht.

@remove <key>
: Entfernt den zuvor mit **@add** eingefügten Konzeptgraph aus der Dienstdatenbank, der mit dem Index <key> referenziert wird.

@join <key> : <cg>
Zu einem in der Dienstdatenbank existierenden Konzeptgraphen mit dem Index <key> wird eine weitere Sicht in Form des Konzeptgraphen <cg> hinzugefügt.

@get-graph <key>
Liefert als Ergebnis über die Antwort **@answer-graph** einen Konzeptgraphen aus der Dienstdatenbank, der mit dem Index <key> referenziert wird.

@number-of-variants <word>
Liefert zu einem Konzepttyp die Anzahl der Graphen in der Dienstdatenbank, deren Wurzelkonzepttyp mit <word> übereinstimmt.

@get-graph <word> : <int>
Liefert den <int>'ten Graphen über **@answer-key** als Antwort, dessen Wurzelkonzepttyp mit <word> übereinstimmt.

9.2.5 Dienstanbieterverwaltung

Ein weiterer Teil der Dienstdatenbank speichert Dienstangebote. Ein Dienstanbieter beschreibt sich selbst über einen Klartext und der Adresse seiner operationalen Schnittstelle. Ein Dienstangebot wird einem Konzeptgraphen zugeordnet, der der Diensttypspezifikation des Anbieters entspricht. Zu einem Konzeptgraphen (d.h. Diensttyp) können zu einem Zeitpunkt mehrere Anbieter existieren.

@add-provider <key> : <maintainer> , <oid>
Fügt zu einem Konzeptgraphen — in der Dienstdatenbank durch <key> referenziert — einen neuen Dienstanbieter hinzu.

@remove-provider <key> , <oid>
Ein Dienstanbieter zieht für den Konzeptgraphen <key> sein Angebot zurück.

@get-provider <key>
Für einen Konzeptgraphen in der Dienstdatenbank werden alle Anbieter ermittelt, die ihren Dienst für diesen Graphen zuvor registriert haben. Die Antwort ist definiert über **@answer-providers**.

9.2.6 Sonstige Befehle

Dieser Abschnitt enthält die Befehle, die keiner konkreten Komponente des *AI–Traders* zugeordnet sind. Besonders hervorzuheben sind die Befehle **@save** und **@import**, die die Persistenz der internen Datenstrukturen aller Komponenten ermöglichen.

@version
Liefert die Versionsnummer des *AI–Traders* zurück.

@verbose-on
Ab sofort werden während der Abarbeitung von Vergleichsregeln Diagnosemeldungen ausgegeben.

@verbose-off
Ab sofort werden bei der Ausführung von Vergleichsregeln keine Diagnosemeldungen mehr erzeugt.

@import <file>
Die Datei mit dem Namen <file> wird eingelesen und die dort enthaltenen Kommandos ausgeführt.

@save <file>
Der aktuelle Zustand des *AI–Traders* wird in der Datei <file> abgespeichert. Der Zustand wird dabei in Form von den hier beschriebenen Kommandos ausgegeben, die bei einem späteren **@import** den Zustand wieder herstellen.

9.3 Spezifikation von Vergleichsregeln

Das Kommando **@define-matching-rule** aus Abschnitt 9.2.3 erlaubt die Registrierung einer neuen Vergleichsregel. Die Syntax der Vergleichsregel basiert auf der Prolog–Notation nach dem *Edinburgh Standard*. Eine Vergleichsregel wird vom *AI–Trader* mit zwei Konzeptgraphen als Parameter aufgerufen. Der Befehlssatz des Prolog–Subsystems wurde um einige Prädikate erweitert, die einen Zugriff auf die Komponenten des *AI–Traders* erlauben (wie beispielsweise die lexikographische Datenbank). Diese neun Prädikate, auf die innerhalb einer Vergleichsregel zugegriffen werden kann, sind im folgenden gemäß der Prolog–Konventionen dokumentiert. Eine Variable mit vorangestelltem + bezeichnet eine Eingabevariable. Ein vorangestelltes − bezeichnet eine Ausgabevariable und ein ? eine Ein– und Ausgabevariable.

match(+*queryGraph*, +*typeGraph*, −*matchResult*)
Vergleicht die Anfrage *queryGraph* eines Importeurs mit dem Diensttyp *typeGraph* eines Exporteurs. Stimmen Angebot und Nachfrage überein, wird die Variable *matchResult* mit `accept` unifiziert, ansonsten mit `reject`. Jede Vergleichsregel muß dieses Prädikat definieren. Es dient dem *AI–Trader* als Einstiegspunkt für den Vergleich zweier Konzeptgraphen.

cg_match(+*queryGraph*, +*typeGraph*, ?*matchResult*)
Dieses Prädikat wendet alle ausgewählten Vergleichsregeln auf die Konzeptgraphen *queryGraph* und *typeGraph* an. Mit diesem Prädikat kann der Vergleich von Konzeptgraphen rekursiv auf Teilgraphen angewendet werden. Die Semantik der Argumente stimmt mit der des Prädikats `match`/3 überein.

cg_getsucc(+*cgNode*, ?*successorList*)
Unifiziert *successorList* mit einer Liste von Nachfolgern des Knotens *cgNode*. Die Elemente der Liste sind wieder *cgNodes*. Das Prädikat ermittelt die Nachfolger sowohl von Konzept– als auch Relationsknoten.

cg_getname(+*cgNode*, ?*nodeName*)
Unifiziert *nodeName* mit einer internen Repräsentation eines Konzept– oder Relationsknotens *cgNode*.

cg_gettype(+*cgNode*, ?*nodeType*)
Testet, ob ein *cgNode* ein Konzept– oder Relationsknoten ist. Die Variable *nodeType* wird entsprechend mit dem Wert `concept` oder `relation` unifiziert.

cg_getvalue(+*cgNode*, ?*value*)
Unifiziert *value* mit dem Wert des Konzeptknotens *cgNode*. Das Prädikat schlägt fehl, wenn *cgNode* kein Konzeptknoten ist oder keinen Wert besitzt.

cg_getinst(+*cgNode*, ?*instanceList*)
Ermittelt die Menge von Instanzen eines Konzeptknotens. Die Variable *instanceList* wird mit der Instanzliste des Knotens *cgNode* unifiziert. Das Prädikat schlägt fehl, wenn *cgNode* kein Konzeptknoten ist.

cg_matchword(+*word*1, +*relationList*, +*word*2)
Überprüft, ob die Wörter *word*1 und *word*2 in der lexikographischen Datenbank über die in *relationList* angegebenen Relationen verbunden sind (siehe auch das Kommando `@are-related` aus dem letzten Abschnitt). Das Prädikat schlägt fehl, wenn keine Wortkette konstruiert werden kann. Ist *relationList* leer, so werden die Wörter syntaktisch verglichen, ohne Berücksichtigung von Groß– und Kleinschreibung. *word*1 und *word*2 sind entweder Zeichenketten oder von `cg_getname`/2 gelieferte Konzept– oder Relationsnamen aus einem Konzeptgraphen.

cg_trace(+*format*, +*argList*)
Gibt eine Statusmeldung aus, wenn deren Ausgabe aktiviert ist (siehe Kommando `@verbose-on` des letzten Abschnitts). Die Argumente entsprechen denen des Prädikats `format`/2.

9.4 Zeitmessungen

Von allen Operationen, die der *AI–Trader* an seiner Schnittstelle anbietet, ist der Import von Diensten der zeitkritischste. In einem offenen verteilten System ist zu erwarten, daß Dienstnutzer im Mittel sich öfters an den Vermittler wenden, als Dienstanbieter. Wie Statistiken des AltaVista–Suchdienstes zeigen, müssen dort täglich über 13 Millionen Import–Operationen bewältigt werden (siehe [4]). Dieser Abschnitt präsentiert einige empirische Untersuchungen des Prototypens, mit der die Laufzeiten von Import–Operationen gemessen wurde.

Für den Test wurden zunächst zwei Schnittstellenspezifikationen ausgewählt. Beide basieren auf der Spezifikation der operationalen Schnittstelle eines ODP–konformen Vermittlers, die mit Hilfe der CORBA–IDL Notation im Annex A der im Dokument *ODP–trading function* (siehe [46]) zu finden ist. Der Quelltext der besagten Schnittstellenspezifikation ist über 600 Zeilen lang und enthält mehrere benutzerdefinierte Typen, Schnittstellen und Module. Die erste Schnittstellenspezifikation, im folgenden als T_1 bezeichnet, ist identisch mit der vollständigen Spezifikation, so wie sie im ODP–Dokument vorgegeben ist. Die zweite Schnittstellenspezifikation, im folgenden mit T_2 bezeichnet, entspricht T_1 ohne dessen Export– und Management–Schnittstellen.

DCE		CORBA		ODP	
$T_1 \leq_{DCE} T_1$	6.72 sek	$T_1 \leq_{CORBA} T_1$	7.02 sek	$T_1 \leq_{ODP} T_1$	9.39 sek
$T_2 \not\leq_{DCE} T_1$	7.21 sek	$T_2 \not\leq_{CORBA} T_1$	7.90 sek	$T_2 \not\leq_{ODP} T_1$	3.83 sek
$T_1 \leq_{DCE} T_2$	3.51 sek	$T_1 \leq_{CORBA} T_2$	3.65 sek	$T_1 \leq_{ODP} T_2$	5.60 sek
$T_2 \leq_{DCE} T_2$	3.61 sek	$T_2 \leq_{CORBA} T_2$	3.71 sek	$T_2 \leq_{ODP} T_2$	4.59 sek

Tabelle 9.1: Zeitmessung unterschiedlicher Subtyp–Vergleichsregeln.

Beide Schnittstellenspezifikationen wurden anschließend mit dem in Abschnitt 8.6.1 beschriebenen Übersetzer in Konzeptgraphen übersetzt. Der Konzeptgraph, der aus T_1 hervorgeht, enthält 4.754 Konzeptknoten und 2.740 Relationsknoten. Analog enthält der Konzeptgraph, der aus T_2 hervorgeht, 2.638 Konzeptknoten und 1.479 Relationsknoten. Die Messungen beziehen sich auf die Zeit, die der Prototyp benötig, um die beiden Konzeptgraphen auf Typkonformität zu testen. Bei drei Vergleichsregeln (die von DCE, CORBA und ODP), sowie zwei Konzeptgraphen, ergeben sich insgesamt 12 unterschiedliche Test. Um beispielsweise $T_1 \leq_{DCE} T_2$ zu testen, wird erst T_1 exportiert und dann T_2 bzgl. der DCE Vergleichsregel für Typkonformität importiert. Für alle Vergleichsregeln ist T_2 nie typkonform zu T_2, dargestellt durch die Notation $\not\leq_i$. Der Grund dafür liegt in der Konstruktion von T_1 und T_2. T_2 besitzt einige Operationen nicht, die in T_1 definiert sind, und kann hierdurch unter keiner Vergleichsregel typkonform zu T_2 sein.

Die Messungen wurden auf einer IBM RS/6000 Modell 580 durchgeführt. Tabelle 9.1 faßt die Ergebnisse der Messungen zusammen. Die Tabelle enthält die Durchschnittszeiten für eine Import–Operation. Die Ergebnisse, die alle im Sekundenbereich für den Vergleich von zwei Konzeptgraphen liegen, deuten auf die Komplexität der wissensbasierten Typkonformität hin. Die Vergleichsregel für die Typkonformität nach ODP benötigt im Mittel länger als die Regeln von DCE und CORBA. Der Grund dafür liegt in der Komplexität der ODP–Vergleichsregel, die strukturelle Typkonformität von rekursiven Typen überprüft[2]. Auf zwei Ergebnisse der Zeitmessung sei explizit hingewiesen. Beide beziehen sich auf die zweite Reihe in Tabelle 9.1. Einerseits ist zu klären, warum die Zeiten für $T_2 \not\leq_{DCE} T_1$ und $T_2 \not\leq_{CORBA} T_1$ länger als die übrigen Zeiten der gleichen Spalte sind. Andererseits ist zu klären, warum die Zeit für $T_2 \not\leq_{ODP} T_1$ kürzer als die anderen Werte der gleichen Spalte ist.

Das Prolog–Subsystem muß während des Vergleich zweier Konzeptgraphen nach den Regeln von DCE und CORBA über *Backtracking* alle Permutationen testen, um Teilgraphen zur Übereinstimmung zu bringen. Erst wenn die letzte Permutation fehlschlägt, kann das endgültige Ergebnis hergeleitet werden. Aus diesem Grund benötigt der *AI–Trader* mehr Zeit um festzustellen, daß zwei Konzeptgraphen nach den DCE– und CORBA–Vergleichsregeln *nicht* zueinander typkonform sind. Der Grund, daß mit der ODP–Vergleichsregel das gleiche Ergebnis schneller hergeleitet werden kann, liegt dagegen an der Spezifikation der Vergleichsregel selbst. Diese Vergleichsregel überprüft die struk-

[2]Einen Hinweis dafür ergibt sich aus dem Umfang der Vergleichsregeln. Während die Vergleichsregel für DCE und CORBA jeweils um die 70 Zeilen Prolog–Kode umfaßen, ist die Vergleichregel für ODP ca. 700 Zeilen Prolog–Kode lang.

turellen Eigenschaften einer Typspezifikation. Kann für einen benutzerdefinierten Typ beispielsweise die Typkonformität nicht hergeleitet werden, schlägt die Typkonformität sofort für die vollständige Typspezifikation fehl.

Aus den in Tabelle 9.1 abgebildeten Zeitmessungen muß die Schlußfolgerung gezogen werden, daß die Implementierung des Prototypen den Anforderungen eines offenen verteilten Systems nicht genügt. Es bieten sich verschiedene Möglichkeiten an, daß Laufzeitverhalten des Prototypens zu verbessern. Einerseits könnte aus einem Konzepgraphen ein Hash–Wert berechnet werden, so daß vor Anwendung einer rechenintensiven Vergleichregel das Ergebnis u.U. einer Hash–Tabelle direkt entnommen werden kann. Weiterhin könnten die Vergleichsregeln zu Gunsten eines verbesserten Laufzeitverhaltens in einer imperativen Programmiersprache neu implementiert werden. Der Nachteil dieser Lösung ist, daß die Ausdruckskraft der ausführbaren Spezifikationen von Prolog verloren geht.

9.5 Zusammenfassung

Der *AI–Trader* ist ein Prototyp eines wissensbasierten Dienstvermittlers, der im Verlaufe von drei Jahren implementiert wurde. Der Prototyp läuft in einer verteilten heterogenen Umgebung. Seine operationale Schnittstelle basiert auf einer Kommandosprache, die sowohl als textuelle Benutzerschnittstelle dient, als auch der Überwindung der Heterogenität von Rechnersystemen. Die Konzepte, die dem Prototypen eines wissensbasierten Dienstvermittler zugrunde liegen, sind in mehreren wissenschaftlichen Veröffentlichungen dokumentiert (siehe [82, 88, 87, 86, 84, 85]).

Im Rahmen der Implementierungsarbeiten entstand ein Übersetzer für CORBA–IDL und DCE–IDL. Der Übersetzer ist in der Lage, eine beliebige Schnittstellenspezifikation auf Basis der CORBA–IDL oder DCE–IDL in einen Konzeptgraphen umzuwandeln, sowie einen Konzeptgraphen wahlweise in eine der beiden Schnittstellenbeschreibungssprachen wieder zurück zu übersetzen. Die dafür entwickelten Werkzeuge sind in die graphische Benutzerschnittstelle des *AI–Traders* integriert.

Die Implementierung des *AI–Traders* ist nicht von proprietären Werkzeugen oder Bibliotheken abhängig. Der *AI–Trader* wurde auf IBM–AIX, Sun–Solaris und Linux portiert. Die graphische Benutzeroberfläche ist ebenfalls für alle drei Plattformen verfügbar (siehe Abbildung 9.2). Aktuelle Informationen zum *AI–Trader*–Projekt sind über das *World Wide Web* abrufbar (siehe [83]). Dort sind die vollständigen Quelltexte der Implementierung sowie vor–kompilierte Versionen der oben genannten Plattformen frei im Rahmen der *GNU–Copyright Notice* verfügbar.

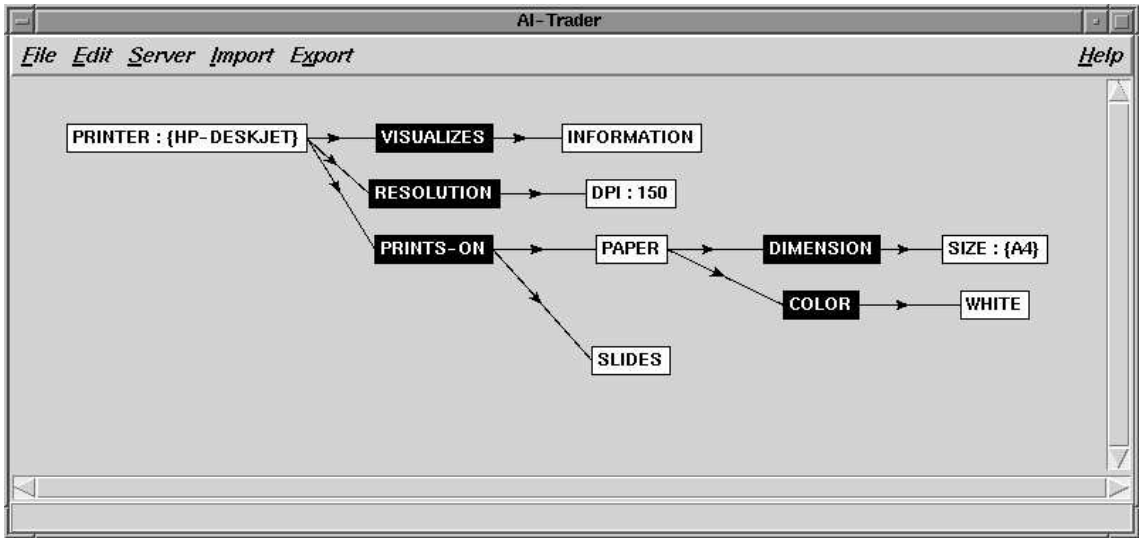


Abbildung 9.2: Graphische Benutzerschnittstelle des *AI–Traders*.

Kapitel 10

Zusammenfassung und Ausblick

10.1 Bewertung der Ergebnisse

Das Thema dieser Arbeit ist die Dienstvermittlung in offenen verteilten Systemen und die Rolle, die ein Typsystem dabei einnimmt. Ein Typsystem besteht aus einer Typbeschreibungssprache und der Definition einer Typkonformität. Die Typbeschreibungssprache erlaubt die Spezifikation von Typen, wohingegen mit der Typkonformität während eines Vermittlungsvorgangs überprüft wird, ob Angebot und Nachfrage zusammenpassen. In dieser Arbeit wurde zunächst nachgewiesen, daß es sinnvoll ist, bei einem Typ zwischen seiner Intension und seiner Extension zu unterscheiden. Die Intension eines Typs ist die Gesamtheit aller Beschreibungen, die auf diesen zutreffen. Die Extension eines Typs repräsentiert dagegen eine konkrete Beschreibung (d.h. Spezifikation eines Dienstangebots). Eine Interpretation ordnet jeder Extension eine Intension zu.

Um in einem offenen verteilten System Dienste vermitteln zu können, müssen sich Dienstnutzer und –anbieter auf die Extensionen aller Typen einigen. Einem Typ kommt hierdurch die Rolle eine Standards zu, der allen beteiligten Parteien *a priori* bekannt sein muß. Daraus resultiert eine injektive Interpretation, die jeder Intension genau eine Extension zuordnet. Die eindeutig bestimmte Extension einer Intension fungiert als systemweiter Standard. Ein Typ als Standard steht im Widerspruch zu der Vielfalt und Dynamik eines offenen Dienstmarktes. Der Standardisierungsprozeß von Extensionen, der einem Vermittlungsvorgang vorausgehen muß, hemmt gerade die Dynamik des Systems. Die Konsequenz daraus ist, daß neben den Diensten auch die Diensttypen Gegenstand der Vermittlung sein müssen. Diese Schlußfolgerung ist bisher noch nicht formuliert worden. Es wäre somit wünscheswert, nicht–injektive Interpretationen zuzulassen, so daß eine Intension mehrere Extensionen besitzen kann, die unterschiedliche Sichten der Dienstnutzer und –anbieter repräsentieren.

Die Analyse einiger bestehender Typsysteme zeigte, daß mit diesen eine nicht–injektive Interpretation nicht realisierbar ist. Im Hauptteil dieser Arbeit wurden zwei neue Typsysteme vorgestellt, die diese Eigenschaft unterstützen. Das deklarative Typsystem erweitert die Schnittstellenbeschreibungssprache eines syntaktischen Typsystems, indem semantische Spezifikationen zugelassen werden. Die deklarative Semantik dient dabei als Grundlage für die Beschreibung der Semantik einer Typspezifikation. Die Extension entspricht einem definiten Programm bestehend aus einer endlichen Menge von Horn–Klauseln. Die

Intension eines Typs korrespondiert mit dem kleinsten Herbrand–Modell des definiten Programms, welches die semantische Spezifikation des Typs darstellt. Die Forderung nach der Möglichkeit nicht–injektiver Interpretationen ergibt sich aus den Eigenschaften der deklarativen Semantik, wonach verschiedene definite Programme ein identisches kleinstes Herbrand–Modell besitzen können.

Das zweite in dieser Arbeit vorgestellte Typsystem entspringt einem wissensbasierten Ansatz. Grundlage bildet eine Wissensrepräsentationstechnik, die anwenderbezogene semantische Spezifikationen erlaubt. Ein Konzeptgraph als wissensbasierte Typspezifikation vereinigt in sich unterschiedliche Beschreibungen eines Typs. Ein Konzeptgraph, der selbst eine Extension darstellt, repräsentiert somit die Vereinigung mehrerer Extensionen eines Typs. Die Intension ist jedoch durch einen Konzeptgraph nicht eindeutig bestimmt. Dieser stellt lediglich eine Approximation dar. Hier liegt ein fundamentaler Unterschied in den beiden Typsystemen. Während eine Extension im deklarativen Typsystem auch immer eindeutig eine Intension charakterisiert, ist dies bei dem wissensbasierten Typsystem nicht der Fall. Die Konsequenz daraus ist, daß dieser Umstand bei einem Vermittlungsvorgang berücksichtigt werden muß. Ein wissensbasierter Vermittler muß über ein spezielles Vermittlungsprotokoll die Verfeinerung einer wissensbasierten Typspezifikation erlauben, die zu einer besseren Approximation der Intension führt.

Das deklarative Typsystem besitzt aufgrund der Unentscheidbarkeit der deklarativen Typkonformität keine praktische Relevanz. Es zeigt jedoch, wie mit Hilfe der deklarativen Semantik der *Open World Assumption* genüge geleistet werden kann. Im Vergleich dazu kann das wissensbasierte Typsystem als „Fuzzyfizierung“ des deklarativen Typsystems angesehen werden. Die wissensbasierte Typbeschreibungssprache ermöglicht im Sinne der Fuzzy Logik unscharfe Spezifikationen, die im Laufe der Zeit verfeinert werden. Ein Vorteil des wissensbasierten Ansatzes ist die Möglichkeit von anwenderbezogenen Typspezifikationen. Ein anderer Vorteil besteht darin, daß eine wissensbasierte Typbeschreibungssprache eine Meta–Sprache repräsentiert, in der Spezifikationen aus anderen Domänen dargestellt werden können. Ungeachtet dieser Vorteile bleibt jedoch der Beweis offen, daß die wissensbasierte Dienstvermittlung tatsächlich eine geeignete Methodik für die Vermittlung von Typen darstellt.

10.2 Offene Probleme und Ausblick

Die Praktikabilität der wissensbasierten Dienstvermittlung kann nicht formal bewiesen werden. Es existieren keine Metriken, mit der die Eignung von Benutzerschnittstellen für Anwender bewertet werden könnten. Der Wert eines wissensbasierten Typsystems liegt in der Möglichkeit von Spezifikationen auf unterschiedlichen Abstraktionsebenen, wie dies bei der Übersetzung von operationalen Schnittstellenspezifikationen nach Konzeptgraphen demonstriert wurde. Hierdurch kann ein generischer Vermittler implementiert werden, der ausschließlich mit Konzeptgraphen arbeitet. Die Ergebnisse dieser Arbeit könnten in die aktuellen Standardisierungsbemühungen der ISO zu einem *Type Repository* einfließen.

Das Problem der Benutzerakzeptanz eines wissensbasierten Dienstvermittlers relativiert sich, wenn Übersetzer zwischen der natürlichen Sprache und den Konzeptgraphen entwickelt sein werden. Dem Anwender bleibt dann die zugrundeliegende Technik in bezug auf die Wissensrepräsentation verborgen. Das Rahmenwerk der in dieser Arbeit vor-

gestellten wissensbasierten Dienstvermittlung bleibt davon unberührt. Ein Ausblick für zukünftige Arbeiten liegt somit im Bereich der Benutzerschnittstellen und im speziellen in der Analyse natürlicher Sprache.

Die Zeitmessungen aus dem letzten Kapitel zeigen, daß der aktuelle Prototyp für den praktischen Einsatz im Hinblick auf die Laufzeitkomplexität Schwächen besitzt. Eine vorrangige Aufgabe wird daher eine effiziente Implementierung eines wissensbasierten Dienstvermittlers sein. Obwohl die in dieser Arbeit verwendete Terminologie sich an der des RM–ODP anlehnt, ist der Prototyp aus operationaler Sicht nicht konform zu der von der ISO standardisierten *Trading Function*. Eine Re–Implementierung sollte somit diesen Standard berücksichtigen, der mittlerweile weite Verbreitung erlangt hat und beispielsweise von der OMG übernommen wird.

Eine Möglichkeit das Konzept der wissensbasierten Vermittlung zu erweitern leitet sich dem Bereich der *Component Based Systems* ab. In diesem Forschungszweig steht Komposition von Objekten zu einem Aggregat im Vordergrund und nicht wie bei der Objektorientierung in der Wiederverwendbarkeit der Objektklassen. Ein Beispiel aus der Touristikbranche verdeutlicht diese Notwendigkeit. Um eine Reise zu buchen sind in der Regel mehrere spezialisierte Dienste wie beispielsweise Transport, Unterkunft, Programm vor Ort, Reiseversicherung, etc. notwendig. Anstatt über ein Reisebüro die Buchung abzuwickeln, könnte ein Reisender direkt mit Hilfe eines *Component Based Systems* mit den spezialisierten Anbietern Fluggesellschaft, Hotel, Versicherung, etc. interagieren.

Eine Spezifikation auf Basis einer Typbeschreibungssprache beschreibt hier einen *Component Type*. Die Aufgabe des Vermittlers besteht in diesem Fall darin, aus einer Menge von Typen einen neuen Typ zu synthetisieren, der gerade den Anforderungen des *Component Types* entspricht. Die Synthese liefert dabei Hinweise, wie die Objekte „komponiert“ werden müssen um den Anforderungen eines Dienstnutzers gerecht zu werden. Diese Technik vermeidet Zwischenhändler und flexibilisiert die strenge eins zu eins Zuordnung von Dienstnutzern und Dienstanbietern.

Die Motivation dieser Arbeit beruht auf der Annahme, daß sich in einem globalen Netzwerk ein elektronischer Dienstmarkt ausbildet, in dem Dienste frei angeboten und nachgefragt werden. Die Vermittlung ist eine von vielen wichtigen Aufgaben, ohne die ein derartiger Dienstmarkt nicht funktionsfähig ist. Die ISO hat über einen eigenen Standard, der bereits erwähnten *Trading Function*, dieser Tatsache Rechnung getragen. In dieser Arbeit wird zum ersten Mal eine Brücke zwischen dem akademischen Ansatz der Dienstvermittlung nach den Vorstellungen der ISO und dem pragmatischen Ansatz in Form von Suchmaschinen im Internet geschlagen. Es ist abzusehen, daß eine gegenseitige Beeinflußung in Zukunft stärker werden wird. In diesem Sinn stellt diese Arbeit einen ersten Schritt in Richtung neuer Methoden bei der Vermittlung und der Spezifikation von Diensten dar.

Anhang A

Initiale Wissensbasis

In diesem Anhang wird eine mögliche initiale Wissensbasis des *AI–Traders* vorgestellt. Die initiale Wissensbasis wird bei einem Neustart des *AI–Traders* eingelesen und definiert seinen Anfangszustand. Sie ist ausschließlich über die in Kapitel 9 vorgestellte Kommandosprache definiert. Nach Einlesen der initialen Wissensbasis sind alle in den Abschnitten 8.5 und 8.6 spezifizierten Vergleichsregeln definiert, sowie die lexikographische Datenbank geladen. Im folgenden wird kurz auf die einzelnen Abschnitte der initialen Wissensbasis über die links stehenden Zeilennummern eingegangen.

Zunächst werden in den Zeilen 2–4 drei Relationen zusammen mit ihren mathematischen Eigenschaften in der lexikographischen Datenbank registriert, sowie in Zeile 6 das Wurzelelement der Konzepthierarchie `SOMETHING` definiert. Aus einer Datei Namens `wordnet.kb` werden die aus WordNet extrahierten Substantive in die lexikographische Datenbank eingelesen. Anschließend werden in den Zeilen 10–20 die 11 Einstiegspunkte der Hyponym–Hierarchie aus WordNet mit dem Wurzelelement `SOMETHING` verbunden.

Als nächstes folgen die in Abschnitt 8.5 vorgestellten Vergleichsregeln. In den Zeilen 24–63 ist die Vergleichsregel `Specialization`, in den Zeilen 67–115 die Vergleichsregel `Quantity` und schließlich in den Zeilen 138–158 die Vergleichsregel `Negation` enthalten. Zu der Vergleichsregel `Quantity` ist in den Zeilen 119–134 die in Abbildung 8.7 enthaltene Konzept– und Relationshierarchie definiert. Am Ende der initialen Wissensbasis werden die in Abschnitt 8.6 spezifizierten Vergleichsregeln importiert (Zeilen 196–198). In den Zeilen 163–193 wird die in Abbildung 8.10 dargestellte Subtyphierarchie für Basistypen mit Hilfe einer eigenen Relation `SUBTYPE` in der lexikographischen Datenbank registriert.

```
 1
 2  @register-relation 'SYNONYM'    : @reflexive, @transitive, @symmetric
 3  @register-relation 'ANTONYM'    : @symmetric
 4  @register-relation 'IS_A'       : @reflexive, @transitive, @acyclic
 5
 6  @add-word 'something'
 7
 8  @import wordnet.kb
 9
10  @add-relation 'entity'                : 'IS_A' : 'something'
11  @add-relation 'psychological_feature' : 'IS_A' : 'something'
12  @add-relation 'abstraction'           : 'IS_A' : 'something'
```

```
13  @add-relation 'location'              : 'IS_A' : 'something'
14  @add-relation 'shape'                 : 'IS_A' : 'something'
15  @add-relation 'state'                 : 'IS_A' : 'something'
16  @add-relation 'event'                 : 'IS_A' : 'something'
17  @add-relation 'act'                   : 'IS_A' : 'something'
18  @add-relation 'group'                 : 'IS_A' : 'something'
19  @add-relation 'possession'            : 'IS_A' : 'something'
20  @add-relation 'phenomenon'            : 'IS_A' : 'something'
21
22
23
24  @define-matching-rule 'Specialization'
25
26    isa(Node1, Node2) :-
27      cg_getname(Node1, Name1), cg_getname(Node2, Name2),
28      cg_matchword(Name1, ['IS_A', 'SYNONYM'], Name2).
29
30    match(Q, T, Res) :-
31      cg_gettype(Q, concept),
32      isa(T, Q),
33      cg_getsucc(T, Tsucc), cg_getsucc(Q, Qsucc),
34      gensym(reject_seen, RejectSym),
35      match_lists_con(Qsucc, Tsucc, Res, RejectSym).
36
37    match(Q, T, Res) :-
38      cg_gettype(Q, relation),
39      isa(T, Q),
40      cg_getsucc(T, Tsucc), cg_getsucc(Q, Qsucc),
41      match_lists_rel(Qsucc, Tsucc, Res).
42
43    match_lists_con([], _, accept, _) :- !.
44
45    match_lists_con(Qlist, Tlist, accept, RejectSym) :-
46      member(Q, Qlist), member(T, Tlist),
47      cg_match(Q, T, Res),
48      (Res = reject
49      ->  flag(RejectSym, _, 1), fail
50      ;   true),
51      !.
52
53    match_lists_con(_, _, reject, RejectSym) :-
54      flag(RejectSym, 1, 0).
55
56    match_lists_rel([], _, accept).
57
58    match_lists_rel([Q|Qlist], Tlist, Res) :-
59      select(Tlist, T, Trest),
60      cg_match(Q, T, accept),
```

```
61      match_lists_rel(Qlist, Trest, Res).
62
63  @end
64
65
66
67  @define-matching-rule 'Quantity'
68
69    isa(Node1, Node2) :-
70      cg_getname(Node1, Name1), cg_getname(Node2, Name2),
71      cg_matchword(Name1, ['IS_A', 'SYNONYM'], Name2).
72
73    isa_str(Node1, Name2) :-
74      cg_getname(Node1, Name1),
75      cg_matchword(Name1, ['IS_A', 'SYNONYM'], Name2).
76
77    match(Qcon, Tcon, accept) :-
78      cg_gettype(Qcon, concept),
79      isa(Tcon, Qcon),
80      cg_getsucc(Qcon, QrelList), cg_getsucc(Tcon, TrelList),
81      member(Qrel, QrelList), member(Trel, TrelList),
82      isa_str(Qrel, "COMPARATOR"), isa_str(Trel, "QUANTITY"),
83      cg_getsucc(Qrel, [Qcon2]), cg_getsucc(Trel, [Tcon2]),
84      isa(Tcon2, Qcon2),
85      cg_getvalue(Qcon2, Qval), cg_getvalue(Tcon2, Tval),
86      cg_getname(Qrel, Comp),
87      quantity_match(Qval, Comp, Tval).
88
89
90    quantity_match(Qval, Comp, Tval) :-
91      cg_matchword(Comp, ['IS_A', 'SYNONYM'], "EQUAL"),
92      !,
93      Qval = Tval.
94
95    quantity_match(Qval, Comp, Tval) :-
96      cg_matchword(Comp, ['IS_A', 'SYNONYM'], "LESS"),
97      !,
98      Qval < Tval.
99
100   quantity_match(Qval, Comp, Tval) :-
101     cg_matchword(Comp, ['IS_A', 'SYNONYM'], "LESS OR EQUAL"),
102     !,
103     Qval =< Tval.
104
105   quantity_match(Qval, Comp, Tval) :-
106     cg_matchword(Comp, ['IS_A', 'SYNONYM'], "GREATER"),
107     !,
108     Qval > Tval.
```

```
109
110   quantity_match(Qval, Comp, Tval) :-
111     cg_matchword(Comp, ['IS_A', 'SYNONYM'], "GREATER OR EQUAL"),
112     !,
113     Qval >= Tval.
114
115 @end
116
117
118
119 @add-word "QUANTITY"
120 @add-word "COMPARATOR"
121
122 @add-word "EQUAL"
123 @add-word "LESS"
124 @add-word "LESS OR EQUAL"
125 @add-word "GREATER"
126 @add-word "GREATER OR EQUAL"
127
128 @add-relation "QUANTITY"           : 'IS_A' : "SOMETHING"
129
130 @add-relation "EQUAL"              : 'IS_A' : "COMPARATOR"
131 @add-relation "LESS OR EQUAL"      : 'IS_A' : "COMPARATOR"
132 @add-relation "LESS"               : 'IS_A' : "LESS OR EQUAL"
133 @add-relation "GREATER OR EQUAL"   : 'IS_A' : "COMPARATOR"
134 @add-relation "GREATER"            : 'IS_A' : "GREATER OR EQUAL"
135
136
137
138 @define-matching-rule 'Negation'
139
140   isa(Node1, Node2) :-
141     cg_getname(Node1, Name1), cg_getname(Node2, Name2),
142     cg_matchword(Name1, ['IS_A', 'SYNONYM'], Name2).
143
144   antonym(Node1, Node2) :-
145     cg_getname(Node1, Name1), cg_getname(Node2, Name2),
146     cg_matchword(Name1, ['IS_A', 'ANTONYM'], Name2).
147
148   match(Qcon, Tcon, reject) :-
149     cg_gettype(Qcon, concept),
150     isa(Tcon, Qcon),
151     cg_getsucc(Qcon, QrelList), cg_getsucc(Tcon, TrelList),
152     member(Qrel, QrelList), member(Trel, TrelList),
153     antonym(Trel, Qrel),
154     cg_getsucc(Qrel, QconList), cg_getsucc(Trel, TconList),
155     member(Qcon2, QconList), member(Tcon2, TconList),
156     isa(Tcon2, Qcon2).
```

```
157
158 @end
159
160
161
162
163 @register-relation 'SUBTYPE' : @reflexive, @transitive, @acyclic
164
165 @add-word 'Object'              /* Interface Reference */
166 @add-word 'TypeCode'            /* Runtime Type Information */
167 @add-word "void"
168 @add-word "small"               /*  8 bit   signed integer */
169 @add-word "unsigned small"      /*  8 bit unsigned integer */
170 @add-word "short"               /* 16 bit   signed integer */
171 @add-word "unsigned short"      /* 16 bit unsigned integer */
172 @add-word "long"                /* 32 bit   signed integer */
173 @add-word "unsigned long"       /* 32 bit unsigned integer */
174 @add-word "hyper"               /* 64 bit   signed integer */
175 @add-word "unsigned hyper"      /* 64 bit unsigned integer */
176 @add-word "float"               /* 32 bit floating point */
177 @add-word "double"              /* 64 bit floating point */
178
179
180 @add-relation "float"          : 'SUBTYPE' : "double"
181 @add-relation "long"           : 'SUBTYPE' : "double"
182 @add-relation "unsigned long"  : 'SUBTYPE' : "double"
183 @add-relation "long"           : 'SUBTYPE' : "hyper"
184 @add-relation "unsigned long"  : 'SUBTYPE' : "hyper"
185 @add-relation "unsigned long"  : 'SUBTYPE' : "unsigned hyper"
186 @add-relation "short"          : 'SUBTYPE' : "float"
187 @add-relation "unsigned short" : 'SUBTYPE' : "float"
188 @add-relation "short"          : 'SUBTYPE' : "long"
189 @add-relation "unsigned short" : 'SUBTYPE' : "long"
190 @add-relation "unsigned short" : 'SUBTYPE' : "unsigned long"
191 @add-relation "small"          : 'SUBTYPE' : "short"
192 @add-relation "unsigned small" : 'SUBTYPE' : "short"
193 @add-relation "unsigned small" : 'SUBTYPE' : "unsigned short"
194
195
196 @import mrule-dce.kb
197 @import mrule-corba.kb
198 @import mrule-odp.kb
```

Anhang B

Beispiel für die Typkonformität nach dem RM–ODP

Im folgenden wird die Behauptung aus Beispiel 8.5 auf Seite 149 bewiesen. Es ist zu zeigen, daß mit $\{short \leq long\} \in \Gamma$ die Aussage $\Gamma \vdash foo \leq_{ODP} bar$ gilt. Die abstrakten Schnittstellenspezifikationen *foo* und *bar* seien wie in Beispiel 8.4 auf Seite 144 definiert.

Es sei $\alpha \equiv \mu\, s.[elem : short, next : s]$. Dann gilt:

$$\begin{aligned}
\alpha &\equiv [elem : short, next : \mu\, s.[elem : short, next : s]] \text{ nach (E.10)} \\
&\equiv [elem : short, next : [elem : short, next : \mu\, s.[elem : short, next : s]]] \text{ nach (E.10)} \\
\alpha &= [elem : short, next : [elem : short, next : \alpha]]
\end{aligned}$$

Im letzten Schritt ist der Typ α als Gleichung formuliert. Analog kann für $\beta \equiv \mu\, t.[elem : short, next : [elem : short, next : t]]$ hergeleitet werden:

$$\begin{aligned}
\beta \equiv [elem : short, next : [elem : short,& \\
&next : \mu\, t.[elem : short, next : [elem : short, next : t]]]] \text{ nach (E.10)} \\
\beta = [elem : short, next : [elem : short, next : \beta]]&
\end{aligned}$$

Der Typ β ist ebenfalls in Form einer Gleichung definiert. Gegeben sei ferner $\gamma \equiv [elem : short, next : [elem : short, next : v]]$. Die Typen α und β sind Fixpunkte von γ bzgl. der Variablen v. Da γ kontraktiv in v ist gilt nach (E.11):

$$\mu\, s.[elem : short, next : s] = \mu\, t.[elem : short, next : [elem : short, next : t]] \quad (*)$$

Es sei $\{short \leq long\} \in \Gamma$. Mit $\Gamma \vdash void \leq void$ wegen (S.1) gilt:

$$\begin{aligned}
&\Gamma \cup \{s \leq t\} \vdash [elem : short, next : s] \leq [elem : short, next : t] \text{ nach (S.8)} \\
&\Gamma \cup \{s \leq t\} \vdash [elem : short, next : [elem : short, next : s]] \leq \\
&\qquad [elem : long, next : [elem : short, next : t]] \text{ nach (S.8)} \\
&\Gamma \vdash \mu\, s.[elem : short, next : [elem : short, next : s]] \leq \\
&\qquad \mu\, t.[elem : long, next : [elem : short, next : t]] \text{ nach (S.10)}
\end{aligned}$$

$$\Gamma \vdash \mu\, s.[elem : short, next : s] \leq$$
$$\mu\, t.[elem : long, next : [elem : short, next : t]] \text{ nach } (*)$$
$$\Gamma \vdash void \rightarrow \mu\, s.[elem : short, next : s] \leq$$
$$void \rightarrow \mu\, t.[elem : long, next : [elem : short, next : t]] \text{ nach (S.6)}$$

Da *foo* und *bar* jeweils genau eine Methode mit identischen Methodennamen besitzen, gilt mit der obigen Herleitung und der Vergleichsregel für die Typkonformität nach dem RM–ODP aus Definition 8.12 auf Seite 149 die Aussage $\{short \leq long\} \vdash foo \leq_{ODP} bar$.

Anhang C

Konzepttypen für operationale Schnittstellenspezifikationen

Konzepttyp	**Instanzen**	**Beschreibung**
ARRAY_TYPE	Typbezeichner	Elementtyp in einem Array
ARRAY	Dimensionszahl	Beschreibt ein Array
BOUND	konst. Ausdruck	Begrenzung einer eindimensionalen Folge
CASE_LABEL	konst. Ausdruck	Wert für eine Fallunterscheidung in einer Vereinigung
CONSTANT_TYPE	Typbezeichner	Typ einer Konstanten
CONSTANT_VALUE	konst. Ausdruck oder Konstantenname	Verkörpert den Wert einer Konstanten
CONSTANT	Konstantenname	Leitet die Beschreibung einer Konstanten–Deklaration ein
DIMENSION_ POSITION	Positionsangabe	Dimension einer Array–Grenze
DIMENSION	konst. Ausdruck	Dimensionsgrenze in einem Array
ELEMENT_TYPE	Typbezeichner	Elementtyp in einer Folge
ENUM_MEMBER	Bezeichner	Komponente in einer Aufzählung
ENUM	Bezeichner	Identifiziert eine Aufzählung
FUNCTION_ POINTER	1: (Zeiger) oder 2: (Zeiger auf Zeiger)	Beschreibt einen Funktionszeiger
INTERFACE_ ATTRIBUTE	`local`	Schnittstellenattribut `local`
INTERFACE_ SERVER_ ENDPOINTS	Protokollfamilie und Endpunkte	Schnittstellenattribut `server_endpoints`

Konzepttyp	**Instanzen**	**Beschreibung**
INTERFACE_UUID	UUID–Nummer	Schnittstellenattribut `uuid`
INTERFACE_ VERSION	Versionsnummer	Schnittstellenattribut `version`
INTERFACE	Schnittstellenbezeichner	Leitet die Beschreibung einer Schnittstelle ein
MEMBER_ ATTRIBUTES	DCE–Attribute	Attribute einer Komponente in einem konstruierten Typ
MEMBER_ POSITION	Positionsangabe	Position einer Komponente in einem konstruierten Typ
MEMBER_TYPE	Typbezeichner	Typ einer Komponente innerhalb eines konstruierten Typs
OPERATION_ ATTRIBUTES	`maybe`, `broadcast` oder `idempotent`	Aufrufsemantik einer Operation
OPERATION_ PARAMETER	Parametername	Parameter in einer Operation
OPERATION_ POSITION	Positionsangabe	Position eines Parameters in einer Schnittstelle
OPERATION_TYPE	Typbezeichner	Rückgabetyp einer Operation
OPERATION	Operationsname	Leitet die Beschreibung einer Operation ein
PARAMETER_ ATTRIBUTES	`in`, `out` oder `inout`	Parameterattribute
PARAMETER_ POSITION	Positionsangabe	Position eines Parameters in einer Signatur
PARAMETER_TYPE	Typbezeichner	Typ eines Parameters
PIPE_TYPE	Bezeichner	Typ der Daten, die durch einen Kanal ausgetauscht werden
PIPE	Bezeichner	Identifiziert einen Kanal
POINTER_TYPE	Typbezeichner	Typ, auf welcher ein Zeiger verweist
POINTER	1: (Zeiger) oder 2: (Zeiger auf Zeiger)	Beschreibt einen Zeiger
SEQUENCE	Keine	Eine eindimensionale Folge beliebigen Typs
SPECIFICATION	keine	Wurzelknoten aller kanonischen Templates
STRING	Keine	Zeichenfolge
STRUCT_MEMBER	Bezeichner	Komponente in einer Struktur
STRUCT	keine oder Bezeichner	Identifiziert eine Struktur
TYPE_ ATTRIBUTES	DCE–Attribute	Attribute eines Typs
TYPE_TYPE	Typbezeichner	Typ eines selbstdefinierten Typs

Konzepttyp	**Instanzen**	**Beschreibung**
`TYPE`	Keine oder Typname	Leitet die Beschreibung eines selbstdefinierten Typs ein
`UNION_ DISCRIMINATOR_ TYPE`	Typbezeichner	Typ einer `switch`-Anweisung
`UNION_MEMBER`	Bezeichner	Komponente in einer Vereinigung
`UNION`	Keine oder Bezeichner	Identifiziert eine Vereinigung

Literaturverzeichnis

[1] N. Adams, P. Curtis, and M. Spreitzer. First-class data-type representations in SCHEME XEROX. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation: Albuquerque, New Mexico, June 23–25, 1993*, volume 28(6) of *ACM SIGPLAN Notices*, pages 139–146, New York, NY, USA, June 1993. ACM Press.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison–Wesley Publishing Company, 1988.

[3] S. Alagic and M.A. Arbib. *The Design of Well–Structured and Correct Programs*. Springer Verlag, 1977.

[4] AltaVista. WebCrawler. http://altavista.digital.com/, Digital Equipment, 1996.

[5] R.M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[6] P. America. Designing an object oriented programming language with behavioural subtyping. In *Foundations of Object Oriented Languages*, LNCS 489, pages 60–90, Noordwijkerhout, Netherlands, May 1990. REX/School Workshop, Springer Verlag.

[7] ANSI. C++ Standard. http://www.hab-weimar.de/springer/april-working-paper/ X3J16/95-0087 WG21/N0687, Computer Science Department, 1996.

[8] M. Apel and P. Ludz. *Philosophisches Wörterbuch*. Walter de Gruyter, sixth edition, 1976.

[9] P. Atzeni and R. Torlone. Efficient updates to independent schemes in the weak instance model. In *ACM SIGMOD International Conf. on Management of Data*, pages 84–93, 1990.

[10] S. Bapat. *Object–Oriented Networks, Models for Architecture, Operations, and Management*. Prentice/Hall International, 1994.

[11] M. Bearman and K. Raymond. Federating Traders: An ODP Adventure. In *International IFIP Workshop on Open Distributed Processing*, October 1991.

[12] T. Berners–Lee. Hypertext Transfer Protocol: A stateless search, retrieve and manipulation protocol. Technical report, Internet Draft, ftp.w3.org://pub/www/doc/http–spec.ps.Z, 1993.

[13] T. Berners–Lee and D. Connolly. Hypertext Markup Language — 2.0. Technical report, Internet Draft, ftp.w3.org://pub/www/doc/html–spec.ps.Z, 1995.

[14] T. Berners-Lee et al. The World–Wide Web. *Communications of the Association for Computing Machinery*, 37(8):76–82, August 1994.

[15] G. Booch. *Object Oriented Design with Applications*. Benjamin Cummings Publishing Company, Inc, Redwood City, California, 1991.

[16] C. Bowman et al. Scalable Internet resource discovery: research problems and approaches. *Communications of the Association for Computing Machinery*, 37(8):98–107, August 1994.

[17] W. Brookes and J. Indulska. A Type Management System for Open Distributed Processing. Technical Report 285, Department of Computer Science, Univeristy of Queensland, February 1994.

[18] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[19] Component Integration Laboratories (CIL), Shaping Tomorrow's Software (White Paper), ftp://cil.org/pub/cilabs/tech/opendoc/OD–overview.ps, 1994.

[20] J. Daniels and S. Cook. Strategies for sharing objects in distributed systems. *Journal of Object Oriented Programming (JOOP)*, pages 27–36, January 1993.

[21] Excite. WebCrawler. http://www.excite.com/, Excite Inc., 1996.

[22] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. The KDD process for extracting useful knowledge from volumes of data. *Communications of the Association for Computing Machinery*, 39(11):27–34, November 1996.

[23] D. Flanagan. *Java in a Nutshell*. O'Reilley & Associates, 1996.

[24] S.J. Garland, J.V. Guttag, and J.J. Horning. An Overview of Larch. In P.E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 329–348. Springer Verlag, Berlin, 1993.

[25] K. Geihs. Infrastrukturen für heterogene verteilte Systeme. *Informatik Spektrum*, 16(1):11–23, 1993.

[26] K. Geihs. *Client/Server Systeme, Grundlagen und Architekturen*. International Thomson Publishing GmbH, 1995.

[27] K. Geihs, H. Gründer, and A. Puder. Object sharing in open distributed processing systems. In *Entwicklung und Management verteilter Anwendungssysteme*, Goethe Universität Frankfurt, October 1993. GI/ITG Fachgruppe Kommunikation und Verteilte Systeme.

[28] K. Geihs and U. Hollberg. A Retrospective on DACNOS. *Communications of the Association for Computing Machinery*, 33(4), 1990.

[29] M.R. Genesereth et al. Knowledge Interchange Format, Version 3.0. Technical Report Logic-92-1, Department of Computer Science, Stanford University, June 1992.

[30] M.R. Genesereth and S.P. Ketchpel. Software Agents. *Communications of the Association for Computing Machinery*, 37(7):48–53, July 1994.

[31] M.R. Genesereth, N.P. Singh, and M.A. Syed. A distributed and anonymous knowledge sharing approach to software interoperation. Technical report, Computer Science Department, Stanford University, November 1994. http://logic.stanford.edu/papers/README.html.

[32] A. Goldberg. *Smalltalk–80: The Language and its Implementation*. Addison–Wesley Publishing Company, 1985.

[33] A. Goodchild. An evaluation scheme for trader user interfaces. In *3rd International IFIP TC6 Conference on Open Distributed Processing (ICODP'95)*, Brisbane, Australia, 20–24 February 1995. Chapman and Hall.

[34] A. Goscinski. *Distributed Operating Systems, The Logical Design*. Addison–Wesley Publishing Company, 1991.

[35] F. Gudermann. Ein mehrphasen Protokoll für eine wissensbasierte Dienstvermittlung in offenen verteilten Systemen. Diplomarbeit, Fachbereich Informatik, Goethe Universität, Frankfurt, February 1995.

[36] Y. Hoffner. Inter–operability and distributed application platform design. In A. Schill et al., editors, *International Conference on Distributed Platforms*, pages 342–356. Chapman & Hall, February 1996.

[37] J. Hopcroft and J. Ullman. *Formal languages and their Relation to Automata*. Addison–Wesley Publishing Company, 1969.

[38] W. Hussy. *Denken und Problemlösen — Grundriš der Psychologie*. Kohlhammer, 1993.

[39] J. Indulska, M. Bearman, and K. Raymond. A Type Management System for an ODP Trader. In *Proceedings of the IFIP TC6/WG6.1 International Conference on Open Distributed Processing*, pages 141–152, Berlin, Germany, 13–16 September 1993. North–Holland.

[40] InfoSeek. WebCrawler. http://www2.infoseek.com/, InfoSeek Corp., 1996.

[41] IONA Technologies Ltd. *Orbix 2 — Distributed Object Technology*. Iona, 1996.

[42] ITU.TS Recommendation X.901 — ISO/IEC 10746–1: Basic Reference Model of Open Distributed Processing Part 1: Overview and Guide to the use of the Reference Model, July 1994.

[43] ITU.TS Recommendation X.902 — ISO/IEC 10746–2: Basic Reference Model of Open Distributed Processing Part 2: Descriptive Model, 1994.

[44] ITU.TS Recommendation X.903 — ISO/IEC 10746–3: Basic Reference Model of Open Distributed Processing Part 3: Prescriptive Model, February 1994.

[45] ITU.TS Recommendation X.904 — ISO/IEC 10746–4: Basic Reference Model of Open Distributed Processing Part 4: Architectural Semantics, 1994.

[46] ODP Trading Function, ITU/ISO Committee Draft Standard ISO/IEC DIS13235 Rec. X.9tr, May 1995.

[47] L.A. De Paula Lima jr. and E.R.M. Madeira. A Model for a Federative Trader. In *3rd International IFIP TC6 Conference on Open Distributed Processing (ICODP'95)*, Brisbane, Australia, 20–24 February 1995. Chapman and Hall.

[48] C. Kaiser. Spezifikation operationaler Schnittstellen mittels kanonischer Konzeptgraphen–Templates. Diplomarbeit, Fachbereich Informatik, Goethe Universität, Frankfurt, August 1996.

[49] F.-J. Kauffels. *Rechnernetzwerksystemarchitekturen und Datenkommunikation.* BI–Wissenschaftsverlag, 1991.

[50] D. Konstantas. Object oriented interoperability. In O. Nierstrasz, editor, *ECOOP*, LNCS 707, pages 80–102, 1993.

[51] J. Kramer. Distributed Systems. In Morris Sloman, editor, *Network and Distributed Systems Management*, chapter 3, pages 47–64. Addison–Wesley Publishing Company, 1994.

[52] L. Kutvonen and P. Kutvonen. Broadening the User Environment with Implicit Trading. In *Proceedings of the IFIP TC6/WG6.1 International Conference on Open Distributed Processing*, pages 129–140, Berlin, Germany, 13–16 September 1993. North–Holland.

[53] G.T. Leavens and Y. Cheon. Extending CORBA–IDL to specify Behavior with Larch. Technical Report TR93–20, Department of Computer Science, Iowa State University, 1993.

[54] F. Lehmann and R. Wille. A Triadic Approach to Formal Concept Analysis. In *3rd International Conference on Conceptual Structures (ICCS'95)*, Santa Cruz, University of California, 14–18 August 1995. Springer Verlag.

[55] G. LeLann. Introduction, objectives and characterisation of distributed systems. In B.W. Lampson, M. Paul, and H.J. Siegert, editors, *Distributed Systems, Architecture and Implementation. An Advanced Course*, pages 1–8. Springer Verlag, 1980.

[56] J.R. Levine, T. Mason, and D. Brown. *Lex & Yacc.* O'Reilley & Associates, second, major rev. edition, 1992.

[57] D.D. Lewis and K. Sparck Jones. Natural language processing for information retrieval. *Communications of the Association for Computing Machinery*, 39(1):92–101, January 1996.

[58] B. Liskov and J. Wing. Family values: A behavioral notion of subtyping. Technical Report CMU–CS–93–187, Computer Science Department, Carnegie Mellon University, Pittsburgh, July 1993.

[59] J.W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, second, extended edition, 1987.

[60] H.W. Lockhart. *OSF DCE: guide to developing distributed applications*. J. Ranade workstation series. McGraw–Hill, Inc., 1994.

[61] J. Lyons. *Einführung in die moderne Linguistik*. C.H. Beck Verlag, München, eighth edition, 1995.

[62] D. MacKinnon, W. McCrumm, and D. Sheppard. *An Introduction to Open Systems Interconnection*. Computer Science Press, 1990.

[63] G. Mann. BEELINE — A Situated, Bounded Conceptual Knowledge System. In *Int. Journal of Systems Research and Information Science*, volume 7, pages 37–53. Overseas Publishers Association, 1995.

[64] J. Marcinkowski and L. Pacholski. Undecidability of the Horn–clause implication problem. In *IEEE Symposium on Foundation of Computer Science*, pages 354–362, Pittsburgh, October 1992.

[65] S. Markwitz. Mechanismen der wissensbasierten Dienstvermittlung in offenen verteilten Systemen. Diplomarbeit, Fachbereich Informatik, Goethe Universität, Frankfurt, May 1995.

[66] R.C. Merkle. Secure communications over an insecure channel. *Communications of the Association for Computing Machinery*, 21:294–299, April 1978.

[67] M. Merz and W. Lamersdorf. Cooperation support for an open service market. In *2nd International IFIP TC6 Conference on Open Distributed Processing (ICODP'93)*. Elsevier Science Publishers B. V., 1993.

[68] M. Merz, K. Müller, and W. Lamersdorf. Service trading and mediation in distributed computing environments. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS '94)*. IEEE Computer Society Press, 1994.

[69] R.S. Michalski. Learning strategies and automated knowledge acquisition. In *Computational Model of Learning*. Springer Verlag, 1987.

[70] G.A. Miller. WordNet: A Lexical Database for English. *Communications of the Association for Computing Machinery*, 38(11):39–41, November 1995.

[71] G.A. Miller et al. Five papers on WordNet. Technical Report CSL Report 43, Cognitive Science Laboratory, Princeton University, March 1993.

[72] K. Morik. Maschinelles Lernen. In Günter Görz, editor, *Einführung in die Künstliche Intelligenz*. Addison Wesley, 1993.

[73] J. S. Mullender. *Distributed Systems*. ACM–Press, frontier series, 1989.

[74] E. Najm and J.B. Stefani. A formal semantics for the ODP computational model. *Computer Networks and ISDN Systems*, 27(8):1305–1329, July 1995.

[75] N. Nicolov, C. Mellish, and G. Ritchie. Sentence Generation from Conceptual Graphs. In *3rd International Conference on Conceptual Structures (ICCS'95)*, Santa Cruz, University of California, 14–18 August 1995. Springer Verlag.

[76] O. Nierstrasz. Regular types for active objects. In *Proceedings ACM Conference on Object Oriented Programming: Systems, Languages and Applications*, OOPSLA, September 1993.

[77] O. Nierstrasz and M. Papathomas. Towards a type theory for active objects. In *ACM OOPS Messenger*, volume 2, pages 89–93, April 1991.

[78] Object Management Group (OMG), The Common Object Request Broker: Architecure and Specification, Revision 2.0, July 1995.

[79] Open Software Foundation. *Introduction to DCE*. Open Software Foundation, Inc., 1992.

[80] J. Ousterhout. *Tcl and Tk toolkit*. Addison–Wesley Publishing Company, 1994.

[81] J. Palsberg and M.I. Schartzbach. *Object–Oriented Type Systems*. John Wiley & Sons Ltd., 1994.

[82] A. Puder. A Declarative Extension of IDL–based Type Definitions within Open Distributed Environments. In *International Conference on Object Oriented Information Systems (OOIS)*, South Bank University, London, 19–21 December 1994. Springer Verlag.

[83] A. Puder. Introduction to the AI–Trader Project. http://www.vsb.informatik.uni–frankfurt.de/projects/aitrader/, Computer Science Department, University of Frankfurt, 1995.

[84] A. Puder and C. Burger. New concepts for qualitative trader cooperation. In A. Schill et al., editors, *International Conference on Distributed Platforms*, pages 301–313. Chapman & Hall, February 1996.

[85] A. Puder and K. Geihs. System support for knowledge–based trading in open service markets. In *7th ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.

[86] A. Puder, F. Gudermann, and S. Markwitz. Ein mehrphasen Protokoll für wissensbasierte Dienstvermittlung. In *Entwicklung und Management verteilter Anwendungssysteme (EMVA)*, Münster, October 1995. Krehl Verlag.

[87] A. Puder, S. Markwitz, and F. Gudermann. Service Trading Using Conceptual Structures. In *3rd International Conference on Conceptual Structures (ICCS'95)*, Santa Cruz, University of California, 14–18 August 1995. Springer Verlag.

[88] A. Puder, S. Markwitz, F. Gudermann, and K. Geihs. AI–based Trading in Open Distributed Environments. In *3rd International IFIP TC6 Conference on Open Distributed Processing (ICODP'95)*, Brisbane, Australia, 20–24 February 1995. Chapman and Hall.

[89] U. Reimer. *Einführung in die Wissensrepräsentation.* B. G. Teubner Stuttgart, 1991.

[90] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the Association for Computing Machinery*, 21:120–126, 1978.

[91] T. Rüdebusch. *CSCW — Generische Unterstützung von Teamarbeit in verteilten DV–Systemen.* Deutscher Universitätsverlag, 1993.

[92] J. Rumbaugh et al. *Object–Oriented Modeling and Design.* Prentice/Hall International, 1991.

[93] M. Santifaller. *Internetworking in a UNIX Environment.* Addison–Wesley Publishing Company, 1991.

[94] J.K. Sheriff. *Charles Peirce's Guess at the Riddle — Grounds for Human Significance.* Indiana University Press, 1994.

[95] J. Siegel. *CORBA: Fundamentals and Programming.* John Wiley & Sons Ltd., 1996.

[96] L. Siegele. Alles in Register. *Die Zeit, Nr. 19*, page 78, May 1996.

[97] H.A. Simon. Why should machines learn? In Michalski, Carbonell and Mitchell, editor, *Machine Lerning: An Artifical Intelligence Approach*, volume 1, pages 25–37, Palo Alto, 1983.

[98] J.F. Sowa. *Conceptual Structures, information processing mind and machine.* Addison–Wesley Publishing Company, 1984.

[99] J.M. Spivey. *The Z Notation — A Reference Manual.* Prentice/Hall International, 1989.

[100] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[101] B. Stroustrup. *Die C++ Programmiersprache.* Addison–Wesley Publishing Company, second edition, 1992.

[102] Sun Microsystems. The Java Language: A White Paper.

[103] A.S. Tanenbaum and R. van Renesse. Distributed Operating Systems. *ACM Computing Surveys*, 17(4):419–470, December 1985.

[104] Microsoft OLE 2.0 Design Team. OLE 2.0 Design Specification. Technical report, Microsoft, ftp://ftp.microsoft.com/developr/drg/ole–info/OLE–2.01–docs/OLE2SPEC.ZIP, April 1993.

[105] R. Tolksdorf. Laura: A coordination language for open distributed systems. In *Proceedings of the 13th IEEE International Conference on Distributed Computing Systems*, ICDCS, pages 39–46, 1993.

[106] H. Vater. *Einführung in die Sprachwissenschaft*. Wilhelm Fink Verlag, München, 1994.

[107] A. Vogel, M. Bearman, and A. Beitz. Enabling Interworking of Traders. In *3rd International IFIP TC6 Conference on Open Distributed Processing (ICODP'95)*, Brisbane, Australia, 20–24 February 1995. Chapman and Hall.

[108] A. Vogel and B. Gray. Translating DCE–IDL in OMG–IDL and vice versa. Technical report, University of Queensland, 1996.

[109] A. Vogel, B. Gray, and K. Duddy. Overcoming the heterogeneity of middleware platforms. Technical report, University of Queensland, 1996.

[110] J.C. Wagner, R.H. Baud, and J.R. Scherrer. Using the Conceptual Graphs Operations for Natural Language Generation in Medicine. In *3rd International Conference on Conceptual Structures (ICCS'95)*, Santa Cruz, University of California, 14–18 August 1995. Springer Verlag.

[111] O. Wanke. Allein im Netz unterwegs. *Süddeutsche Zeitung, Nr. 134*, page 33, June 1996.

[112] P. Wegner and S. B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *ECOOP*. Springer, August 1988.

[113] J. Wielemaker. SWI–Prolog. Technical Report Reference Manual 2.5.0, Dept. of Social Science Informatics (SWI), University of Amsterdam, January 1996.

[114] N. Wirth. *Compilerbau*. B. G. Teubner Stuttgart, 1981.

[115] Yahoo! WebCrawler. http://www.yahoo.com/, Yahoo! Corp., 1996.

[116] Z. Yang and A. Vogel. Achieving interoperability between CORBA and DCE applications using bridges. In A. Schill et al., editors, *International Conference on Distributed Platforms*, pages 144–155. Chapman & Hall, February 1996.

Lebenslauf

Persönliche Daten:

	Arno Puder wohnhaft in Frankfurt am Main geb. am 08.03.66 in Toronto/Kanada

Schulausbildung:

1972–1976	Grundschule in Neunkirchen, NRW
1976–1985	Privates Gymnasium Antonius Kolleg in Neunkirchen, NRW Abitur: Mai 1985

Praktika:

5/1985–8/1985	MDS–Deutschland GmbH

Wehrdienst:

10/1985– 10/1986	Materialamt der Luftwaffe, Köln Porz

Hochschulausbildung:

10/1986–4/1993	Studium der Informatik an der Universität Kaiserslautern Nebenfach: Elektrotechnik
seit 6/1993	Wissenschaftlicher Mitarbeiter an der Johann Wolfgang Goethe–Universität, Frankfurt