Objektorientierte Modellierung von Informationssystemen

Gunter Saake
Informatik, Abteilung Datenbanken
Technische Universität Braunschweig
3300 Braunschweig

Zusammenfassung

Im Bereich des Entwurfs von Informationssystemen spielt die *formale Beschreibung* auf der konzeptionellen Ebene eine wichtige Rolle im Entwurfsprozeß. Etablierte formale Beschreibungsmethoden sind oft nicht geeignet, die inhärente Komplexität interaktiver Informationssysteme mit persistenten Daten in der Modellbildung adäquat zu beherrschen. Als Ergebnis müssen in diesen Methoden Teilbereiche der Systemmodellierung in von einander unabhängigen Formalismen oder gar informell beschrieben werden, so daß keine übergreifende formale Basis der Gesamtbeschreibung bestehen kann.

Der Ansatz der Modellierung von Informationssystemen durch *Objektgesellschaften* versucht dieses Manko zu beheben, indem alle Komponenten eines Informationssystems — persistente Daten, Anwendungsprozesse, Benutzerschnittstellen etc. — einheitlich durch ein einziges Konzept, nämlich durch *Objekte*, beschrieben werden. Dieser Ansatz wird in der vorliegenden Arbeit anhand der Spezifikationssprachen **Oblog**$^{+}$ und TROLL verfolgt, die eine abstrakte Spezifikation eines Systems als Ansammlung kommunizierender Objekte ermöglicht.

Die objektorientierte Spezifikation bietet einen formalen Rahmen, um Objekte als eingekapselte Einheiten von Struktur und Verhalten als Basiseinheiten einer Systemmodellierung zu beschreiben. Die Spezifikation einzelner Objekte basiert auf dem Einsatz temporaler Logik und Prozeßbeschreibungssprachen. Als semantische Interpretationsstrukturen für Objekte können lineare Prozesse mit Beobachtungsstrukturen eingesetzt werden.

Objekte können in vielfältiger Beziehung zueinander stehen; sie kommunizieren miteinander, sie erben Eigenschaften von anderen Objekten, sie sind Klassen zugeordnet, sie setzen sich aus anderen Objekten zusammen. Der in dieser Arbeit vorgestellte Spezifikationsansatz überträgt die Strukturierungskonzepte semantischer Datenmodelle und Wissensrepräsentationsformalismen zur Beschreibung derartiger Beziehungen auf das vorgestellte formale Objektkonzept. Als semantische Grundlage dient jeweils die sichere Objektinklusion. Aufbauend auf diesen Konzepten können komplexe Systeme als *Objektgesellschaften* beschrieben werden.

Vorbemerkungen

Das vorliegende Skript basiert auf zwei Vorlesungen gleichen Titels, die der Autor im Wintersemester 1990/91 an der Universität Dortmund und im Sommersemester 1992 an der TU Braunschweig gehalten hat.

Die Präsentation im vorliegenden Skript hält sich in der Regel eng an die Vorlesungen, obwohl eventuelle Abweichungen in der Kapitelaufteilung, der Durchnumerierung oder einzelner Notationen auftreten können. Einzelne Grafiken wurden weggelassen, falls sie nicht in einem LaTeX-verträglichen Format vorlagen. Einige in der Vorlesung nur angerissene Themenbereiche — z.B. Semantik von Klassen und Klassentypen oder heterogene Objektklassen — sind in der vorliegenden Fassung vertieft worden. Diese Abschnitte sind im Inhaltsverzeichnis und in den Überschriften mit dem Symbol $*$ markiert und gehören in der Regel nicht zum Prüfungsstoff für die beiden oben erwähnten Vorlesungen.

Die Literaturangaben sind für ein Vorlesungsskript oft sehr ausführlich, um einen Überblick über verwandte Methodiken und Ansätze zu geben. Am Schluß einiger Kapitel wird darum eine kurze kommentierte Auflistung der für eine vertiefende Lektüre geeigneten Werke angegeben. Hierbei wurden bevorzugt auch deutschsprachige Lehrbücher berücksichtigt, sofern entsprechende Werke vorlagen.

Konzepte und Sprachvorschläge der vorliegenden Arbeit wurden in Zusammenarbeit mit Thorsten Hartmann, Ralf Jungclaus und Cristina Sernadas entwickelt. Die Diskussionen mit F. Costa, H.-D. Ehrich, J. Fiadeiro, G. Kappel, R. Kutsche, A. Sernadas und vielen ungenannten weiteren Diskussionspartnern haben die hier präsentierte Sicht von Informationssystemen als Objektgesellschaften stark beeinflußt. Mein besonderer Dank gilt Andreas Berner, Thorsten Hartmann, Ralf Jungclaus und den Hörern der betreffenden Vorlesungen für hilfreiche Kommentare und Bemerkungen zu verschiedenen Versionen der vorliegenden Arbeit.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Thema des vorliegenden Skripts ist die formale Beschreibung von Informationssystemen mit objektorientierten Methoden. Während formale Beschreibungsmethoden im Bereich der klassischen Programmspezifikation und auch der Kommunikationsprotokolle inzwischen etablierte Verfahren sind, dominieren momentan im Bereich der Modellierung von Informationssystemen eher informelle Verfahren. Werden in diesem Gebiet tatsächlich formale Methoden eingesetzt, so beinhalten diese oft nur einen kleinen Teilaspekt des gesamten Systems, etwa die Modellierung der gespeicherten Information in Form eines Schemas eines Datenmodells.

In den meisten Modellierungsverfahren für Informationssysteme wird die Informationsstruktur und die Systemdynamik in unabhängigen (und oft unverträglichen) Formalismen beschrieben, so daß keine einheitliche formale Beschreibung des Gesamtsystems vorliegt. Ohne eine einheitlich durchgängige Modellierung des gesamten Systems können aber keine Korrektheitsaussagen über alle Aspekte des Informationssystems getroffen werden. Ein Ansatz für eine derartige integrierte formale Beschreibung von Informationssystemen ist Inhalt dieser Arbeit.

Zur Motivation der folgenden Ausführungen ist es sicher sinnvoll, zuerst die Begriffe *Informationssystem*, *formale Spezifikation* und *objektorientierte Beschreibung* zu charakterisieren. Auch wenn diese Begriffe etablierte Termini im Gebiet der praktischen Informatik sind, ist ihr Gebrauch doch oft sehr unterschiedlich. Ziel dieser einführenden thematischen Aufarbeitungen ist es, das Umfeld der vorliegenden Arbeit festzulegen und die nötigen Begriffsbildungen vorzubereiten.

Im Anschluß an die motivierende Einführung folgt ein kurzer Überblick über die Hauptkapitel der Arbeit. Jedes Kapitel wird mit einer kurzen Literaturübersicht beendet, die Einstiegspunkte in die weiterführende Literatur bietet.

1.1 Charakterisierung des Begriffs Informationssystem

Als Einstieg in die Begriffsbildung ist es sinnvoll, auf etablierte Nachschlagewerke über Begriffsbildungen der Informatik zurückzugreifen, etwa auf den Schülerduden [BI86]. Die klassische Charakterisierung eines Informationssystems ist dort die eines "*Systems zur Speicherung, Wiedergewinnung, Verknüpfung und Auswertung von Informationen*" [BI86]. Als Bestandteile eines Informationssystems werden in [BI86] eine Datenverarbeitungsanlage, ein Datenbanksystem und Auswertungsprogramme genannt. Dieser Charakterisierung nach besteht ein Informationssystem somit im wesentlichen aus einer Datenbank (den gespeicherten Informationen) und Anwendungsprogrammen.

Neuere Ansätze zur Modellierung von Informationssystemen versuchen allerdings von dieser eher technisch orientierten Sicht zu abstrahieren und den Begriff "Informationssystem" auf der konzeptionellen Ebene zu charakterisieren. Der dynamische Aspekt der Berechnungsabläufe in einem Informationssystem wird ebenfalls mehr in den Mittelpunkt gestellt (vergleiche die Artikel [SFSE89] und die Tagungsbände [SvH91, SC92]). Als wesentliche (allerdings teilweise optionale) Funktionalitäten eines Informationssystems können dieser Sicht folgend folgende Punkte genannt werden :

1. Persistente, d.h. dauerhafte, Speicherung von Informationen.
2. Wiedergewinnung der gespeicherten Informationen nach beliebigen Anfragekriterien.
3. Anwendungsspezifische Auswertung und Aufbereitung der gespeicherten Informationen.
4. Korrekte (d.h., integritätsbewahrende) Aktualisierung der gehaltenen Informationen (in Form von Änderungstransaktionen).
5. Integration weiterer Informationsquellen, so z.B.
    - aktiver Systemkomponenten wie Systemuhr / Kalender,
    - Informationszugriff über Netze (Electronic Mail, öffentliche Datenbanken, etc.),
    - Sensoren (z.B. in CIM-Anwendungen),
    - Zugriff auf beliebige andere informationsverarbeitende Systeme (eventuell inklusive Konsultation menschlicher Benutzer !).
6. Modellierung von Benutzerschnittstellen und Benutzerführung.
7. Verteilungsaspekte und Aspekte autonomer Systeme.

Die beiden ersten Punkte definieren die Funktionalität eines Datenbankverwaltungssystems, während die folgenden Punkte 3 und 4 Funktionalitäten sind, die in der Regel nicht einem allgemeinen Datenverwaltungssystem sondern spezifischen Datenbank-Anwendungsprogrammen zugeordnet werden. Die letzten Punkte (5 bis 7) gehen über die Funktionalitäten hinaus, die üblicherweise Datenbank-Anwendungen zugeordnet werden. Diese neuen Funktionalitäten von Informationssystemen erfordern ebenfalls neue Konzepte der Systemmodellierung.

1.2 Formale Spezifikation in der Softwareerstellung

Die Rolle der formalen Spezifikation für die Erstellung von Softwaresystemen kann am besten anhand eines modifizierten Lebenslaufmodells dargestellt werden. Derartige Phasenmodelle spiegeln zwar in der Regel nicht unbedingt den realen Erstellungsprozeß derartiger Systeme wider, sind aber durchaus gut geeignet die Beziehungen zwischen verschiedenen Entwurfsdokumenten zu klären. Ein derartiges Lebenslaufmodell wird in Abb. 1.1 skizziert.

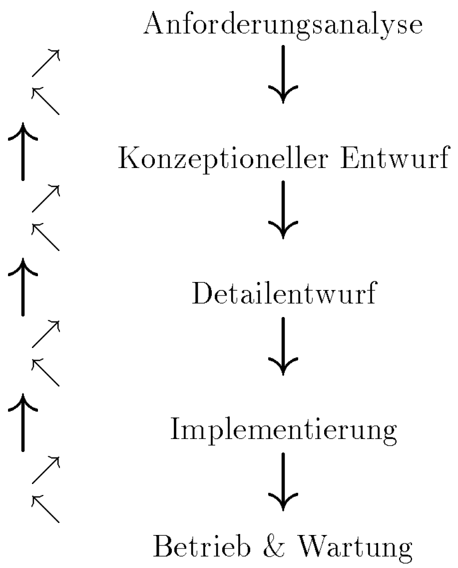

Abbildung 1.1: Lebenslaufmodell für Software-Erstellung

Die einzelnen Phasen dieses Software-Lebenszyklus lassen sich wie folgt kurz charakterisieren :

- Die *Anforderungsanalyse* ist die größtenteils informelle Phase der Problemdefinition, d.h., die Phase, in der in Zusammenarbeit mit den späteren Systembenutzern die gewünschte Systemfunktionalität festgelegt wird.

- Der *konzeptionelle Entwurf* ist die formale Beschreibung des zu realisierenden Systems auf einem hohen Abstraktionsniveau. Das Ergebnis dieser Phase sollte möglichst vollständig in einer *formalen Spezifikation* aller Systemfunktionen und Parameter beschrieben werden, damit diese Spezifikation (oft *konzeptionelles Modell* genannt) die verifizierbare Grundlage des Vertrags zwischen Anwendern und Systemerstellern bilden kann.

  Das konzeptionelle Modell legt fest, *was* realisiert werden soll, aber nicht *wie* es realisiert wird.

- Der folgende *Detailentwurf* legt die Realisierung der Systemkomponenten fest (das *wie*), aber abstrahiert von konkreten Implementierungssprachen und Basissystemen. Im Datenbankentwurf entspricht diese Phase dem *logischen Entwurf*, in dem die Datenbank in dem zur Implementierung verwendeten Datenmodell beschrieben wird, aber keine systemspezifische Datenmodellierungssprache benutzt oder konkrete Zugriffsmethoden festgelegt werden.

- Als letzte Entwurfsphase folgt die konkrete *Implementierung*, gefolgt von

- dem *Betrieb* und der *Wartung* des Systems.

Spätestens bei der Inbetriebnahme des Systems wird deutlich, daß diese Entwurfsphasen keine strikte Sequenz von Dokumenten oder Entwurfsphasen sein können, weil "spätere", d.h. konkretere, Phasen immer wieder zu Rückkopplungen führen, die Teile der früheren Entwurfsdokumente verändern. Diese Änderungen pflanzen sich dann wiederum in die folgenden Phasen hinein fort. Diese Änderungen basieren in der Regel auf erst in späteren Entwicklungsphasen erkannten, nachträglich modifizierten oder zusätzlichen Systemfunktionen. Diese Rückkopplungen sind in Abb. 1.1 durch die nach oben gerichteten Pfeile angedeutet.

Anhand des skizzierten Software-Lebenslaufs, der auch für Informationssysteme gilt, wird die besondere Rolle der Spezifikation im konzeptionellen Entwurfs deutlich : Die formale Spezifikation bildet den Übergang von der Problemsicht der Anforderungsanalyse hin zur Systemsicht der späteren Phasen. Dieser Übergang geht einher mit einem Übergang von Konzepten der Anwender hin zu Konzepten der Systementwickler — und dieser Übergang muß für beide Seiten verständlich und nachvollziehbar sein. Der konzeptionelle Entwurf ist die früheste Entwurfsphase, bei der durchgängig formale Methoden zur Qualitätssicherung eingesetzt werden können. Diese kurzen Bemerkungen deuten auf die zentrale Bedeutung des konzeptionellen Modells hin — Fehler und mangelnde Qualität in dieser Entwurfsstufe pflanzen sich unausweichlich in spätere Phasen fort.

1.3 Überblick über Spezifikationsansätze

Die vorliegende Schrift behandelt in erster Linie den *konzeptionellen Entwurf* von Informationssystemen, d.h. folgend der vorherigen Motivation die Phase der formalen Spezifikation. Die Aufgabe der *formalen Spezifikation* ist die abstrakte Beschreibung von (Software-) Komponenten in einem exakten Formalismus, da nur ein derartiger Formalismus die Verifikation von Systemeigenschaften ermöglicht. Eine Spezifikationssprache muß darum eine formale Semantik haben, die dem benutzten Abstraktionsniveau entspricht. Im folgenden wird der Begriff 'Spezifikation' in diesem strengen Sinne verwendet, d.h. als die formal fundierte abstrakte Modellierung eines Systems.

Im folgenden werden einige etablierte Spezifikationsansätze kurz skizziert und ihre Grundlagen und Eigenschaften diskutiert. Eher informale Entwurfsmethoden werden nicht berücksichtigt. Ein Überblick über den Einsatz von Spezifikationsmethoden beim Entwurf komplexer Systeme kann in [CHJ86] gefunden werden. Dort werden auch mehrere konkrete Spezifikationssprachen und Methoden vorgestellt. Der Artikel von J. Wing [Win90] gibt einen guten Überblick über den Einsatz formaler Spezifikationsmethoden. Das Buch von W. Turski und T. S. E. Maibaum [TM87] präsentiert eine gelungene Aufarbeitung der Konzepte der formalen Spezifikation von Softwaresystemen.

Der folgende Überblick beschränkt sich im wesentlichen auf eine Charakterisierung der zugrundeliegenden *mathematischen Modellbildungen*, die zur Definition der formalen Semantik von Spezifikationen benutzt werden. Insbesondere werden weder konkrete Sprachen noch Spezifikationstechniken behandelt. Für Informationssysteme wichtige Ansätze wie semantische Datenmodelle wurden in die Auflistung integriert, auch wenn diese oft nicht zu den formalen Spezifikationsmethoden gezählt werden, da semantische Grundlagen erst in letzter Zeit intensiv untersucht wurden [GH91].

1.3.1 Spezifikation von Funktionen

Funktionen sind Abbildungen zwischen Wertebereichsmengen. Die Theorie der Funktionen ist ein etabliertes Gebiet der Mathematik, und Funktionen sind ein geeignetes mathematisches Modell für terminierende Programme ohne Interaktion, die Eingabeparameter in Ausgaben überführen. Bei der Spezifikation von Funktionen werden in der Regel elementare Wertebereiche, z.B. die ganzen Zahlen, als gegeben vorausgesetzt. Viele etablierte Ansätze zur formalen Spezifikation basieren auf Funktionen über komplexen Wertebereichen, etwa modellbasierte Spezifikationsmethoden wie Z [Spi89] oder VDM [Jon86].

Spezifikationsformalismen für Funktionen sind z.B. Spezifikation mittels Vor- und Nachbedingungen oder die Definition von rekursiven Funktionen mittels Gleichungssystemen.

Da Informationssysteme interaktive und (zumindestens als idealisiertes Modell) nichtterminierende Systeme sind, wird die alleinige Spezifikation von Funktionen als Beschreibungsmittel hier nicht weiter betrachtet. Das allgemeine Konzept von Funktionen als formalem Konstrukt ist allerdings auch in den folgenden komplexeren abstrakten Modellbildungen enthalten.

1.3.2 Abstrakte Datentypen

Die Idee *abstrakter Datentypen* ist es, abstrakte Wertebereiche durch ihnen zugehörige Funktionen einzukapseln. Ein typisches Beispiel für einen derartigen Datentyp sind die natürlichen Zahlen zusammen mit den Operationen Addition und Multiplikation und Prädikaten wie 'kleiner gleich'. Dieser Ansatz ermöglicht es, auch komplexe Wertebereiche (z.B. Graphen oder Warteschlangen) auf einer abstrakten Ebene zu modellieren, ohne auf die konkrete Realisierung der Datenstrukturen eingehen zu müssen. Die zugrundeliegende mathematische Modellbildung sind Algebren, d.h. Wertemengen mit Funktionen.

Ein etablierter Formalismus zur formalen Spezifikation von abstrakten Datentypen ist die algebraische Gleichungsspezifikation und deren Erweiterungen [EM85, EGL89]. Hierbei werden die Funktionen durch Angabe von Gleichungen beschrieben.

Für die Modellierung von Informationssystemen sind abstrakte Datentypen als alleiniger Spezifikationsmechanismus nicht geeignet, da Interaktivität und Persistenz dem Ansatz prinzipiell fremd sind, auch wenn die Kodierung dieser Konzepte durch komplexe Datenstrukturen und verborgene Zustandssorten möglich ist. Zur angemessenen abstrakten Modellierung der Basisdatenstrukturen eines Informationssystem sind abstrakte Datentypen allerdings unverzichtbar.

1.3.3 Allgemeine Prädikatenlogik und Erweiterungen

Die *Prädikatenlogik* ist die Grundlage vieler Spezifikationstechniken, die aber in der Regel syntaktische und semantische Einschränkungen festlegen. Eine Spezifikation in allgemeiner Prädikatenlogik besteht aus einer Reihe von logischen Axiomen, die das zu spezifizierende System erfüllen soll. Die bereits vorgestellten Spezifikationen von Funktionen und abstrakten Datentypen mittels Gleichungsaxiomen sind hierbei als Spezialfälle enthalten. Einige Autoren zählen auch Logikprogrammierung zu diesen eingeschränkten Spezifikationsmechanismen.

Interpretationsstrukturen für prädikatenlogische Spezifikationen sind wiederum Algebren (also Werte und Funktionen), auf denen zusätzlich Prädikate definiert sind.

Die Allgemeinheit von beliebigen prädikatenlogischen Spezifikationen ist wohl der wichtigste Nachteil dieses Ansatzes, der z.B. Konsistenzüberprüfungen für Spezifikationen sehr erschwert. Für den praktischen Einsatz kann darum beim heutigen Stand der Technik auf syntaktische Einschränkungen nicht verzichtet werden.

Von Interesse für die Spezifikation von Informationssystemen sind insbesondere Erweiterungen der Prädikatenlogik um temporale und modale Operatoren, die eine abstrakte Beschreibung der zeitlichen Entwicklung des Systems ermöglichen [Ser80, Krö87, FS88, Saa91b, Eme90]. Interpretationsstrukturen sind dann Folgen bzw. gerichtete Graphen von einfachen Interpretationsstrukturen für die basierende Prädikatenlogik. Auf diese *temporalen* bzw. *modale Logiken* wird in den folgenden Abschnitten noch genauer eingegangen werden.

1.3.4 Datenmodelle für persistente Datenbestände

Im Bereich der Modellierung von Datenbanken werden sogenannte *semantische Datenmodelle* [HK87] eingesetzt, um persistente Datenbestände zu modellieren. Wir subsumieren hier unter dem Begriff 'semantische Datenmodelle' im Gegensatz zu einigen Autoren alle Datenmodelle, die von konkreten implementierten Datenmodellen abstrahieren und die semantisch relevante Modellierungskonzepte wie Klassenbildung, Spezialisierungen, komplexe Objekte und Objektbeziehungen unterstützen.

Auf die in Datenmodellen eingesetzten Konzepte wird in Abschnitt 2.3.2 ausführlicher eingegangen. Ein Schema in einem semantischen Datenmodell kann als eine Menge von prädikatenlogischen Axiomen aufgefaßt werden, beschreibt aber zusätzlich auch zeitabhängige Eigenschaften wie Persistenz oder Schlüsseldefinitionen zur zeitpunktunabhängigen Identifikation. Semantische Modellbildung für einzelne Datenbankzustände sind ebenfalls Interpretationsstrukturen für eine mehrsortige Prädikatenlogik, während die Eigenschaft der Persistenz wie bei temporalen Logiken nur anhand von Zustandsfolgen modelliert werden kann.

Semantische Datenmodelle sind, wie der Name bereits andeutet, gut geeignet um große Datenbestände strukturiert zu modellieren. Für Informationssysteme ebenfalls relevante Aspekte wie Anwendungsfunktionen und Interaktivität können in ihnen nur in einem zusätzlichen Formalismus modelliert werden.

1.3.5 Prozeßspezifikation

Bei den bisherigen Ansätzen wurde jeweils die fehlende Unterstützung der Modellierung von Interaktivität kritisiert. Dieser Aspekt steht (unter anderem) im Vordergrund bei der Modellierung *kommunizierender Prozesse* [Mil89, Hen88].

Ein Teil der Beschreibungsmethoden für kommunizierende Prozesse basiert auf der Charakterisierung der Prozeßabläufe mittels logischer oder algebraischer Axiomatisierung. Die derartige Spezifikation von Prozessen geschieht mit algebraischen Ansätzen (z.B. CSP [Hoa85]) bzw. mit Prozeßkalkülen (z.B. CCS [Mil80]). Hierbei werden Prozesse semantisch als abstrakte Werte modelliert.

Der Petrinetzansatz und verwandte Methoden hingegen [Rei85] beschreiben parallele Abläufe direkt, indem Netze aus unabhängigen Transitionen definiert werden, die durch die Weitergabe von 'Token' miteinander kommunizieren.

In den etablierten Prozeßbeschreibungsformalismen wird die Datenstrukturierung nicht besonders unterstützt, da der Schwerpunkt dieser Formalismen auf der Modellierung des Kommunikationsverhaltens liegt. Da die abstrakte Modellierung von Informationsstrukturen zu den Grundaufgaben des konzeptionellen Entwurfs von Informationssystemen gehört, sind die etablierten Prozeßspezifikationsmethoden als alleiniger Beschreibungsformalismus ebenfalls dafür nicht geeignet.

1.4 Das objektorientierte Entwurfsparadigma

Das objektorientierte Paradigma wurde erfolgreich in viele Gebiete der Informatik integriert, so im Bereich der Programmiersprachen, der Datenmodelle oder des Software Engineerings [KA90, BB92]. In diesen unterschiedlichen Forschungsrichtungen herrschen durchaus unterschiedliche Ansichten darüber, was das Adjektiv "objektorientiert" denn nun eigentlich auszeichnet.

In dieser Arbeit soll versucht werden, den Begriff des "Objekts" als Einheit des formalen Entwurfs komplexer Systeme zu charakterisieren und zu formalisieren. Dafür muß von der Implementierungsnähe und operationalen Sichtweise objektorientierter Programmiersprachen genauso abstrahiert werden wie von der primären Konzentration auf Datenstrukturierung und effizienten Zugriff auf gespeicherte Objekte, wie sie in vielen objektorientierten Datenmodellen gefunden werden kann.

Das *Objekt als Basiskonzept des Entwurfs* kann wie folgt beschrieben werden :

- Objekte sind die Basiseinheiten von *Struktur und Verhalten*, aus denen sich das zu beschreibende System zusammensetzt [SFSE89].

- Objekte haben beobachtbare Eigenschaften, die sogenannten *Attribute* der Objekte.

- Objekte haben einen *eingekapselten internen Zustand*, auf dem die Attribute einen eingeschränkten Lesezugriff erlauben.

- Der interne Zustand eines Objekts wird ausschließlich durch das Eintreten *objektspezifischer Ereignisse* manipuliert. In objektorientierten Programmiersprachen entspricht dies dem Aufruf objektspezifischer Methoden.

- Objekte können *persistent* im Sinne der Datenbankterminologie sein, d.h. der interne Zustand von Objekten überlebt einzelne Systemläufe.

In der Sprechweise der Programiersprachenwelt entspricht dieses Objektkonzept einem Modul mit persistenten Variablen (der 'Struktur'), auf die nur über eine funktionale Schnittstelle manipulierend zugegriffen werden kann. Das hier vorgestellte Basiskonzept vereinigt hierbei Konzepte objektorientierter Programmiersprachen mit für

Informationssysteme wichtigen Konzepten, die sich bei der Modellierung persistenter Datenbanken als sinnvoll erwiesen haben.

Neben diesem grundlegenden Basiskonzepte der (einzelstehenden) Objekte sind weitere Modellierungskonzepte notwendig, die es erlauben aus einzelnen Objekten Ansammlungen miteinander in Beziehung stehender Objekte zusammenzusetzen. Derartige Ansammlungen werden im folgenden auch als *Objektgesellschaften* bezeichnet.

- Objekte *kommunizieren* miteinander, indem sie Ereignisse bzw. Methoden anderer Objekte aufrufen.
- Objekte können zu *Objektklassen* gruppiert sein, deren Exemplare einem bestimmten *Typ* angehören.
- Zwischen Objekten können vielfältige *Beziehungen* bestehen.
- Objekte und Objektklassen können in *Spezialisierungshierarchien mit Vererbung* eingeordnet sein.
- Objekte können *aktiv* oder *passiv* sein, je nach dem ob es ihnen in ihrem Kontext erlaubt ist, ein Ereignis aus eigener Initiative einzuleiten.
- Mehrere Objekte können zu *komplexen Objekten* zusammengesetzt werden.

Objektgesellschaften bestehen somit aus einer Ansammlung vieler, eventuell komplex strukturierter, persistenter Objekte, die möglicherweise aktiv miteinander kommunizieren. Wir schließen diesen Abschnitt mit einer Behauptung, die in den folgenden Ausführungen noch eingehender zu begründen sein wird :

Objekte sind mit den etablierten Entwurfsparadigmen nicht adäquat beschreibbar !

1.5 Gliederung der Arbeit

Das folgende **Kapitel 2** behandelt den konzeptionellen Entwurf von Informationssystemen. Es werden allgemeine Anforderungen an den konzeptionellen Entwurf diskutiert und diese Anforderungen den etablierten Spezifikationsformalismen gegenübergestellt. Ein Abschnitt ist dem mehrschichtigen konzeptionellen Entwurf, der ein Informationssystem in Schichten um eine zentrale Datenbank herum modelliert, gewidmet. Nach einer allgemeinen Kritik dieses Ansatzes folgt eine Diskussion des Bezugs des klassischen Entwurfsansatzes zu objektorientierten Entwurfsprinzipien.

Das folgende **Kapitel 3** behandelt die formale Beschreibung von einzelnen Objekten als Basiseinheit eines konzeptionellen Entwurfs. Objektbeschreibungen in einer objektorientierten Spezifikationssprache, einer Version der Sprache TROLL [JSHS91], werden anhand Beispielen vorgestellt. Ein Abschnitt beschäftigt sich mit semantischen Modellbildungen für Objekte, ein weiterer mit den verschiedenen Spezifikationslogiken die bei der Objektbeschreibung eingesetzt werden. Die Umsetzung von Objektbeschreibungen in Formeln einer temporalen Logik ermöglicht die Semantikfestlegung der Spezifikationskonstrukte von TROLL. Das Kapitel schließt mit der Erweiterung der vorgestellten Konzepte von einzelnen Objekten auf Objektklassen.

Kapitel 4 behandelt die verschiedenen Beziehungen, die zwischen Objekten und Objektklassen auftreten können. Als semantische Grundlagen behandeln zwei Abschnitte die Kommunikation zwischen Objekten mittels Ereignisaufruf und die Einbettung von Objekten in andere Objekte als Grundlage von komplexen Objekten. Ein Abschnitt diskutiert Sprachmittel und Bedeutung von Objekthierarchien mit Vererbungsbeziehungen, wobei zwischen Spezialisierungen und temporären Rollen unterschieden wird. Es folgen drei Abschnitte über die verschiedenen Arten komplexer Objekte, heterogener Objektklassen und ungerichteter Beziehungen zwischen Objekten. Der abschließende Abschnitt behandelt den Aufbau von *Systemen* aus Objekten, die hier Objektgesellschaften genannt werden.

Literaturhinweise

Die Begriffsbildung für Informationssysteme kann in verschiedenen Lehrbüchern gefunden werden [EN89]. Unterschiedliche Versionen von Software-Lebenslaufmodellen werden in Büchern zum Thema Software Engineering behandelt [Den91, GJM91], die entsprechenden Modelle für Datenbankanwendungen sind u.a. in [EN89, Kapitel 16] behandelt.

Formale Spezifikation wird ausführlich in dem Buch [CHJ86] anhand konkreter Sprachen und Fallbeispiel diskutiert. J. Wing's Artikel [Win90] präsentiert eine gute Einführung in diese Thematik. Das Buch von W. M. Turski und T. S. E. Maibaum präsentiert die dem Begriff der formalen Spezifikation zugeordneten Konzepte in Form einer gründlichen Aufarbeitung [TM87].

Das objektorientierte Paradigma wird in den folgenden Kapiteln noch ausführlich diskutiert. Dort werden auch weiterführende Literaturhinweise gegeben. Einen guten informellen Einstieg in die objektorientierte Terminologie und Basiskonzepte in verschiedenen Anwendungsbereichen bietet das Buch von S. Khoshafian und R. Abnous [KA90]. Eine Diskussion grundlegender formaler Konzepte des objektorientierten Ansatzes präsentiert P. Wegner in dem Artikel [Weg90]. Eine deutschsprachige Übersicht gibt [BB92].

Kapitel 2

Konzeptioneller Entwurf

Inhalt dieses Kapitels sind die Möglichkeiten, konventionelle Entwurfsmethoden für die konzeptionelle Modellierung von Informationssystemen einzusetzen. Nach einer Analyse der Anforderungen an eine Entwurfsmethode wird die Eignung etablierter Spezifikations- und Modellierungstechniken kritisch analysiert. Aufbauend auf dieser Kritik, wird ein mehrschichtiger Ansatz zur Spezifikation von Informationssystemen vorgestellt, der den kombinierten Einsatz etablierter Techniken vorschlägt.

Auch dieser kombinierte Einsatz erweist sich als weitgehend ungeeignet für die Handhabung großer Entwurfsdokumente. Aus dieser Erkenntnis heraus wird zum Schluß des Kapitels die Entwicklung objektorientierter Methoden für den konzeptionellen Entwurf motiviert.

2.1 Anforderungen

In dem Entwurfszyklus von Informationssystemen ist der konzeptionelle Entwurf die erste Phase, die direkt von den Systemerstellern durchgeführt wird [EN89]. In der vorherigen Phase, der Anforderungsanalyse, werden die Anforderungen an ein Anwendungssystem in einer den Anwendern vertrauten, oft informalen Beschreibungssprache festgelegt. Der konzeptionelle Entwurf kann somit als eine Art "Vertrag" zwischen den Anwendern und den Systemerstellern aufgefaßt werden. Aus dieser besonderen Rolle folgen einige, auf den ersten Blick widersprüchliche Anforderungen an einen Beschreibungsformalismus für den konzeptionellen Entwurf:

- Die Grundlage eines überprüfbaren Vertrages muß eine "beweisbare" Systembeschreibung sein. Der Entwurfsformalismus muß somit auf einer formalen Modellbildung basieren, um eine formale Verifikation der späteren Implementierung sowie Konsistenzprüfungen zu ermöglichen.

- Die konzeptionelle Beschreibung sollte andererseits möglichst nahe an der Terminologie und den Konzepten der Anwender sein. Stichworte hierfür sind grafische Aufbereitung formaler Spezifikationen sowie die Möglichkeit der frühen Erstellung von Prototypen.

Weitere Forderungen an einen Beschreibungsformalismus sind u.a. Möglichkeiten der Strukturierung und Modularisierung insbesondere für größere Entwürfe sowie die Wahl von sauberen und angemessenen Sprachkonzepte, um die Fehlerrate im Entwurfsprozeß bereits in diesem frühen Entwurfsstadium zu senken.

Der klassische Datenbankentwurf beschäftigt sich hauptsächlich mit dem Entwurf der eigentlichen Datenbank, d.h. mit den zu speichernden Strukturen (z.B. Relationen) und mit Integritätsbedingungen, die von korrekten Datenbankausprägungen erfüllt werden müssen (z.B. funktionale Abhängigkeiten). Der Entwurf der zu einem Informationssystem gehörenden Anwendungsprogramme muß dabei in einem unabhängigen weiteren Formalismus erfolgen.

Um ein Informationssystem zu entwerfen, müssen sehr viele, auf dem ersten Blick unzusammenhängende Teile beschrieben werden. Die *zentrale Datenbank* wird üblicherweise durch ein Datenbankschema beschrieben, das die eigentliche Strukturbeschreibung sowie eine Menge von Integritätsbedingungen für Datenbankzustände beinhaltet. Oft wird eine Menge von vorgegebenen elementaren Datenbankänderungen ebenfalls zum Datenbankschema dazugezählt [Lip89]. Auf der Ebene des konzeptionellen Entwurfs beschreibt ein Datenbankschema die Funktionalität der Datenhaltungskomponente. Weitere wichtige Informationen für spätere Entwurfsschritte, z.B. typische Anfragen oder Anwendungsprozesse, werden dabei in der Regel nicht berücksichtigt.

Der *Entwurf der Anwendungssoftware* entspricht dem klassischen Softwareentwurf zum Beispiel durch Petrinetze oder Datenflußdiagrammen. Die dort eingesetzten Verfahren unterstützen allerdings in der Regel nicht die Behandlung persistenter Objekte wie sie für Informationssysteme charakteristisch sind. Bei dem Entwurf von Informationssystemen müssen diese beiden Teilbereiche integriert werden.

Ein weiterer zu behandelnder Teilbereich ist die Beschreibung des erlaubten zeitlichen Verhaltens *unabhängig von Anwendungstransaktionen*. Dieser Teilbereich ist insbesondere dann relevant, wenn das Informationssystem Direktänderungen der Datenbank unter Umgehung von Anwendungssoftware zuläßt beziehungsweise wenn nachträgliche Erweiterungen und Änderungen der Anwendungsprogramme zu erwarten sind.

Im folgenden Abschnitt wird ein Vorschlag für einen Beschreibungsformalismus vorgestellt, der die angesprochenen Teilbereiche integriert.

2.2 Eignung bisheriger Spezifikationsmethoden für den Entwurf von Informationssystemen

In Abschnitt 1.3 wurden mehrere etablierte Spezifikationsansätze kurz vorgestellt. In Abbildung 2.1 wird versucht, diese Methoden bezüglich ihrer Eignung für den konzeptionellen Entwurf von Informationssystemen einzuordnen.

Entwurfsmethode :	Funkt.	ADTen	Datenmodelle	temp. Logik	Prozesse
Datenstrukturierung	–	+	+	(+)	–
anwendungsspez. Funktionen	+	+	–	(+)	–
Persistenz	–	–	+	+	(–)
Klassifizierung / Spezialisierung	–	(+)	+	(–)	–
zeitl. Verhalten	–	–	(–)	+	+
Kommunikation / Interaktion	–	–	–	–	+
Aktivität versus Passivität	–	–	–	–	(+)

Abbildung 2.1: Eignung etablierter Spezifikationsmethoden für den konzeptionellen Entwurf von Informationssystemen.

Zu diesem Zwecke wurde eine Aufstellung einiger für den Entwurf von Informationssystemen grundlegender Konzepte aufgestellt — diese Liste umfaßt die Begriffe Datenstrukturierung, anwendungsspezifische Funktionen, Persistenz, Klassifizierung und Spezialisierung / Generalisierung von Informationseinheiten, zeitliches Verhalten, Kommunikation und Interaktion, sowie Aktivität versus Passivität — und den einzelnen Spezifikationsparadigmen gegenübergestellt. Die Eignung der Paradigmen zur Modellierung dieser Konzepte wurde dann mittels '+' bzw. '–' bewertet.

Die vorgenommene Bewertung soll nun im Einzelnen begründet werden.

- Die Spezifikation von Funktionen etwa mit Vor- und Nachbedingungen ist für die meisten aufgeführten Aspekte prinzipiell nicht geeignet, da weder eine angemessene Datenstrukturierung noch eine zeitliche Entwicklung geeignet modelliert werden kann.
- Abstrakte Datentypen (ADTen) modellieren Mengen von *Werten* mit zugehörigen spezifischen Funktionen. Sie sind somit gut geeignet zur Datenabstraktion

und Funktionsspezifikation, solange keine zeitlichen Aspekte wie Persistenz ins Spiel kommen. Die Möglichkeit der Parametrisierung und Untersortenbildung [EM85, EGL89] kann einige Aspekte der Klassifizierung und Spezialisierungshierarchien modellieren, vernachlässigt aber den zustandsabhängigen Aspekt dieser Modellierungskonzepte.

- Die verschiedenen Datenmodelle sind in der Regel sehr gut geeignet zur Strukturierung persistenter Datenbestände — dafür wurden sie ja speziell entwickelt. Konzepte wie Anwendungsfunktionen, Interaktivität etc. sind ihnen jedoch prinzipiell fremd.

- Die einfache Prädikatenlogik wurde nicht extra aufgeführt, da für sie im Prinzip das selbe wie für abstrakte Datentypen gilt. Ihre modalen oder temporalen Erweiterungen sind, da sie diese einfache Prädikatenlogik als Unterkalkül enthalten, im Prinzip auch zur Datenstrukturierung und Funktionsdefinition geeignet, aber ihre Stärken liegen eindeutig in der deskriptiven Beschreibung des zeitlichen Systemverhaltens (welche die Definition von Persistenz als Teilaspekt beinhaltet).

- Prozeßspezifikation ist als einzige der vorgestellten Entwurfsmethodiken in der Lage, Kommunikation und Interaktivität angemessen zu modellieren. Die heutigen Methoden bieten aber in der Regel nur wenig Unterstützung zur Modellierung von Datenstrukturierung und persistenten Informationseinheiten. Die Unterscheidung zwischen Aktivität und Passivität wird oft zugunsten einer deskriptiven Beschreibung des Gesamtsystems fallengelassen.

Ziel einer Entwurfsmethodik, die auf diesen etablierten Ansätzen aufbaut, muß nun eine geeignete Kombination der Stärken der einzelnen Methodiken sein, die in einer integrierten Entwurfsmethodik resultiert.

2.3 Mehrschichtiger konzeptioneller Entwurf

In den vorigen Abschnitten wurde motiviert, daß der Entwurf von Informationssystemen vielerlei, auf den ersten Blick unverträgliche Teilaspekte beinhaltet. Für die formale Semantik und für Konsistenzprüfungen wird eine umfassende und einheitliche Modellbildung benötigt, die alle diese Bestandteile umfaßt. Für einen praktisch einsetzbaren Spezifikationsformalismus andererseits ist es sinnvoll, für all diese Teilaspekte bereits erprobte Beschreibungsmethoden zu verwenden. Diese beiden gegenläufigen Forderungen können in einem geschichteten Ansatz, wie er etwa in [SFNC84, CS87, HNSE87, Saa91a, EGH+90, EGH+92] vorgeschlagen wurde, integriert werden, wie in Abbildung 2.2 skizziert.

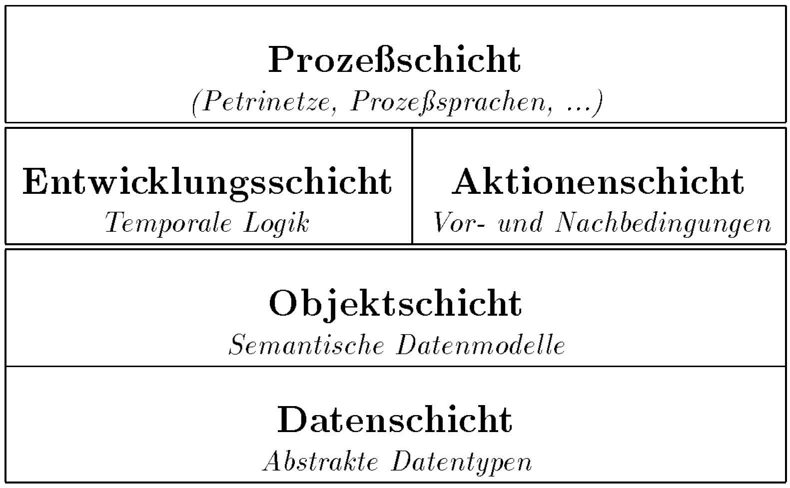

Abbildung 2.2: Schichtenmodell des konzeptionellen Entwurfs.

Die Schichtenarchitektur in Abbildung 2.2 zeigt in gewissem Sinne eine zentralisierte Sicht auf den Aufbau eines konzeptionellen Models. Wenn wir auch Aspekte der logischen Verteilung und Autonomie berücksichten wollen, erhalten wir das in Abbildung 2.3 gezeigte Bild, bei dem — basierend auf einer gemeinsamen Datenschicht als Verständigungsgrundlage — mehrere autonome Datenbanken unter einer Anwendungsschicht zusammengefaßt werden.

Die Pfeile zwischen Prozeß- und Aktionenschicht in Abbildung 2.3 deuten an, daß Manipulationen der Datenbank ausschließlich über die Schnittstelle der Aktionen erfolgen können. Ferner ist in dem Bild berücksichtigt, daß Anwendungsprozesse unabhängig von Datenbanken auf Datentypen Zugriff haben können.

Die Idee des geschichteten Ansatzes ist es, schrittweise eine immer komplexere Modellbildung zu entwickeln, die am Ende ausreichend mächtig ist, um komplexe auch Informationssysteme zu beschreiben. Die einzelnen Schichten eines konzeptionellen Entwurfs werden im folgenden kurz erläutert. Bei der Erläuterung wird jeweils auf die zugrundeliegende semantische Modellbildung und auf die in Frage kommenden Spezifikationsformalismen und -sprachen eingegangen. Die Darstellung lehnt sich an den in [Saa91a] präsentierten Ansatz an.

2.3.1 Datenschicht

Die *Datenschicht* beschreibt den zustandsunabhängigen Anteil der Strukturen eines Informationssystems. Hier werden die Basisdatenstrukturen beschrieben, die der Datenbank und den Anwendungsprozessen gemeinsam sind. Zu beschreibende Elemente sind *zustandsunabhängige Datenstrukturen und zugehörige Operationen*, mit anderen

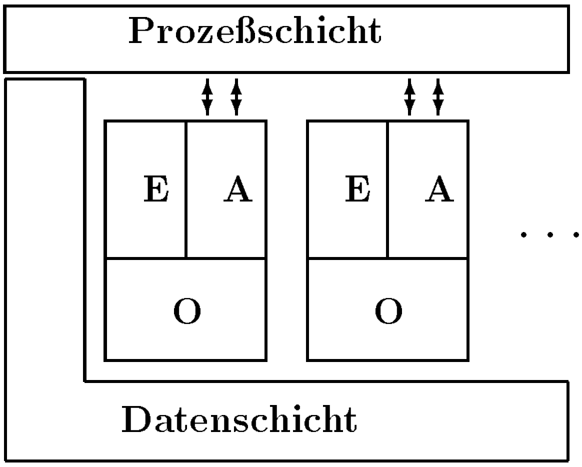

Abbildung 2.3: Modifizierte Schichtenarchitektur.

Worten also *abstrakte Datentypen*.

Die explizite Spezifikation einer Datenschicht ist im klassischen Datenbankentwurf eher unüblich. Die Motivation für eine explizite Spezifikation der Datenschicht ist, daß selbst einfache Anwendungen mit den durch bekannte Datenbankmodelle vorgegebenen elementaren Wertebereichen nicht auskommen. Dies gilt erst recht für die sogenannten Nicht–Standard–Anwendungen zum Beispiel im Ingenieurbereich [KL88]. Neben den vorgegebenen Datentypen wie `integer` und `string` müssen somit zusätzlich anwendungsspezifische Datentypen spezifiziert werden können.

Typische Beispiele für im Informationssystementwurf zu spezifizierende Datentypen sind

- geometrische und graphische Datentypen wie Punkt, Linienzug, Polygon, etc.
- Aufzählungstypen wie Farbe, Maßeinheit, etc.
- ingenieurwissenschaftlich relevante Datentypen wie Matrix, Vektor, etc.
- insbesondere für Multimediaanwendungen und verwandte Gebiete wichtige Datentypen wie Bitmap, Bildsequenz, Text, etc., die im klassischen Datenbankentwurf nur als uninterpretierte 'Byte-Container' modelliert werden können.

Zur Beschreibung von Datentypen wird die algebraische Spezifikation von abstrakten Datentypen eingesetzt [EM85, EGL89]. Einfache Datentypen werden hierbei direkt durch Angabe eines Gleichungssystems beschrieben, während zur Konstruktion komplexerer Strukturen *parametrisierte Datentypkonstruktoren* wie `list` oder `set` eingesetzt werden können. Die Modellbildung für die algebraische Spezifikation sind Algebren im mathematischen Sinne, d.h. abstrakte Wertebereiche plus Funktionen auf diesen Wertebereichen.

Beispiel 2.3.1 Der geometrische Datentyp `point` modelliert Punkte in der Zahlenebene. Zusammen mit den zugehörigen Operationen kann er wie folgt spezifiziert werden[1]:

```
DATATYPE point BASED ON real;
SORTS point;
OPERATIONS distance :       (point × point): real;
           xcoord, ycoord : (point): real;
           createpoint :    (real × real): point;
           add :            (point × point): point;
           ...
VARIABLES p,q : point;
          x,y,x1,y1 : real;
EQUATIONS
    x = xcoord(createpoint(x,y));
    y = ycoord(createpoint(x,y));
    distance(createpoint(x,y),createpoint(x1,y1))
      = sqrt((x-x1)*(x-x1) + (y-y1)*(y-y1));
    add(p,q)
      = createpoint(xcoord(p)+xcoord(q),ycoord(p)+ycoord(q));
    ...
```

□

Ein Modell für den Typ `point` in dieser Beispielspezifikation ist die Menge abstrakter Punkte auf der Zahlenebene (isomorph zu `real`×`real`) zusammen mit den angegebenen Operationen. In der Regel gibt es mehrere mathematische Modelle für eine Spezifikation, aus der ein (bis auf Isomorphie eindeutiges) Modell als Semantik gewählt werden muß [EM85, EGL89].

[1] In dieser Arbeit werden neuentwickelte Beispiele mit deutschen Bezeichnungen ausgeführt, während aus der Literatur entnommene Beispiel in der originalen (in der Regel englischen) Notation übernommen wurden. Einige der in diesem Abschnitt aufgeführten Beispiel stammen aus [Saa91a].

2.3.2 Objektschicht

Auf der *Objektschicht*[2] werden die möglichen Zustände einer Datenbank beschrieben. Zu beschreibende Elemente sind die *korrekten Datenbankzustände*, d.h. die möglichen Mengen von *informationstragenden persistenten Objekten*.

Bestandteile der Objektschichtbeschreibung sind insbesondere abstrakte Strukturbeschreibungen in einem Datenmodell sowie die Angabe von Integritätsbedingungen. Für den konzeptionellen Entwurf sollte das verwendete Datenmodell Modellierungsmechanismen wie komplexe Objekte, beliebige Beziehungen zwischen Objekten und Generalisierungshierarchieen mit Vererbung unterstützen. In Frage kommen hierfür eine ganze Reihe von modernen Vorschlägen für konzeptionelle Datenmodelle bzw. Objektmodelle. Beispiele für in Frage kommende Datenmodelle sind

- die sogenannten *semantischen Datenmodelle* [HK87, PM88],
- *(erweiterte) Entity-Relationship-Modelle* [HNSE87, EN89],
- sowie die sich aktuell in der Entwicklung befindlichen *objektorientierten Datenmodelle* [ABD+89, Bee90].

Moderne Vorschläge für konzeptionelle Datenmodelle unterstützen in der Regel bereits eine Vielzahl an Modellierungskonzepten, die eine abstrakte Beschreibung der zu speichernden Informationseinheiten (hier : 'Objekte') und der Beziehungen zwischen diesen ermöglichen. Einen Überblick über einige sinnvolle Modellierungskonzepte gibt die folgende Auflistung.

- Das erste Modellierungsprimitiv ist das Konzept eines abstrakten *Objekts*. Objekte modellieren die gespeicherten Informationseinheiten und sind Abstraktionen der in dem Anwendungsgebiet auftretenden 'Dinge'. Objekte sind abstrakt in dem Sinne, daß nicht sie selber sondern nur ihre Eigenschaften beobachtbar sind. Eigenschaften von Objekten werden *Attribute* genannt und sind daten- oder objektwertige Funktionen.
- Objekte können zu elementaren *Objekttypen* gruppiert werden, sofern sie gleichartige Eigenschaften aufweisen. Beispiele für Objekttypen sind `PERSON` oder `FIRMA` mit zugehörigen datenwertigen Attributen, z.B. das Attribut `name` vom Typ `string` bzw. das Attribut `location` vom Typ `point`. Ein Beispiel für ein objektwertiges Attribut ist das Attribut `boss` vom Typ `PERSON`, das eine Eigenschaft von Objekten des Typs `FIRMA` beschreibt.

[2] Der Begriff Objektschicht hat sich für diese Schicht etabliert, da auf dieser Schicht die *strukturellen Objekte* der Datenbank modelliert werden. Dieser Objektbegriff hat nur wenig mit dem ansonsten in diesem Skript benutzten Objektbegriff als Einheit von Struktur *und* Verhalten zu tun. In diesem Abschnitt werden die Begriffe 'Objektschicht' und 'Objekt' in diesem rein strukturellen Sinne benutzt, um die Konsistenz zur Notation in der Literatur zu wahren.

- Abstrakte Objekte haben eine *Identität* unabhängig von ihren beobachtbaren Eigenschaften, d.h. unterschiedliche Objekte können gleiche Attributwerte haben. Um diese Objekte unterscheiden zu können, wird ein sogenannter *Objektidentifikationsmechanismus* [KC86] benötigt. Diese Identifikation kann auf der konzeptionellen Ebene explizit durch Angabe von Schlüsselattributen [EDG88, HNSE87] oder implizit als Eigenschaft des Datenmodells [ABD$^+$89, Bee90] festgelegt werden.

- Mit einem Objekttyp ist zu jedem Zeitpunkt die *Klasse* aktuell existierender Ausprägungen, d.h. existierender Objekte dieses Typs, assoziiert. In der Regel wird davon ausgegangen, daß unterschiedliche Klassen disjunkt sind. In vielen Fällen sind aber diese Klassen konzeptionell nicht disjunkt, so z.B. die Klassen `PERSON` und `MANAGER`. In diesem Fall sprechen wir auch von *Klassen- oder Typkonstruktion* [HNSE87]. Derartige konstruierte Klassen erben ihre Identifikation (und je nach Datenmodell eventuell auch ihre Eigenschaften) von den Basisobjekttypen (d.h., den elementaren, nicht konstruierten Typen, auf denen die Konstruktionshierarchie fußt). Die formale Semantik dieser Konstruktionen ist in [AH87, HNSE87, Hoh90] beschrieben.

  Einige für den konzeptionellen Entwurf interessante Typkonstruktionen werden im Folgenden aufgelistet und erläutert.

  - *Spezialisierung* wird benutzt um eine Klassenhierarchie aufzubauen (`is_a`-Beziehungen). Als Beispiel können wir die beiden unabhängigen Unterklassen `PATIENT` und `MANAGER` von `PERSON` definieren. Spezialisierung bedeutet eine Untermengenbeziehung zwischen den aktuellen Klassenausprägungen und, abhängig vom verwendeten Datenmodell, die Vererbung von Eigenschaften des Eingabetyps.
  - *Partitionierung* ist ein Spezialfall der Spezialisierung bei dem in einem Schritt mehrere *disjunkte* Unterklassen konstruiert werden. Ein Beispiel ist die Partitionierung von `PERSON` in `WOMAN` und `MAN`.
  - *Generalisierung* ist in gewissem Sinne der umgekehrte Schritt zur Spezialisierung — mehrere (disjunkte) Eingabeklassen werden zu einer neuen Klasse generalisiert. Ein Beispiel ist die Generalisierung der Objekttypen `PERSON` und `FIRMA` zu juristischen Personen (`LEGAL_PERSON`).

- Ein anderes Modellierungskonzept bekannt durch die Entity-Relationship-Ansätze [Che76] ist die Definition allgemeiner *Beziehungen* zwischen Objekten, z.B. die Beziehung `Works_For` zwischen Personen und Firmen. Viele Beziehungen (so die bereits erwähnten `is_a`-Beziehungen) haben eine spezielle Bedeutung und werden darum in einigen Datenmodellen explizit gekennzeichnet und bei Konsistenztests und Transformationen gesondert behandelt.

- Eine weitere Beziehung zwischen Objekten, die gesondert behandelt werden sollte, ist die Zusammensetzung *komplexer Objekte* aus anderen Objekten. Dieses Modellierungskonzept hat sich insbesondere in den sogenannten Nicht-Standard-Anwendungsbereichen (Ingenieurtätigkeiten [KL88], Büroinformationssysteme [Tsi85], kartographische Anwendungen [LNE89]) als unverzichtbares Konzept herausgestellt.

Semantische Modelle für die Objektschicht sind Interpretationsstrukturen für eine mehrsortige Prädikatenlogik, die die vorgegebenen Integritätsbedingungen erfüllen. Besonders beachtet werden muß hierbei die konsistente Integration der Datenschichtalgebren als fest vorgegebene Teilstruktur sowie die Modellierung des Objektidentifikationsmechanismus. Eine detaillierte Behandlung dieses Problemfelds kann in [EDG88] gefunden werden. Die semantische Modellkonstruktion für ein erweitertes Entity-Relationship-Modell wurde in [HG88, GH91, Hoh90] beschrieben.

Zusätzlich zur Strukturbeschreibung werden die möglichen Ausprägungen der Datenbank durch explizite statische *Integritätsbedingungen* eingeschränkt. Viele (einfache) Integritätsbedingungen können oft bereits durch Modellierungskonstrukte des verwendeten Datenmodells implizit ausgedrückt werden, so funktionale Abhängigkeiten oder Kardinalitäten von Beziehungen. Explizite Integritätsbedingungen werden in Prädikatenlogik formuliert.

Beispiel 2.3.2 Als Beispiel formulieren wir die statische Integritätsbedingung, daß ein Manager bzw. eine Managerin immer mehr verdient als die Angestellten, die in seiner bzw. ihrer Abteilung arbeiten.

```
FOR ALL (P : PERSON), (M : MANAGER), (D : DEPARTMENT) :
  ( Works_For(P,D) AND D.manager = M AND NOT P = PERSON(M) )
      IMPLIES P.salary < M.salary;
```

In dem Beispiel wurde eine implizite Konversionsfunktion eines `MANAGER`-Objekts in ein `PERSON`-Objekt benutzt, die durch die Unterklassenhierarchie (Spezialisierungen) definiert ist. □

Integritätsbedingungen werden in der semantischen Konstruktion als prädikatenlogische Axiome aufgefaßt, die die erlaubten Ausprägungen von Datenbankzuständen einschränken.

2.3.3 Entwicklungsschicht

Die *Entwicklungsschicht* beschreibt die möglichen zeitlichen Entwicklungen des Datenbestandes unabhängig von Anwendungstransaktionen. Grundlage hierfür ist eine temporale Objektidentität [KC86], d.h. die Identität von Datenbankobjekten überlebt Datenbankänderungen. Diese Eigenschaft ist in wertebasierten Datenmodellen, z.B. dem Relationenmodell und seinen Erweiterungen, nicht gegeben. Einschränkungen der zeitlichen Entwicklung werden als *dynamische* bzw. *temporale Integritätsbedingungen* bezeichnet. Beispiele für temporale Integritätsbedingungen sind

- *"Das Gehalt von Angestellten darf nicht sinken."*
- *"Flugzeuge müssen mindestens einmal im Jahr gewartet werden, spätestens aber alle 100.000 Kilometer Flugleistung."*
- *"Angestellte müssen ihren jährlichen Urlaub spätestens bis zum Mai des folgenden Jahres genommen haben."*

Temporale Integritätsbedingungen können in verschiedenen Spezifikationsformalismen beschrieben werden :

- Als logikbasierte Spezifikationssprache kann eine an die Anforderungen des Datenbankentwurfs angepaßte *temporale Logik* benutzt werden. Verschiedene temporale Logiken wurden für diesen Zweck in der Literatur vorgeschlagen [Ser80, ELG84, Saa88, Lip89, Saa91b, Cho92a].

  Während [Ser80, ELG84, Saa88, Lip89, Saa91b] zukunftsgerichtete temporale Logiken basierend auf kausaler Zeit (auch Transaktionszeit genannt) vorschlagen, schlägt J. Chomicki in [Cho92a] vergangenheitsgerichtete temporale Logik vor und präsentiert in [Cho92b] eine Erweiterung dieser Logik um Echtzeitoperatoren.

- Ein alternative Methode, temporale Bedingungen zu formulieren, ist die Spezifikation von Übergangsautomaten oder einfachen Petrinetzen. Der Einsatz von Petrinetzen für diese Aufgabe wurde vorgeschlagen z.B. in [EKTW86] und in der Dissertation von A. Oberweis [Obe90].

Diese beiden Ansätze sind in dem Sinne äquivalent, daß aus temporalen Formeln die äquivalenten Übergangsautomaten konstruiert werden können [LS87, Saa88, Lip89, HS91]. Auch der umgekehrte Weg ist möglich, da aus Übergangsautomaten in temporaler Logik formulierte transitionale Integritätsbedingungen abgeleitet werden können.

Da in späteren Kapiteln eine ausführliche Einführung in temporale Logik folgen wird, wird an dieser Stelle nur das folgende Beispiel für eine temporale Integritätsbedingung aufgeführt.

Beispiel 2.3.3 Die temporale Integritätsbedingung "*Gehälter von Angestellten dürfen nicht sinken.*" kann wie folgt formuliert werden :

```
FOR ALL (E : EMPLOYEE) (s : integer):
  ALWAYS (E.salary = s IMPLIES ALWAYS NOT E.salary < s );
```

Der temporale Operator `ALWAYS` bezeichnet eine Quantifizierung über *alle* zukünftigen Zustände. Das erste `ALWAYS` definiert die von ihm gebundene Teilformel als Invariante, d.h. die Formel muß von allen eingefügten `EMPLOYEE`-Objekten in allen zukünftigen Datenbankentwicklungen erfüllt werden. Die innere Implikation fordert, daß, falls das Gehalt eines Angestellten (der Wert des `salary`-Attributes) jemals gleich dem Wert `s` ist, es dann in allen Folgezuständen größer gleich diesem Wert sein muß. Dies folgt aus der inneren Quantifizierung mit `ALWAYS`. □

Semantische Modellbildung für die Entwicklungsschicht sind Folgen von Datenbankzuständen, d.h. Folgen von prädikatenlogischen Interpretationsstrukturen. Formal können derartige Folgen als Kripke-Strukturen modellierte werden [Saa88, Saa89].

Da Spezifikationen in temporaler Logik leicht unübersichtlich werden, können endliche Automaten zur grafischen Verdeutlichung aus derartigen Formeln konstruiert werden, die gleichzeitig eine Konsistenzüberprüfung ermöglichen [Saa88, SL89, Lip89, LF89, Saa91b].

2.3.4 Aktionenschicht

Auf der *Aktionenschicht* werden — anwendungsspezifisch — die elementaren Datenbankänderungstransaktionen beschrieben. Diese elementaren Aktionen sind die kleinsten integritätsrespektierenden Bestandteile von Anwendungsprogrammen. In der Datenbankterminologie werden integritätsrespektierende Datenbankänderungen als *Transaktionen* bezeichnet. Wir bezeichnen diese Schicht bewußt nicht als Transaktionsschicht, da einerseits mit dem Begriff 'Transaktion' oft weitere Konzepte verbunden werden, von denen auf der konzeptionellen Ebene abstrahiert werden sollte, und andererseits der moderne Transaktionsbegriff [WR90] oft von der Elementaritätsforderung abweicht (Stichwort 'lange Transaktionen').

Zu beschreibende Elemente der Aktionenschicht sind modellierungsabhängige Datenbankänderungen, d.h. (parametrisierte) Funktionen, die einen vorgegebenen korrekten Datenbankzustand in einen neuen korrekten Zustand überführen.

Beispiele für elementare Aktionen sind :

- Eintrag einer Flugreservierung in eine Reisebüro-Datenbank.
- Löschen eines Angestellten.
- Lohnerhöhung aller Angestellten der Abteilung 'Forschung & Entwicklung' um zehn Prozent.

Für derartige Aktionen sind in der Literatur mehrere Beschreibungsformalismen vorgeschlagen worden.

- Die Spezifikation von Aktionen durch Vor- und Nachbedingungen ermöglicht eine deskriptive abstrakte Beschreibung des Effekts einer Aktion [Lip89, Lip90, EGH+90, EGH+92]. Ein Problem dabei ist, daß die Änderung in der Regel nicht eindeutig determiniert ist, d.h., daß als semantische Modellbildung Relationen zwischen Datenbankzuständen anstatt Funktionen eingesetzt werden müssen.
- Viele Datenmodelle und Entwurfsmethoden bieten operationale Sprachen zur Definition von Datenbankaktionen an, die an Konzepten imperativer Programmiersprachen orientiert sind [EGH+90, EGH+92]. Allerdings ist bei diesen Ansätzen oft keine ausreichend formale Semantik angegeben.

Wie bereits erwähnt, ist die semantische Modellbildung für Aktionen eine Menge von Funktionen (bzw. Relationen) zwischen Datenbankzuständen. Die semantische Modellbildung für Folgen von Aktionen entspricht dann der der Entwicklungsschicht (aus diesem Grund sind die beiden Schichten in Abbildung 2.2 auf gleicher Höhe angeordnet). Eine interessante Fragestellung ist darum die Konsistenz der kombinierten Semantik beider Schichten, d.h. die Existenz durch Aktionen ausführbarer Zustandsfolgen, die korrekt bezüglich der Entwicklungsschicht sind [Lip89, LS88].

Als Beispiel für eine Aktionsspezifikation geben wir die Definition einer elementaren Transaktion mittels Vor- und Nachbedingungen an.

Beispiel 2.3.4 Die Transaktion `FireEmployee` entfernt eine Person aus der Datenbank, falls er oder sie nicht momentan Manager einer anderen Person ist.

```
ACTION FireEmployee (person_name : string);
  VARIABLES P : PERSON;
  PRECONDITION P.name = person_name IMPLIES
               NOT EXISTS (PP : PERSON) P = PERSON(PP.manager);
  POSTCONDITION NOT EXISTS (P : PERSON) P.name = person_name;
```

Die Objektvariable `P` ist implizit allquantifiziert über alle aktuell gespeicherten Personen der Klasse `PERSON`. In diesem Beispiel wird die explizite Konvertierungsfunktion von `MANAGER`-Objekten zu `PERSON`-Objekten syntaktisch als `PERSON(..)` notiert. □

Insbesondere bei der Spezifikation mittels Vor- und Nachbedingungen müssen zwei implizite Regeln beachtet werden, die es in vielen Fällen erlauben, aus der Relation möglicher Änderungen eine eindeutige Funktion als Semantik der Änderungsaktion auszuwählen :

- Die *Integritätsregel* besagt, daß Aktionen integritätsbewahrend sein müssen, also korrekte Zustände wieder in korrekte überführen. Dies kann in der Regel durch zusätzliche Vor- und Nachbedingungen erreicht werden (Stichwort 'Update Propagation').

- Die *Minimalitäts-* oder *implizite Rahmenregel* besagt, daß die Änderung durch eine Aktion möglichst minimal sein sollte, um die Nachbedingung zu erfüllen. Eine bekannte Faustregel dafür ist z.B., daß nur explizit erwähnte Prädikate und Funktionen geändert werden dürfen. Die Auswahl einer minimalen Änderung ist im Allgemeinen nicht immer möglich, so bei disjunktiven Nachbedingungen. Eine ausführliche Betrachtung dieser Problematik kann in den Arbeiten von U. Lipeck [Lip89, Lip90] gefunden werden.

Diese beiden Regeln arbeiten natürlich in gewisser Weise gegensätzlich : Die Integritätsregel erweitert eventuell den Aktionseffekt um die Integrität zu wahren, während die implizite Rahmenregel versucht diesen Effekt möglichst minimal zu halten.

2.3.5 Prozeßschicht

Auf der *Prozeßschicht* (oder auch *Anwendungsschicht* [Saa91a]) werden die eigentlichen Anwendungsprozesse eines Informationssystems beschrieben. Dies ist in der Regel nicht durch Vor– und Nachbedingungen wie bei der Aktionenschicht möglich, da derartige Anwendungen in der Regel interaktiv sind oder sogar auf mehrere Datenbanken und andere Ressourcen zugreifen. Mit anderen Worten, derartige Anwendungsprogramme sind *kommunizierende Prozesse* und müssen mit entsprechenden Formalismen spezifiziert werden, zum Beispiel durch Petrinetze oder abstrakte Prozeßbeschreibungssprachen.

Beispiele für derartige Anwendungsprozesse sind

- eine Flugbuchungssitzung im Reisebüro, bei der in einer interaktiven Beratungssitzung ein freier passender Flug ausgewählt und schlußendlich gebucht wird,
- sogenannte 'lange' Transaktionen im Ingenieurbereich, bei denen Versionenverwaltung und Teamarbeit eine Rolle spielen,
- oder das Erstellen eines Firmen-Monatsberichts, bei dem Daten aus mehreren (oft geographisch verteilten) Teilbereichen zusammengestellt und aufbereitet werden muß.

Wie bereits im Abschnitt 1.3.5 ausgeführt, kommen als Spezifikationsformalismen für die Prozeßschicht die verschiedenen Prozeßbeschreibungsmethoden in Frage, die bisher entwickelt wurden :

- *Petrinetze* [Rei85] ermöglichen eine direkte Darstellung der verteilten Abläufe von Prozessen. Für Petrinetze existiert eine Fülle von Analyse- und Prototyping-Werkzeugen.

- *Prozeßalgebren und -kalküle* beschreiben Prozesse durch Komposition komplexer Prozesse aus elementaren Prozessen [Hoa85, Hen88, Mil89].

- *Temporale Logiken* werden zur deskriptiven Beschreibung von Prozeßeigenschaften eingesetzt [MP81, MP92], aus denen operationale Prozeßdefinitionen generiert werden können [MW84].

Die semantische Modellbildung für diese oberste Schicht (die ja alle anderen als Komponenten enthält) ist naturgemäß sehr komplex. Die Datenbank selber und die Anwendungsprozesse werden durch kommunizierende Prozesse beschrieben, wobei dem Datenbankprozeß eine Folge von Datenbankzuständen als sogenannte Beobachtung zugeordnet ist. Die elementaren Aktionen bilden das Ereignisalphabet des Datenbankprozesses, und jeder Folge von derartigen Ereignissen wird eine Folge von Datenbankzuständen als Beobachtung zugeordnet. Einer Datenbankentwicklung entspricht somit ein, in der Regel nicht terminierender, linearer Prozeß.

Andere Komponenten eines Informationssystems werden ebenfalls durch Prozesse beschrieben :

- *Interaktive Benutzerschnittstellen* sind Beispiele für Komponenten eines Informationssystems, die konzeptionell nur als kommunizierende Prozesse beschreibbar sind.

- Die sogenannten *langen Transaktionen* [LP83, KLMP84] sind Beispiele für in der Regel operational beschriebene, terminierende Prozesse, die eine verteilte, oft langwierige Tätigkeit beschreiben.

- *Aktive Systemkomponenten* wie die Systemuhr oder ein Kalender müssen als aktive Prozesse beschrieben werden, die eventuell andere Ereignisse anstoßen können. Zu diesen aktiven Komponenten gehören konzeptionell auch Sensoren und ähnliche Überwachungskomponenten in Steuerungssystemen.

- Andere *Softwaresysteme* wie CAD-Software oder Produktionsplanungssysteme müssen in der Regel als eingekapselte Prozesse beschrieben werden, von denen nur die Zugriffsschnittstelle bekannt ist.

- Als letzter Punkt sollte erwähnt werden, daß in der Regel *mehrere, eventuell verteilte, Datenbanken* als unabhängige Prozesse modelliert werden müssen. Dies bedeutet in letzter Konsequenz den Schritt von der zentralen Systemarchitektur hin zu Anwendungssystemen bestehend aus interoperablen Komponenten heterogener Architektur [SW91].

Die Forderung nach über den klassischen, elementaren Transaktionsbegriff hinausgehender Konzepte wird oft mit dem Aufkommen sogenannter "Nicht-Standard-Anwendungen" verbunden, etwa im CAD- oder Büro-Bereich [LP83, KLMP84, Tsi85].

Das folgende Beispiel zeigt, daß komplexe Anwendungsprozesse auch in typischen etablierten Datenbankanwendungen eher die Regel als die Ausnahmen sind.

Beispiel 2.3.5 Ein typisches Beispiel für einen Anwendungsprozeß ist eine Sitzung im Reisebüro, in deren Verlauf eine Flugbuchung vorgenommen werden soll.

> Der eigentliche Buchungsprozeß besteht u.a. aus einer Schleife, in der — in Interaktion mit dem Kunden — ein freier Platz für einen Flug gesucht wird. Da Flüge oft bereit voll ausgebucht sind, müssen mehrfach Parameter (die Reisewünsche) in Absprache mit dem Kunden modifiziert werden.
>
> Während des Buchungsprozesses werden mehrfach Anfrageereignisse (eventuell mit provisorischen Sperrwünschen für Plätze) an den Datenbankprozeß geschickt. Eine erfolgreiche Sitzung endet mit einer Buchungsaktion der Datenbank (und dem Anstoß eines weiteren Prozesses, der sich um die Abwicklung der Rechnungserstellung kümmert).

Zu beachten ist bei diesem Buchungsablauf insbesondere, daß in einer Schleife wiederholt Buchungsparameter interaktiv vom Benutzer abgerufen werden, deren Werte von den Ergebnissen bisheriger Anfragen abhängen können. Dieser Prozeß kann also *nicht* als elementare Transaktion mit fester Parameteranzahl modelliert werden ! □

Wie bereits erwähnt, können für den Entwurf von Anwendungsprozessen verschiedene Beschreibungsmethoden aus dem Bereich des Software Engineerings eingesetzt werden, die zum Entwurf reaktiver Systeme entwickelt wurden.

Im Datenbankbereich haben sich — besonders beim Entwurf (vergleiche [Lip89]) — Sprachvorschläge und Konzepte hauptsächlich auf elementare Transaktionen beschränkt. Die Erweiterung dieser Ansätze wurde spätestens notwendig mit der Erkenntnis, daß für viele Anwendungsklassen das klassische Transaktionskonzept nicht adäquat ist und darum neue Konzepte wie etwa 'lange Transaktionen' oder 'Entwurfstransaktionen' entwickelt wurden.

Ein Beispiel für einen Sprachvorschlag zur Beschreibung von Anwendungsprozessen im Datenbankbereich ist das ConTract-Modell von A. Reuter und H. Wächter [WR90, RW91]. Das ConTract-Modell ermöglicht die Komposition komplexer Prozesse aus elementaren, klassischen Transaktionen.

Beispiel 2.3.6 Als Beispiel für eine Prozeßbeschreibung im ConTract-Modell wird eine Ablaufspezifikation in Anlehnung an das Beispiel 2.3.5 in Abbildung 2.4 dargestellt.

Das Beispiel beschreibt einen Anwendungsprozeß, der aus mehreren elementaren Transaktionen S_1 bis S_{11} zusammengesetzt ist. Elementare Transaktionen können dabei die Eingabe von Parametern, Abfragen an Datenbanken oder Änderungen von

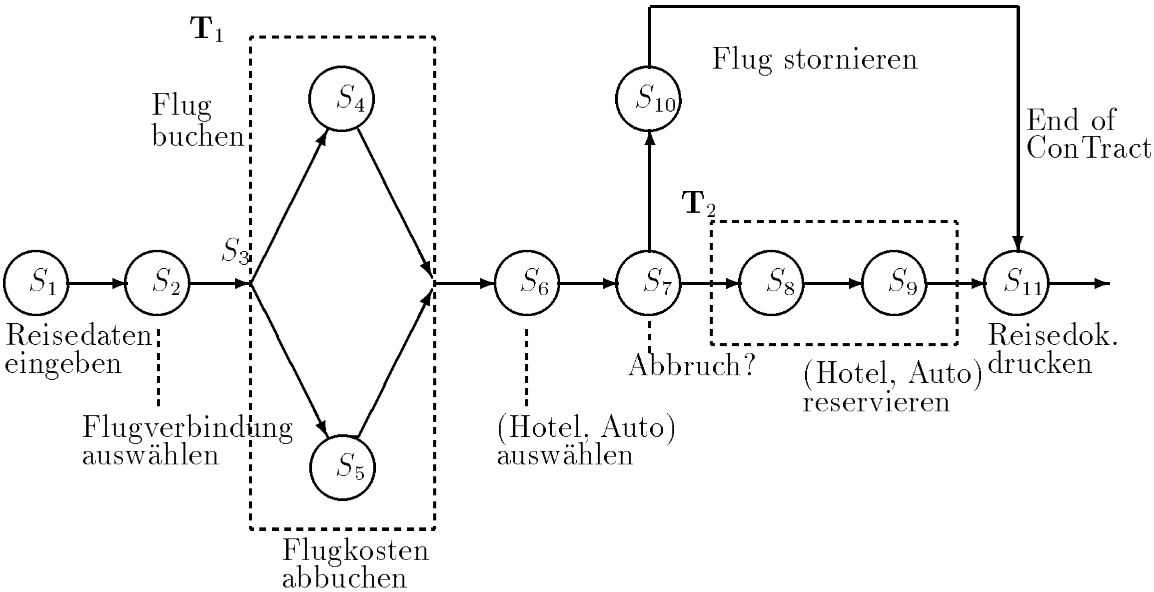

Abbildung 2.4: Beispiel einer Ablaufspezifikation im ConTract-Modell (aus [WL91]).

Datenbanken sein. Teilprozesse können logisch zu Transaktionen ($\mathbf{T}_1$ und $\mathbf{T}_2$) zusammengefaßt werden, d.h., der Teilprozeß wird entweder ganz oder gar nicht ausgeführt und hinterläßt die betroffene Datenbank in einem konsistentem Zustand. □

2.4 Probleme mit dem vorgestellten Ansatz

Der vorgestellte Schichtenansatz unterstützt eine 'horizontale' Strukturierung der Entwurfsdokumente mit dem Ziel, die Komplexität der semantischen Modellbildung in den Griff zu bekommen. Die einzelnen Schichten, insbesondere die für Informationssysteme zentrale Objektschicht, werden hingegen durch Spezifikationsdokumente beschrieben, deren Beschreibungsformalismen keine weitere 'vertikale' Strukturierung in Teilsysteme unterstützt. Das Hauptproblem des vorgestellten Ansatzes ist somit die Handhabung großer Entwurfsdokumente.

Aus Erfahrungsberichten über den Einsatz von ER-Modellen in praktischen Projekten ist bekannt, daß die Objektschicht bereits für kleinere Anwendungen eine Komplexität erreicht, in der die ansonsten übliche graphische Darstellung des Gesamtschemas nicht mehr praktikabel ist. Die existierenden Ansätze, ein Modularisierungskonzept in den ER-Ansatz zu integrieren, beschränken sich in der Regel auf die graphische Darstellung und vernachlässigen den semantischen Aspekt der Strukturierung.

Warum ist die Handhabung großer Entwurfsdokumente ein derartig gravierendes Problem im Systementwurf ? Um diese Frage zu beantworten, betrachten wir ein paar wichtige (und besonders kritische) Tätigkeiten im Entwurf:

- Eine zentrale Tätigkeit ist sicher das Erstellen, Ändern und Präsentieren von Entwurfsdokumenten.
- Erstellte Dokumente müssen auf Konsistenz überprüft und gegen die Anforderungen validiert werden.
- Die Entwurfsdokumente bilden die Basis für die Erstellung von lauffähigen Prototypen und für die schlußendliche Implementierung (und Wartung).
- All diese Tätigkeiten werden bei größeren Systemen in Teamarbeit durchgeführt.

Bei all diesen Tätigkeiten spielt die Größe der Einzeldokumente, die in jedem Schritt als Einheit behandelt werden müssen, eine zentrale Rolle. Wir können grob drei Komplexitätsproblembereiche wie folgt klassifizieren :

1. Die *Entwurfskomplexität* betrifft die Größe der Einzeldokumente an sich. Das typische Beispiel ist ein (unstrukturiertes) ER-Diagram mit mehreren hundert Objektklassen und Beziehungstypen. Dieses Problem tritt insbesondere auf der Objekt- und Entwicklungsschicht auf, da (statische und dynamische) Integritätsbedingungen globale Auswirkungen haben können und somit nicht rein lokal behandelt werden dürfen. Dies gilt ebenfalls für die Aktionenschicht (Stichwort 'update propagation' aufgrund der Integritätsregel). Sowohl für die Datenschicht (ADTen) als auch für die Prozeßschicht hingegen sind formale Modularisierungskonzepte bekannt.

   Die Entwurfskomplexität ist in gewissem Sinne inhärent zum Schichtenansatz, da eine Strukturierung des Gesamtsystems (und damit der einzelnen Dokumente) vertikal zur horizontalen Schichtenbildung erfolgen müßte.

2. Die *Modellbildungskomplexität* betrifft das Problem schichtenübergreifender Konsistenzprüfungen. Obwohl der Schichtenansatz u.a. entwickelt wurde, um diese Problematik in den Griff zu bekommen, stellt diese Konsistenzfrage ein großes Problem dar. Allein die Erfüllbarkeit prädikatenlogischer Axiome (der statischen Integritätsbedingungen) bildet einen Teilkomplex, der für die (globale !) Objektschicht nur schwer in den Griff zu bekommen ist. Weitere kritische Teilbereiche sind sicherlich der Abgleich der temporalen Integritätsbedingungen mit den Aktionenspezifikationen (siehe z.B. die Diskussion in [Lip89, SL89, Saa91b]) sowie die Analyse der Anwendungsprozesse und ihres Kommunikationsverhaltens.

3. Die *Änderungskomplexität* betrifft die Änderbarkeit und Erweiterbarkeit von Entwurfsdokumenten. Diese Vorgänge sind besonders in frühen Entwurfsphasen

sehr häufig und ihre Unterstützung von großer Bedeutung. Dieser Aspekt ist auch unter dem Stichwort *inkrementeller Entwurf* bekannt. Während das rein syntaktische Ändern von Dokumenten z.B. durch strukturorientierte Editoren unterstützt werden kann, treten bei der Konsistenzprüfung und der Prototyperzeugung prinzipielle semantische Probleme auf, die im Kern auf der Fragestellung beruhen, welche Änderungen lokal behandelt werden können bzw. welche Änderungen möglicherweise globale Auswirkungen haben.

Ein weiterer mit der Änderungskomplexität in Zusammenhang stehender Aspekt ist die Unterstützung der *Wiederverwendbarkeit* von Teilentwürfen, wobei neben dem syntaktischen Aspekt auch die garantierte Konsistenz der verwendeten Teilentwürfe und die etwaige Wiederverwendbarkeit von Implementierungen dieser Teilentwürfe eine Rolle spielen.

All diese drei Komplexitätsaspekte können natürlich nicht unabhängig voneinander betrachtet werden. Gemeinsam ist ihnen allen, daß die Komplexität der erwähnten Probleme in der Regel mehr als linear mit der Größe des zu entwerfenden Informationssystems anwachsen.

Ein weiteres Problem mit dem vorgestellten Ansatz ist der Zuschnitt auf die 'klassische' Anwendungsarchitektur von Informationssystemen mit einer zentralen Datenbank. Neuere Konzepte der Anwendungs- und Datenbanktechnik lassen sich nur schwer integrieren. Beispiele hierfür sind

- die Integration einer *Versions- und Variantenverwaltung* durch die Datenbank, wie sie insbesondere in Nicht-Standard-Anwendungsgebieten eine große Rolle spielen,
- die insbesondere im Bürobereich wichtigen *aktiven Datenbankkomponenten* (der Ablauf von Fristen 'triggert' gewisse Ereignissequenzen) [Day88, BM91],
- sowie der Zugriff aus der Datenbank hinaus auf andere Systemkomponenten (Zugriff auf Systemuhr / -kalender).

All diese Probleme resultieren aus der Rolle der Datenbank als einem einzigen *passiven* Prozeß und der hierarchischen Einordung dieses Prozesses unterhalb der Prozeßschicht. Diese Beispiele zeigen auch das Problem der *Adäquatheit* der vorgestellten Anwendungsarchitektur für moderne Software-Architekturen auf — Stichworte wie verteilte Anwendungen, Interoperabilität, offene Systeme etc. zeigen einen Tend zu Systemarchitekturen auf, die dem zentralistischen Architekturansatz der vorgestellten Methodik entgegenlaufen [SW91].

Der in den folgenden Abschnitten vorgestellte objektorientierte Entwurfsansatz hat das Ziel, gerade diese Nachteile des Schichtenansatzes in den Griff zu bekommen, ohne auf die bewährten Entwurfsprinzipien dieses Ansatzes völlig zu verzichten.

2.5 Beziehungen zu objektorientierten Entwurfsprinzipien

Das vorgestellte Schichtenmodell unterstützt bisher nicht eine Modularisierung des konzeptionellen Entwurfs. Eine derartige Modularisierung erlaubt lokale Konsistenzüberprüfungen sowie eine Aufteilung des Entwurfsdokuments auf mehrere EntwerferInnen. Weiterer potentieller Vorteil ist die Wiederverwendbarkeit von Teilentwürfen.

Ein vielversprechender Ansatz dafür ist die bereits in Abschnitt 1.4 vorgestellte *objektorientierte Spezifikation von Informationssystemen* wie sie von A. Sernadas, C. Sernadas und H.-D. Ehrich in [SSE87] angeregt wurde. In diesem Ansatz wird eine mehrschichtige Spezifikation vertikal aufgeteilt in Objektmodule, die sowohl den Struktur- als auch den Dynamikanteil lokal zu einzeln Informationseinheiten (den "Objekten") beschreiben. Um die Beziehung zu dem vorgestellten Schichtenansatz deutlicher zu machen, rekapitulieren wir kurz die wesentlichen Eigenschaften derartiger Objekte :

- Objekte sind beobachtbar durch Attribute, d.h. sie haben einen *strukturierten beobachtbaren aktuellen Zustand.*
- Die zeitliche Entwicklung von Objekten wird allein durch die *Abfolge von Ereignissen* bestimmt.
- Zusammenfassend kann somit gesagt werden : *Objekte sind beobachtbare Prozesse !*

Objekte sind demnach eindeutig beschrieben durch einen Prozeß über objektspezifischen Ereignissen und einen Lesezugriff auf beobachtbare Attribute. Im Vergleich mit den Ausführungen über die Schichtenarchitektur folgern wir :

> *Die semantische Modellbildung für Objekte ist identisch zur Modellbildung für die Prozeßschicht.*

Insbesondere der Datenbankprozeß selber ist ein typisches Beispiel für einen beobachtbaren Prozeß, da sein Zustand über Anfragen an den aktuellen Datenbankzustand beobachtet werden kann.

Aus dieser grundlegenden Erkenntnis heraus können wir einige weitere Schlußfolgerungen ziehen, die den Bezug des Schichtenansatzes zum objektorientierten Entwurf deutlicher machen :

- Die in vier Schichten beschriebene Datenbank des Schichtenansatzes kann als ein (allerdings in der Regel ziemlich komplexes) Objekt aufgefaßt werden. Als ein Seiteneffekt können auf diese Weise bereits existierende Datenbanken in den objektorientierten Entwurf auf natürliche Weise integriert werden.

- Passive Objekte sind in der Sichtweise des Schichtenansatzes *'Datenbanken im Kleinen'*. Zur Objektbeschreibung können somit auch Datenbankentwurfstechniken eingesetzt werden.
- Konzepte zur sicheren Objektkomposition sind gut geeignet, um konzeptionelle Datenbankbeschreibungen semantisch fundiert zu strukturieren.
- Das Zusammenspiel von lokaler Analyse und sicherer Objektkomposition hilft bei objektorientierter Strukturierung, die Komplexität der Konsistenzprüfungen in den Griff zu bekommen.
- Der objektorientierte Ansatz unterstützt auf vielfältige Weise die Wiederverwendbarkeit von Beschreibungs- und Systemteilen.
- Der Begriff des eingekapselten Objektes überträgt die aus Programmiersprachen bekannten Vorteile der Modularisierung auf den konzeptionellen Entwurf von Informationssystemen.

Diese Aufzählung macht deutlich, wie die objektorientierte Schemamodularisierung helfen könnte, die beschriebenen Probleme des Schichtensansatzes zu bewältigen.

Nach diesen Reflektionen über den prinzipiellen Zusammenhang zwischen objektorientierten Entwurfsprinzipien und dem Schichtenansatz wird nun in den folgenden Kapiteln eine Entwurfssprache vorgestellt, die den objektorientierten konzeptionellen Entwurf unterstützt.

Literaturhinweise

Der mehrschichtige Entwurf von Datenbankanwendungen wurde von mehreren Arbeitsgruppen vorgeschlagen, so von [SFNC84, CS87, Saa91a, EGH+92]. Die vorgeschlagene Schichtung und Auswahl von Spezifikationsformalismen ist den Artikeln [Saa91a, EGH+92] entnommen.

Empfehlenswerte Lehrbücher über algebraische Spezifikation abstrakter Datentypen sind die Bücher von H. Ehrig und B. Mahr [EM85, EM90] und das deutschsprachige Buch von H.-D. Ehrich, M. Gogolla und U. Lipeck [EGL89].

Konzeptionelle Datenmodelle werden in den Übersichtsartikeln von R. Hull und R. King [HK87], S. Urban und L. Delcambre [UD86], und von J. Peckham und F. Maryanski [PM88] diskutiert. Spezielle erweiterte Entity-Relationship-Modelle können im Lehrbuch von R. Elmasri und S. Navathe [EN89] und in den Artikeln von R. Elmasri et al [EWH85] und von G. Engels et al [EGH+92] gefunden werden.

Dynamische Integritätsbedingungen sind ausführlich in der Habilitation von U. Lipeck [Lip89] und der Dissertation von G. Saake [Saa88] behandelt. Die Diskussion der Spezifikation von Transaktionen mit Vor- und Nachbedingungen kann ebenfalls in der Habilitation von U. Lipeck [Lip89] gefunden werden. Spezielle Artikel zu diesem Themenbereich mit Bezug zu dem hier vorgestellten Ansatz sind [Ser80, ELG84, LEG85, LS87, SL88, LS88, Lip90, HS90, HS91, Saa91b].

Die Spezifikation von Anwendungsprozessen basierend auf konzeptionellen Datenbankbeschreibungen ist ein neuerer Ansatz, der sich noch nicht in Lehrbüchern niedergeschlagen hat. Die benötigten Basiskonzepte können in der Literatur über Nichtstandardtransaktionskonzepte gefunden werden, so etwa im Artikel von H. Wächter und A. Reuter [WR90].

Kapitel 3

Formale Beschreibung von Objekten

In den folgenden Abschnitten soll ein *Objektkonzept* als Basiseinheit des konzeptionellen Entwurfs formalisiert werden. Wir beginnen in diesem Kapitel mit der Formalisierung einzelner isolierter Objekte und ihrer Klassifizierung in Objektklassen, d.h. wir betrachten vorerst keine Modellierungskonzepte die die Zusammenhänge zwischen mehreren Objekten beschreiben, so etwa Konzepte wie Objektkommunikation, Konstruktion komplexer Objekte etc.

Bereits in den Abschnitten 1.4 und 2.5 wurden Eigenschaften von Objekten als Entwurfskonzepte diskutiert, die wir im folgenden kurz rekapitulieren werden. *Einzelne Objekte* werden durch folgende Eigenschaften charakterisiert :

- Objekte haben *beobachtbare Attribute und Ereignisse*, d.h. sie haben eine wohldefinierte Lese- und Manipulationsschnittstelle zur Außenwelt.
- Objekte benutzen *Grunddatenstrukturen* als Wertebereiche für Attribute und Ereignisparameter, d.h. sie importieren abstrakte Datentypen als Wertebereiche.
- Objekte haben einen *internen Zustand*, der insbesondere die aktuellen Werte der Attribute bestimmt, von dem aber auch der Effekt von Ereignissen abhängen kann.
- Der bisherige Lebenslauf eines Objektes ist durch den bisherigen *Ereignisprozeß* vorgegeben, der den aktuellen internen Zustand determiniert.
- Objekte können *erzeugt und gelöscht* werden.
- Objekte können *persistent gespeichert* sein.
- Mehrere Objekte werden in *Objektklassen* klassifiziert. Innerhalb einer Klasse haben sie einen eindeutigen *Identifikator*.

Um ein derart charakterisiertes Objektkonzept zu formalisieren, müssen mehrere Punkte beachtet werden :

1. Der Bereich der *semantischen Modelle* für Objektbeschreibungen muß festgelegt werden, d.h. diejenigen Strukturen, die den zu beschreibenden Objekten formal entsprechen.

2. Ein *Kalkül* zur Beschreibung von Objekten muß entwickelt werden, d.h. eine formale Sprache zur Formulierung von Eigenschaften derartiger mathemathischer Strukturen.

   Zwischen Kalkülausdrucken und semantischen Modellen muß eine 'erfüllt'-Relation festgelegt werden, die für ein gegebenes Modell festlegt, ob es die im Kalkülausdruck beschriebenen Eigenschaften hat oder nicht. Diese Beziehung legt dann für einen gegebenen Kalkülausdruck die Klasse aller semantischen Modelle fest, die den Ausdruck erfüllen. Wünschenswert ist hierbei zusätzlich eine konstruktive Semantik, die zumindest ein ausgezeichnetes Element der beschriebenen Modellklasse für einen vorgegebenen Kalkülausdruck konstruktiv bestimmt.

3. Ein letzter, oft vernachlässigter Punkt ist die Entwicklung einer geeigneten *Spezifikationssprache* als 'syntaktischer Zucker' für Kalkülausdrücke. Eine derartige Sprache muß lesbare und benutzerfreundliche Darstellungen von Objektspezifikationen ermöglichen. Die Sprache sollte den Modellierungsprozeß durch geeignete syntaktische Regeln unterstützen, so etwa durch strenge explizite Typisierung von Variablen, Blockbereiche für die Gültigkeit von Variablen, explizite Notationen für Variablendeklarationen und Import von Datentypen u.v.a.m.

Im folgenden wird ein Formalismus zur Beschreibung von Objekten und Objektklassen vorgestellt. Anforderungen und der Aufbau von Objektbeschreibungen sind Inhalt des ersten Abschnitts. Als beispielhafte objektorientierte Spezifikationssprachmittel werden Konzepte der Sprachen OBLOG [SSE87], **Oblog**$^+$ [SJ91, JSS91] und TROLL [SJ92, JSH91, JHSS91, JSHS91] vorgestellt, die im Rahmen einer europäischen Zusammenarbeit im Gebiet 'Grundlagen der objektorientierten Spezifikation von Informationssystemen' (ESPRIT BRA IS-CORE) entwickelt wurden.

In Gegensatz zu den ersten beiden Sprachen, gibt es für TROLL eine vollständige Sprachbeschreibung [JSHS91]. OBLOG und **Oblog**$^+$ sind Vorläufer der TROLL-Sprache und unterscheiden sich sich von ihr in der konkreten Syntax und in den zur Verfügung gestellten Modellierungskonzepten, aber nicht in den Basiskonzepten der objektorientierten Spezifikation. Im folgenden werden sowohl **Oblog**$^+$- und TROLL-Sprachmittel vorgestellt, da beide Sprachen für bestimmte Konstruktionen unterschiedliche Sprachkonzepte und syntaktische Realisierungen einsetzen. Viele Sprachmittel von **Oblog$^+$** finden sich in TROLL z.T. in abgewandelter Syntax und mit anderen Schlüsselworten wieder.

In den folgenden Abschnitten folgen Beispiele für Objekt- und Objektklassenspezifikationen als informeller Einstieg in die Konzepte der Spezifikationssprachen **Oblog$^+$** und TROLL. Im Abschnitt 3.3 wird eine semantische Modellbildung für einzelne Objekte definiert. Es folgen Abschnitte, die die verschiedenen Kalküle zur Beschreibung von Objekteigenschaften einführen und ihre Umsetzung in die Spezifikationssprache vorstellen. Das Kapitel schließt mit der Formalisierung von Objektklassen und Klassentypen als Mechanismus zur Beschreibung ganzer Klassen von 'ähnlichen' Objekten.

3.1 Aufbau von Objektbeschreibungen

Die semantische Modellbildung für ein einzelnes Objekt ist ein *beobachtbarer Prozeß*. Um ein derartiges Objekt isoliert formal zu beschreiben, müssen insbesondere die folgenden Punkte spezifiziert werden :

- Der Objektbezeichner, über den auf das Objekt zugegriffen werden kann.
- Die Schnittstelle des eingekapselten Objektes, d.h. die dem Objekt zugeordneten Attribute und Ereignisse.
- Die Datentypen, die die Wertebereiche für Attribute und Ereignisparameter bilden.
- Die erlaubten Ausprägungen der beobachteten Attributentwicklungen durch Angabe von Integritätsbedingungen.
- Der Effekt der Ereignisse auf die beobachteten Attribute.
- Der Prozeß selber, d.h. die erlaubten Lebensläufe des Objekts. Hierzu können verschiedene Prozeßbeschreibungsformalismen eingesetzt werden :
  - *Sicherheitsbedingungen* beschränken das mögliche Eintreten von Ereignissen durch Angabe von zwingenden Vorbedingungen.
  - *Deskriptive Lebendigkeitsforderungen* geben Ziele in Form von Ereignissen an, die von einem Objekt langfristig erfüllt werden müssen.
  - Als dritte Möglichkeit kommt die Angabe von *expliziten Prozeßdefinitionen* in Frage.

Ein einzelnes Objekt wird spezifiziert durch die Zuordnung eines eindeutigen Objektnamens und eine *Objektbeschreibung* (engl. 'template'). Eine Objektbeschreibung ist syntaktisch in **Oblog$^+$** wie folgt aufgebaut :

template
  data types *importierte Datentypen;*
  attributes *Attributnamen und -typen;*
  events *Ereignisnamen und -parameter;*
  constraints *Integritätsbedingungen;*
  valuation *Attributänderungen durch Ereignisse;*
  Prozeßspezifikation;

In der Sprache TROLL wird die Prozeßspezifikation durche ein explizites Schlüsselwort **behavior** eingeleitet.

Eine Objektbeschreibung bestimmt die Struktur und Semantik eines Objekts (oder auch mehrerer Objekte). Vereinfacht gesehen, entspricht sie somit von der Konzeption her einer Typbeschreibung in einer Programmiersprache. Ein *Objekt* entspricht dann einer *Variablen* zu einer Typbeschreibung. Zur Deklaration einer derartigen 'Objektvariablen' muß dann — analog zu Variablen einer Programmiersprache — ein Name und eine Objektbeschreibung angegeben werden. Ein einzelstehendes Objekt wird somit wie folgt beschrieben :

object `ObjektName`
  template *Objektbeschreibung;*
end object `ObjektName;`

Um eine Menge gleichartiger Objekte zu beschreiben, werden *Objektklassen* definiert. Analog zu der Unterscheidung zwischen Objektbeschreibungen und Objekten, unterscheiden wir *Klassentypen*[1] und *Objektklassen* :

- Ein *Klassentyp* beschreibt die möglichen Ausprägungen von Objektklassen, liefert also eine *intensionale* Beschreibung von Objektklassen. Zur Definition eines Klassentyps gehört die Festlegung der Struktur möglicher Klassenobjekte sowie die Festlegung eines *Namensraums* zur Unterscheidung der Objekte (Festlegung der Objektidentifikation, siehe [KC86]).

- Eine *Objektklasse* definiert eine Ausprägung einer Klassentypdefinition. Sie gibt also eine *extensionale* Festlegung. Zur Objektklassendeklaration gehört die Festlegung des Klassentyps der Klasse sowie die Angabe des Klassennamens.

Die Analogie von Objekten zu Variablen eines Programmiersprache kann auf Objektklassen ausgeweitet werden, wobei Objektklassen dann Variablen eines mengenwertigen Typs entsprechen. Die Objektidentifikation findet keine direkte Analogie in

[1]Korrekter wäre natürlich die längere Bezeichnung *Objektklassentypen*, auf die aus Lesbarkeitsgründen verzichtet wurde. In der Literatur wird auch die Bezeichnung Objekttyp für Klassentyp verwendet, die aber dem Konzept der Objektbeschreibungen zugeordnet werden sollte ("Typ eines Objekts").

der klassischen Programmiersprachenterminologie, entspricht aber der Festlegung von Schlüsseln für relationale Datenbankschemata.

Um einen Klassentyp zu beschreiben, müssen wir im einfachsten Fall den *Namensraum* (oder auch Surrogatraum) festlegen sowie eine Objektbeschreibung angeben. In späteren Abschnitten werden weitere Möglichkeiten zur Festlegung von Klassentypen diskutiert. Eine Objektklasse wird durch Angabe eines eindeutigen Objektklassennamens sowie der Festlegung eines Klassentyps definiert. Eine Objektklasse kann demnach wie folgt deklariert werden :

> **object class** `ObjektKlassenName`
>   **data types** *importierte Datentypen (für Namensraumfestlegung);*
>   **identification** *Namensraumfestlegung;*
>   **template** *Objektbeschreibung;*
> **end object class** `ObjektKlassenName`;

Wie auch bei Spezifikationen einzelner Objekte, erfolgt die Spezifikation des zu einer Klasse gehörigen Klassentyps in der Regel integriert mit der Definition der Objektklasse. Die konzeptionelle Unterscheidung von intensionaler und extensionaler Beschreibung wird wichtig im Zusammenhang mit Vererbungshierarchien, bei denen die Vererbung auf der Klassentypebene (syntaktische Vererbung) von der Vererbung auf der Klassenebene (semantische Vererbung) unterschieden werden muß (siehe Abschnitt 4.4.1).

3.2 Beispiele für Objektbeschreibungen

Die Konzepte der Beschreibungssprachen **Oblog$^+$** und TROLL sollen anhand einer konkreten Beispielmodellierung erläutert werden. Unsere Beispielwelt, aus der auch in den folgenden Kapiteln weitere Teile modelliert werden, besteht aus Büchern in einer Bibliothek und Benutzern dieser Bibliothek. In der Bibliothek sind Exemplare von Büchern vorhanden, die Bücher selbst sehen wir als abstrakte Objekte an.

Ein Buch wird beschrieben durch seinen Titel, die Liste der Autoren, den Verlag, die Auflage und das Erscheinungsjahr. Bücher werden veröffentlicht und können dann in die Bibliothek aufgenommen werden. Wird ein Buch in die Bibliothek aufgenommen, so wird mindestens ein Exemplar erworben.

3.2.1 Beispiele für Einzelobjekte

Wir beginnen mit der Beschreibung eines abstrakten Buches `book`. Das `book`-Objekt ist konzeptionell ein nicht änderbarer Datenbankeintrag, der die Daten eines Buches (wie Autoren, Titel, etc.) umfaßt. Mit anderen Worten, bei einem `book`-Objekt spielt die dynamische Prozeßkomponente keinerlei Rolle (das einzige Ereignis ist das

Einfügen in die Datenbank) und wir können uns ganz auf die Aspekte der Strukturbeschreibung konzentrieren.

Wie erwähnt, besteht eine Objektdefinition aus der Angabe eines eindeutigen *Objektnamens* (hier `book`) und einer Objektbeschreibung, dem sogenannten *Template* :

```
object book
  template
    Objektbeschreibung;
end object book;
```

Die Objektbeschreibung beginnt mit der Festlegung der importierten Datentypen und der Schnittstelle des Objekts, der sogenannten *lokalen Objektsignatur*. Die lokale Objektsignatur legt die Namen, Wertebereiche und Parametertypen für neu definierte Attribute und Ereignisse fest. Für das Objekt `book` werden diese Angaben wie folgt notiert :

```
  template
    data types nat, string, |PERSON|, list(|PERSON|);
    attributes
      Title: string;
      FirstAuthor: string;
      Authors: list(|PERSON|);
      Publisher: string;
      Edition: nat;
      Year: nat;
      derived NoAuthors: nat;
    events
      birth Published;
```

Das Objekt `book` importiert in diesem Beispiel die Datentypen `nat`, `string`, `|PERSON|` und **list**(`|PERSON|`). Es existieren verschiedene Arten von Datentypen, die importiert werden können :

- Standarddatentypen wie die natürlichen Zahlen `nat`, die ganzen Zahlen `integer`, die Wahrheitswerte `bool`, Datumsangaben `date` oder Zeichenketten `string` stehen in einer Datentypbibliothek fest zur Verfügung und können importiert werden. Eine Auflistung der Standarddatentypen in der Sprache TROLL kann in [JSHS91] gefunden werden.

- Benutzerdefinierte Datentypen (vergleiche Abschnitt 2.3.1) können explizit definiert werden und dann in dem gleichen Spezifikationsdokument benutzt werden.

- Wertebereiche für *Objektidentifikatoren* sind Datentypen, die den Namensraum einer Objektklasse bilden. Ein Beispiel hierfür ist der Namensraum |`PERSON`|, der Werte enthält, die auf Objekte der Klasse `PERSON` zeigen können. Diese Werte können eingekapselt sein, d.h. sie sind nur als uninterpretierbare Werte benutzbar, oder mit Funktionen versehen sein, wie der Namensraum |`PERSON`|, für dessen Werte die Funktion `name` vom Typ `string` definiert ist, die den Namen der Person aus dem Objektidentifikator berechnet.

- Alle importierten Datentypen können als formale Parameter für Datentypkonstruktoren eingesetzt werden. Die Instanzierungen werden in den Sprachen TROLL und **Oblog$^+$** explizit importiert, wie in diesem Beispiel **list**(|`PERSON`|). Alternativ würde es in einer Spezifikationssprache auch ausreichen, den Konstruktor zu importieren und nicht dessen Instanziierungen.

  Standardmäßig stehen in den Sprachen **Oblog$^+$** und TROLL u.a. die Konstruktoren **list** für die Listenbildung, **set** für die Mengenbildung und **record** bzw. **tuple** für die Tupelkonstruktion mit zugehörigen Operationen zur Verfügung.

Die lokale Objektsignatur legt die Schnittstelle des Objekts fest. Für das Objekt `book` haben wir die Attribute Buchtitel `Title`, den ersten Autor `FirstAuthor`, die Liste aller Autoren `Authors`, den Herausgeber `Publisher`, die Auflage `Edition`, das Erscheinungsjahr `Year` und als einziges Ereignis die Publikation `Published`. Das Attribut `Authors` hat als Wertebereich den Type **list**(|`PERSON`|), d.h. Listen von Personenidentifikatoren.

In der lokalen Signatur wird in der Regel nur Name und Typ von Attributen und Ereignissen festgelegt. Die Ausnahmen sind die Angaben, ob ein Ereignis ein neues Objekt kreiert (**birth**) bzw. zerstört (**death**), und ob ein Attribut oder Ereignis von anderen Attributen und Ereignissen durch eine Ableitungsregel abgeleitet wird (**derived**).

Da das **book**-Objekt kein Verhalten zeigt, bleibt als verbleibender Spezifikationsteil die Angabe von *Integritätsbedingungen* zur Einschränkung der Attributausprägungen und die Angabe der Ableitungsregel für das Attribut `NoAuthors` :

```
constraints
   static
      FirstAuthor = name(Authors[1]);
      Year > 1500;
   derivation rules
      NoAuthors = length(Authors);
```

Unter dem Schlüsselwort **static** werden hierbei statische Integritätsbedingungen als Invarianten für Attributwerte angegeben. Die erste Bedingung erzwingt daß der Erstautor auch erstes Element der Autorenliste ist, während die zweite Bedingung eine Einschränkung für Werte des `Year`-Attributs angibt. Dynamische Integritätsbedingungen (vergl. Abschnitt 2.3.3) können auch angegeben werden, ergeben aber bei dem `book`-Objekt natürlich keinen Sinn. Sie werden später anhand anderer Beispiele eingeführt.

Unter dem Schlüsselwort **derivation rules** werden die Ableitungsregeln für abgeleitete Attribute festgelegt, hier wird der Attributwert für die Anzahl der Autoren `NoAuthors` mittels einer mit dem Datentyp **list**(|`PERSON`|) importierten Funktion **length** aus dem Attribut `Authors` berechnet. Die allgemeine Form einer Ableitungsregel lautet :

`AttributTerm =` *Term des passenden Datentyps*;

Bevor wir uns einem etwas 'dynamischeren' Objekt zuwenden, geben wir zusammenfassend die komplette Spezifikation von `book` an.

Beispiel 3.2.1 Die komplette Beschreibung des `book`-Objekts (bis auf die Initialisierung der Attributwerte nach dem Eintritt von `Published`) lautet nun :

```
object book
  template
    data types nat, string, |PERSON|, list(|PERSON|);
    attributes
      Title: string;
      FirstAuthor: string;
      Authors: list(|PERSON|);
      Publisher: string;
      Edition: nat;
      Year: nat;
      derived NoAuthors: nat;
    events
      birth Published;
    constraints
      static
        FirstAuthor = name(Authors[1]);
        Year > 1500;
      derivation rules
        NoAuthors = length(Authors);
end object book;
```

In der Sprache TROLL kann die Initialisierung syntaktisch durch Parameter der Geburtsereignisse erreicht werden, die den gleichen Namen wie die korrespondierenden konstanten Attribute haben, also etwa `Published(Title: string, ..)`. □

Als ein zweites Beispiel betrachten wir ein in der Bibliothek vorhandenes Exemplar eines Buches, das durch das Objekt `copy` modelliert wird.

Betrachten wir wieder erst die Benennung und die lokale Signatur des Objektes :

```
object copy
   template
      data types bool, date, |BOOK|,|USER|,list(|USER|), nat;
      attributes
         DocNo: nat;
         constant Of: |BOOK|;
         OnLoan: bool;
         Due: date;
         Borrowers: list(|USER|);
      events
         birth GetCopy;
         death ThrowAway;
         CheckOut(|USER|, date, nat);
         Return;
      ...
end object copy;
```

Das Attribut `Of` ist ein konstantes Attribut (Schlüsselwort **constant**), das anzeigt von welchem Buch das betreffende Exemplar eine Kopie ist (der Namensraum |`BOOK`| wird im folgenden Abschnitt noch eingehender diskutiert). Das Objekt `copy` hat als weitere Attribute ein boolesches Attribut `OnLoan`, das den aktuellen Ausleihstatus anzeigt, die Angabe des nächsten Rückgabedatums (`Due`) sowie die Liste aller bisherigen Ausleiher (`Borrowers`). Ereignisse des Objekts `copy` sind die Anschaffung eines Exemplars (`GetCopy`), die Ausleihe und Rückgabe des Exemplars (`CheckOut` bzw. `Return`), sowie die endgültige Entfernung aus der Bibliothek (`ThrowAway`). Dem `CheckOut`-Ereignis sind die Einzelheiten der Ausleihe als Parameter beigegeben, hier der Ausleiher, das Ausleihdatum, und die Dauer der Ausleihe.

Wir interessieren uns nun inbesondere für die möglichen dynamischen Entwicklungen eines `copy`-Objekts. Deshalb wurde in der Spezifikation auf die Angabe von Integritätsbedingungen verzichtet.

Der erste Teil der Dynamikbeschreibung beschreibt den Effekt der Ereignisse auf die Attribute in Form von *Auswertungsregeln* nach dem Schlüsselwort **valuation** :

```
valuation
   variables U: |USER|, d: date, n: nat;
   [GetCopy] OnLoan = false;
   [GetCopy] Borrowers = emptylist();
   [CheckOut(U,d,n)] OnLoan = true;
   [CheckOut(U,d,n)] Due =  add_days_to_date(d,n);
```

```
[CheckOut(U,d,n)] Borrowers = append(U,Borrowers);
[Return] OnLoan = false;
```

Die Auswertungsregeln bieten einen auf das explizite Setzen von Attributwerten beschränkten Mechanismus zur Spezifikation von Nachbedingungen für Ereignisse an. So initialisiert das `GetCopy`-Ereignis die Attribute `OnLoan` und `Borrowers`. Das `CheckOut`-Ereignis ändert die Attribute `OnLoan`, `Due` und `Borrowers`, wobei eine Operation des Datentyps `date` benutzt wird, um eine Anzahl von Tagen zu dem Ausleihdatum hinzu zu addieren. **append** und **emptylist** sind Operationen des **list**-Datentypkonstruktors. Das `Return`-Ereignis setzt den Ausleihstatus wieder auf **false** zurück.

Das allgemeine Muster für eine Auswertungsregel ist :

`[EreignisTerm] AttributTerm =` *Term des passenden Datentyps*;

Durch diesen eingeschränkten Aufbau der Auswertungsregeln entsprechen Auswertungsregeln den Zuweisungen in imperativen Programmiersprachen.

Auf die Auswertungsregeln folgt die eigentliche Prozeßbeschreibung. Die *Sicherheitsbedingungen* werden in **Oblog$^+$** nach dem Schlüsselwort **safety** aufgeführt. In der Sprache TROLL lautet das korrespondierende Schlüsselwort **permission**.

Sicherheitsbedingungen sind notwendige Vorbedingungen für Ereignisse und entsprechen somit den *Anwendbarkeitsbedingungen* (enabling conditions) aus [Lip89]. Die Vorbedingung ist als Formel ausgedrückt, die die Einschränkung für das Ereignisauftreten unter Bezug auf

- die aktuellen Attributwerte vor dem Eintreten des Ereignisses, oder
- den internen Zustand des Objekts gegeben durch die Folge der bisherigen Ereignisse ausdrücken kann.

Der interne Zustand des Objekts ist nur implizit durch die bisherige Objektentwicklung festgelegt und kann darum nicht direkt zur Formulierung der Sicherheitsbedingungen herangezogen werden. Als Kalkül zur Formulierung wird eine *vergangenheitsgerichtete temporale Logik* benutzt, die sich auf die gesamte bisherige Objektentwicklung bezieht. Der Bezug auf in der Vergangenheit eingetretene Ereignisse wird durch das **after**-Prädikat ermöglicht, das genau im Zustand nach dem Eintreten des Parameterereignisses den Wert **true** annimmt. Die folgenden Sicherheitsbedingungen sind zwingende Vorbedingungen für das Eintreten von `copy`-Ereignissen :

```
safety
   variables U: |USER|, d: date, n: nat;
   {OnLoan = false} CheckOut(U,d,n);
   {exists(U: |USER|, d: date, n: nat)
      sometime(after(CheckOut(U,d,n)))
         since last (after(Return))} Return;
```

Variablen in Sicherheitsbedingungen sind implizit allquantifiziert. Die erste Regel besagt, daß ein Exemplar nur ausgeliehen werden darf, wenn das `OnLoan`-Attribut aktuell den Wert **false** hat. Die zweite Sicherheitsbedingung ist in temporaler Logik formuliert und besagt, daß ein Exemplar nur zurückgegeben werden kann, wenn es seit dem letzten Ausleihen nicht bereits schon zurückgegeben worden ist[2]. Die temporale Logik wird in den folgenden Abschnitten ausführlich diskutiert und definiert.

Der zweite Anteil der Prozeßbeschreibung von `copy` besteht aus der Formulierung von *Lebendigkeitsforderungen* die in **Oblog+** nach dem Schlüsselwort **liveness** aufgeführt werden. In TROLL lautet das korrespondierende Schlüsselwort **obligation**.

Lebendigkeitsforderungen sind 'zukünftig zu erfüllende Ziele' eines Objekts. Formal bedeutet dies, daß ein Objekt erst gelöscht werden darf, wenn alle seine Lebendigkeitsziele erfüllt sind. Formuliert werden Lebendigkeitsforderungen als Menge von *Livenesstermen* ähnlich regulären Ausdrücken, die von allen korrekten Objektentwicklungen erfüllt werden müssen. Die aktuelle Menge von zu erfüllenden Livenesstermen ist zustandsabhängig, da Ziele bereits erfüllt sein können aufgrund der bisherigen Entwicklung bzw. zustandsabhängig neue Ziele hinzukommen können.

Als Beispiel haben wir folgende (zustandsabhängige) Lebendigkeitsforderung für das `copy`-Objekt :

```
liveness
   {exists(U:|USER|, d:date, n:nat) after(CheckOut(U,d,n)) } ==> Return;
```

Die Forderung besagt, daß ein ausgeliehenes Exemplar auch tatsächlich zurückgegeben werden muß. Das Lebendigkeitsziel 'Ereignis `Return` muß zukünftig stattfinden' wird jeweils im Zustand direkt nach der Ausleihe (gemäß der Vorbedingung **after**(`CheckOut`(U,d,n))) neu in die Menge der offenen Lebendigkeitsforderungen aufgenommen.

Wir beschliessen auch die Diskussion des `copy`-Objekts mit dessen vollständiger Spezifikation.

Beispiel 3.2.2 Das Objekt `copy` wird vollständig wie folgt beschrieben :

```
object copy
   template
      data types bool, date, |BOOK|,|USER|,list(|USER|), nat;
      attributes
         DocNo: nat;
         constant Of: |BOOK|;
         OnLoan: bool;
         Due: date;
```

[2] Auch diese Bedingung liesse sich eleganter mit Zugriff auf das `OnLoan`-Attribut formulieren. Darauf wurde verzichtet, um ein Beispiel für den Einsatz temporaler Logiken zu geben.

```
      Borrowers: list(|USER|);
   events
      birth GetCopy;
      death ThrowAway;
      CheckOut(|USER|, date, nat);
      Return;
   valuation
      variables U: |USER|, d: date, n: nat;
      [GetCopy] OnLoan = false;
      [GetCopy] Borrowers = emptylist();
      [CheckOut(U,d,n)] OnLoan = true;
      [CheckOut(U,d,n)] Due =  add_days_to_date(d,n);
      [CheckOut(U,d,n)] Borrowers = append(U,Borrowers);
      [Return] OnLoan = false;
   safety
      variables U: |USER|, d: date, n: nat;
      {OnLoan = false} CheckOut(U,d,n);
      {exists(U: |USER|, d: date, n: nat)
         sometime(after(CheckOut(U,d,n))) since last
            (after(Return))} Return;
   liveness
      {exists(U: |USER|, d: date, n: nat) after(CheckOut(U,d,n)) }
         ==> Return;
end object copy;
```

□

Mit der vollständigen Spezifikation von copy beenden wir die Diskussion der Beschreibung einzelner isolierter Objekte.

3.2.2 Beispiele für Objektklassen

Die Basisidee für den Schritt von Einzelobjekten zu Objektklassen ist die Benutzung eines Objektes als *Prototyp* für andere Objekte. In **Oblog**$^+$ kann dies direkt durch syntaktische Wiederverwendung der Objektbeschreibung geschehen, notiert wie folgt :

```
object book2
   template book.template
end object book2;
```

Dieses Sprachmittel ermöglicht (in Analogie zu Programmiersprachen gesprochen) eine endliche Menge *explizit benannten* Objektvariablen zur selben Objektbeschreibung. Jedem dieser Objekte muß im *Spezifikationsdokument* ein eindeutiger Namen zugeordnet werden. Dieser Ansatz ist für die formale Spezifikation von Informationssystemen, die in der Regel Daten über viele, oft erst zur Laufzeit bekannte, Objekte einer Art bereit halten, nicht geeignet.

Ziel der Einführung von *Objektklassen* ist es, einen möglicherweise potentiell unendlichen Namensraum für Objekte einer Objektbeschreibung festzulegen, zu dessen Namenwerten dann zur *Systemlaufzeit* Objekte generiert werden können. In unserer Beispielmodellierung wollen wir ja zu jeder Kombination Titel und Erstautor einen Bucheintrag generieren können. Ein derartiger Mechanismus wird durch die Einführung von *Klassentypen* und *Objektklassen* zur Verfügung gestellt.

Wir beginnen die Diskussion von Klassentypen und Objektklassen mit der Erweiterung des `book`-Beispiels auf die Spezifikation der Objektklasse `BOOK`, die die Klasse der in der Bibliothek registrierten Bucheinträge beschreibt.

Der Namensraum einer Objektklasse wird in den behandelten Sprachvorschlägen nach dem Schlüsselwort **identification** (in TROLL) oder **surrogate** (in **Oblog$^+$**) festgelegt. Hier kann ein beliebiger Datentyp angegeben werden, dessen Werte als Namensraum für den Klassentyp bestimmt sind. Als Schreibkonvention wurde eine der Attributangabe ähnliche Notation gewählt, die der Schlüsselangabe für relationale Datenbankschemata entspricht.

Für die Klasse `BOOK` wird der Namensraum wie folgt festgelegt, in dem die beiden Attribute `Title` und `FirstAuthor` als Objektidentifikatoren festgelegt werden. Natürlich müssen auch für die Namensraumfestlegung die benötigten Datentypen explizit importiert werden :

```
object class BOOK
   data types string;
   identification
      Title: string;
      FirstAuthor: string;
   template ...
end object class BOOK;
```

Der Datentyp des Namensraum einer Objektklasse ist jeweils durch eine Tupelkonstruktion festgelegt, hier also

$$|\mathtt{BOOK}| := \textbf{tuple}(\mathtt{Title{:}\ string,\ First\_Author{:}\ string}).$$

Die Tupelselektoren derartiger zusammengesetzter Namensräume können in der Spezifikation wie gewöhnliche Attribute verwendet werden. Die Namensräume können in Objektspezifikationen wie andere Datentypen importiert werden und dann als Zeiger auf Objektinstanzen dieser Klassen dienen. Die Attributnamen der verwendeten Schlüsselattribute sind dann als Funktionen auf Werten dieser Datentypen definiert (analog zur Tupelkonstruktion als parametrisierten Datentypkonstruktor).

Auf die Festlegung des Namensraumes folgt die Beschreibung der Objekte der Klasse in Form eines Objekttemplates. Die komplette Spezifikation der Klasse `BOOK` lautet nun :

Beispiel 3.2.3 Die Objektklasse `BOOK` wird wie folgt spezifiziert :

```
object class BOOK
   data types string;
   identification
      Title: string;
      FirstAuthor: string;
   template
      data types nat, string, |PERSON|, list(|PERSON|);
      attributes
         Authors: list(|PERSON|);
         Publisher: string;
         Edition: nat;
         Year: nat;
         derived NoAuthors: nat;
      events
         birth Published;
      constraints
         static
            FirstAuthor = name(Authors[1]);
            Year > 1500;
         derivation rules
            NoAuthors = length(Authors);
end object class BOOK;
```

□

Als zweites Beispiel betrachten wir die Objektklasse `COPY`. Der Namensraum wird hierbei durch den Wert des Attributs `DocNo` festgelegt. Zusätzlich wird das konstante Attribut `Of` ebenfalls bei der Spezifikation des Namensraumes mit eingebracht. Auf diese — im ersten Anschein wenig motivierte — Integration kontanter Attribute in die Namensraumfestlegung wird im folgenden Kapitel noch eingegangen werden.

Beispiel 3.2.4 Die Objektklasse `COPY` wird wie folgt beschrieben :

```
object class COPY
   data types nat, |BOOK|;
   identification
      DocNo: nat;
   constants
```

```
        Of: |BOOK|;
    template
        data types bool, date, |USER|, list(|USER|), nat;
        attributes
            OnLoan: bool;
            Due: date;
            Borrowers: list(|USER|);
        events
            birth GetCopy;
            death ThrowAway;
            CheckOut(|USER|, date, nat);
            Return;
        valuation
            variables U: |USER|, d: date, n: nat;
            [GetCopy] OnLoan = false;
            [GetCopy] Borrowers = emptylist();
            [CheckOut(U,d,n)] OnLoan = true;
            [CheckOut(U,d,n)] Due =  add_days_to_date(d,n);
            [CheckOut(U,d,n)] Borrowers = append(U,Borrowers);
            [Return] OnLoan = false;
        safety
            variables U: |USER|, d: date, n: nat;
            {OnLoan = false} CheckOut(U,d,n);
            {exists(U: |USER|, d: date, n: nat)
                sometime(after(CheckOut(U,d,n))) since last
                    (after(Return))} Return;
        liveness
            {exists(U: |USER|, d: date, n: nat) after(CheckOut(U,d,n)) }
                ==> Return;
end object class COPY;
```

□

Nach den motivierenden Beispielen zur Objektspezifikation wird in den folgenden Abschnitten die formale Grundlage der diskutierten objektorientierten Spezifikationssprachen entwickelt.

3.3 Semantische Modellbildung für Objekte

In diesem Abschnitt wird ein formales Modell für objektorientierte Spezifikationen vorgestellt, das Objekte eines Systems als beobachtbare Prozesse modelliert. Die Modellbildung basiert auf Algebren als Basiswertebereiche für Beobachtungen und Ereignisparameter, die im ersten Teilabschnitt formal eingeführt werden.

3.3.1 Signaturen und Algebren

Algebren sind die mathematischen Modellbildungen für abstrakte Wertebereiche mit zugehörigen Operationen. Algebren sind somit die semantische Modellbildung für *abstrakte Datentypen* [EM85, EGL89]. Die *Signatur* einer (mehrsortigen) Algebra bestimmt die Namen von Wertebereichen und Operationen auf Werten.

Definition 3.3.1 Eine *Signatur* $\Sigma = (S, \Omega, \Pi)$ besteht aus

1. einer Menge $S = \{s, r, \ldots\}$ von Sorten,
2. einer Menge $\Omega = \{f\colon s_1 \times \ldots \times s_n \to s_0,\ g\colon r_1 \times \ldots \times r_m \to r_0, \ldots\}$ von Funktionssymbolen mit ≥ 0 Argumentsorten und einer Wertsorte,
3. und einer Menge $\Pi = \{p\colon s_1 \times \ldots \times s_n,\ q\colon r_1 \times \ldots \times r_m, \ldots\}$ von Prädikatensymbolen mit ≥ 0 Argumentsorten. □

Für alle Sorten $s \in S$ ist ein Prädikatensymbol $=_s : s \times s$ mit fester Interpretation vorgegeben und in der Signatur implizit als zweistelliges Prädikatensymbol enthalten. Dieses *Gleichheitsprädikat* (bzw. seine Semantik) ist definiert durch die Gleichheit auf den Trägermengen, d.h. den Interpretationen der Sortensymbole. In der Notation von Formeln wird in der Regel $=$ statt $=_s$ verwendet, sofern die Sorte s nicht extra hervorgehoben werden soll.

Beispiel 3.3.2 Die Signatur $\Sigma_{\mathtt{nat}}$ der natürlichen Zahlen kann wie folgt angegeben werden:

$$\begin{array}{rcl}
S_{\mathtt{nat}} & = & \{\mathtt{nat}\} \\
\Omega_{\mathtt{nat}} & = & \{\mathtt{zero}\colon \to \mathtt{nat}, \mathtt{succ}\colon \mathtt{nat} \to \mathtt{nat}, \\
 & & \ \mathtt{plus}\colon \mathtt{nat} \times \mathtt{nat} \to \mathtt{nat}, \mathtt{times}\colon \mathtt{nat} \times \mathtt{nat} \to \mathtt{nat}, \ldots\} \\
\Pi_{\mathtt{nat}} & = & \{\mathtt{equal}\colon \mathtt{nat} \times \mathtt{nat}, \ldots\}
\end{array}$$

Konstantensymbole (hier `zero`) werden durch nullstellige Funktionen modelliert. □

Signaturen legen eine Sprache fest, in der über Algebren einer bestimmten Struktur gesprochen werden kann. Eine gegebene Algebra zu einer Signatur Σ wird oft auch als (mögliche) *Interpretation* dieser Signatur bezeichnet.

Definition 3.3.3 Eine *Interpretation* I einer Signatur $\Sigma = (S, \Omega, \Pi)$ ist eine Algebra (genannt auch Σ*–Struktur*) $I(\Sigma) = (I(S), I(\Omega), I(\Pi))$, die definiert ist durch

1. $I(S)$ bestehend aus einer nichtleeren Trägermenge $I(s)$ für jede Sorte $s \in S$,

2. $I(\Omega)$ bestehend aus einer Funktion $I(f): I(s_1) \times \ldots \times I(s_n) \to I(s_0)$ für jedes Funktionssymbol $f: s_1 \times \ldots \times s_n \to s_0 \in \Omega$,

3. und $I(\Pi)$ bestehend aus einer Relation $I(p) \subseteq I(s_1) \times \ldots \times I(s_n)$ für jedes Prädikatensymbol $p: s_1 \times \ldots \times s_n \in \Pi$.

Die Menge aller Σ–Strukturen für eine Signatur Σ wird als Menge der *möglichen Interpretationen* bezeichnet. □

Beispiel 3.3.4 Das in Beispiel 3.3.2 angegebene Fragment der Signatur $\Sigma_{\mathtt{nat}}$ kann durch die Interpretation $I_{\mathtt{nat}}$ der natürlichen Zahlen $\mathbb{N}$ interpretiert werden :

$$\begin{aligned}
I_{\mathtt{nat}}(\mathtt{nat}) &= \mathbb{N} \\
I_{\mathtt{nat}}(\mathtt{succ}) &= +1: \mathbb{N} \to \mathbb{N} \\
I_{\mathtt{nat}}(\mathtt{zero}) &= 0: \to \mathbb{N} \\
I_{\mathtt{nat}}(\mathtt{add}) &= +: \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\
I_{\mathtt{nat}}(\mathtt{times}) &= *: \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\
I_{\mathtt{nat}}(\mathtt{equal}) &= =: \mathbb{N} \times \mathbb{N}
\end{aligned}$$

Diese (intuitiv naheliegende) Interpretation ist allerdings nur eine von vielen möglichen Interpretationen. Eine andere (nicht so naheliegende) Interpretation $I'_{\mathtt{nat}}$ wäre

$$\begin{aligned}
I'_{\mathtt{nat}}(\mathtt{nat}) &= \mathbf{boolean} \\
I'_{\mathtt{nat}}(\mathtt{zero}) &= \mathbf{true}: \to \mathbf{boolean} \\
I'_{\mathtt{nat}}(\mathtt{succ}) &= \neg: \mathbf{boolean} \to \mathbf{boolean} \\
I'_{\mathtt{nat}}(\mathtt{add}) &= \wedge: \mathbf{boolean} \times \mathbf{boolean} \to \mathbf{boolean} \\
I'_{\mathtt{nat}}(\mathtt{times}) &= \vee: \mathbf{boolean} \times \mathbf{boolean} \to \mathbf{boolean} \\
I'_{\mathtt{nat}}(\mathtt{equal}) &= =: \mathbf{boolean} \times \mathbf{boolean}
\end{aligned}$$

wobei die interpretierende Algebra durch die Algebra der Wahrheitswerte **boolean** definiert wird. Aufgabe der *algebraischen Spezifikation* ist es, für eine gegebene Signatur die intendierte Interpretationsstruktur möglichst eindeutig zu beschreiben und damit bis auf Isomorphie eindeutig festzulegen. □

In der üblichen Notation der algebraischen Spezifikation bestehen Algebren in der Regel nur aus Mengen und Funktionen. Prädikate werden dann als `Boolean`-wertige Funktionen modelliert. Die Prädikate wurden hier explizit in die Definition von Algebren und Signaturen mit aufgenommen, da diese bei der Definition der Prädikatenlogik für Algebren eine besondere Rolle spielen.

3.3.2 Ein formales Objektmodell : Objekte als beobachtbare Prozesse

In diesem Abschnitt soll eine formale Modellbildung für Objektspezifikationen entwickelt werden. Intuitiv sind Objekte beobachtbare Prozesse, d.h. für einzelne Objekte Folgen von Ereignissen und Beobachtungen der Attributwerte.

Beispiel 3.3.5 Wir betrachten ein einfaches Objekt `konto` mit Attributen `Guthaben` und `Überzogen` sowie Änderungsereignissen `Create`, `Einzahlung` und `Abhebung`. Die lokale Objektsignatur von `konto` lautet :

```
template
   data types integer, boolean;
   attributes
      Guthaben: integer;
      Überzogen: boolean;
   events
      birth Eröffnung;
      Einzahlen(integer);
      Abheben(integer);
```

Die zeitliche Entwicklung eines `konto`-Objekts kann nun als Tabelle aufgeschrieben werden, in der in der ersten Spalte die auftretenden Ereignisse und in den folgenden Spalten die Entwicklung der Attributwerte aufgelistet wird. Eine mögliche zeitliche Entwicklung ist in Abbildung 3.1 abgebildet.

Lebenslauf	Beobachtungen :	`Guthaben`	`Überzogen`
`Eröffnung`;		0	**false**
`Einzahlen`(50);		50	**false**
`Einzahlen`(100);		150	**false**
`Abheben`(200);		-50	**true**
...		...	...

Abbildung 3.1: Lebenslauf und Beobachtungen einer möglichen Entwicklung des `konto`-Objekts.

□

Einer Folge von Ereignissen des Objekts ist durch die zustandsabhängigen Attributwerte eine Folge von Attributbelegungen zugeordnet. In der semantischen Modellbildung kann dies durch eine Funktion modelliert werden, die korrekte Anfangsstücke von Ereignissequenzen auf Attribut-Wert-Paare abbildet.

Die einfache Modellierung von Objektentwicklungen muß erweitert werden, wenn auch das *gleichzeitige* Auftreten von Ereignissen modelliert werden soll. Dies kann dadurch erreicht werden, daß wir Folgen von Mengen *gleichzeitig eintretender Ereignisse* betrachten.

Beispiel 3.3.6 Betrachten wir hierzu in unserem `konto`-Beispiel die Bedingung, daß das Einrichten eines Kontos immer mit einer gleichzeitigen Einzahlung von 5.- DM verbunden ist. Der erste Eintrag unserer neuen Tabelle 3.2 ist nunmehr eine Menge von zwei gleichzeitig eintretenden Ereignissen. □

Lebenslauf	Beobachtungen :	`Guthaben`	`Überzogen`
{`Eröffnung`,`Einzahlen`(5)};		5	**false**
{`Einzahlen`(50)};		55	**false**
{`Einzahlen`(100)};		155	**false**
{`Abheben`(200)};		−45	**true**
...		...	...

Abbildung 3.2: Mögliche Entwicklung des `konto`-Objekts mit gleichzeitig eintretenden Ereignissen.

Mit der Möglichkeit, mehrere Ereignisse gleichzeitig eintreten zu lassen, haben wir die semantische Modellbildung verallgemeinert und somit einem weiteren Einsatzspektrum angepaßt. Aus ähnlichen Erwägungen können wir ebenfalls die Modellbildung für Attributbeobachtungen verallgemeinern, indem wir auch hier Mengen von Attribut-Wert-Paaren zulassen. Diese Erweiterung ermöglicht die direkte semantische Modellierung folgender Situationen :

- Eine leere Menge von Beobachtungen für ein bestimmtes Attribut modelliert auf direkte Art ein nichtinitialisiertes Attribut.
- Mehrere Beobachtungen für das selbe Attribut modellieren nicht vollständig determinierte Attributwerte, die insbesondere in frühen Entwurfsphasen eine Rolle spielen. Beispiele hierfür sind semantische Modelle für Integritätsbedingungen wie
  - "Direkt nach der Kontoeröffnung muß das Konto ein Guthaben von mindestens 5.- DM aufweisen."
  - "Einem eingefügten Auto muß eine Registriernummer aus dem Bereich A.100.000 bis A.499.999 zugeordnet werden, die bisher noch nicht vergeben wurde."

Im Gegensatz zu gleichzeitig auftretenden Ereignissen, werden nicht vollständig determinierte Attribute in den folgenden Abschnitten keine Rolle spielen. Nach diesen einführenden Bemerkungen, werden nun semantische Strukturen definiert, die Objekte als *beobachtbare Prozesse* darstellen.

Definition 3.3.7 Wir beginnen mit der Definition einiger grundlegender Begriffe.

- X ist eine aufzählbare Menge von Ereignissen. Die Menge X setzt sich aus den Einfüge-Ereignissen X_b (birth events), den Änderungsereignissen X_u (update events) und den Löschereignissen X_d (death events) zusammen :

$$X = X_b \cup X_u \cup X_d$$

  Die Mengen X_b, X_u und X_d sind paarweise disjunkt.

- Mit $S(X)$ werden nichtleere Mengen von gleichzeitig eintretenden Ereignissen bezeichnet, auch *Schnappschüsse (snapshots) über* X genannt, d.h. $S(X)$ ist definiert durch die Potenzmenge über X:

$$S(X) = 2^X - \{\}$$

- Mit $\mathcal{LC}_X$ bezeichnen wir die Menge der möglichen *Lebensläufe* über X, d.h.

$$\mathcal{LC}_X = S(X)^* \cup S(X)^\omega$$

  wobei $S(X)^*$ die endlichen Folgen von Schnappschüssen bezeichnet und $S(X)^\omega$ für die unendlichen Folgen steht. Auf den n-ten Schnappschuß einer Folge kann mit der Notation s_n zugegriffen werden.

□

Mit Hilfe dieser grundlegenden Begriffe können wir jetzt *Prozesse* über einem Ereignisalphabet definieren.

Definition 3.3.8 Ein *Prozeß* P ist definiert durch ein Paar (X, Λ) mit

- X ist eine Menge von Ereignissen, und
- $\Lambda \subseteq \mathcal{LC}_X$ ist die Menge der erlaubten Lebensläufe mit $\epsilon \in \Lambda$. ϵ ist die leere Folge von Schnappschüssen.

Des weiteren müssen die folgenden Bedingungen für jedes $\lambda \in \Lambda$ erfüllt sein :

$$\begin{array}{rcl}
\lambda = \langle s_1, \ldots \rangle & \Rightarrow & \exists (e \in X_b) \colon e \in s_1 \\
\lambda = \langle s_1, \ldots, s_n \rangle & \Rightarrow & (\exists (e \in X_d) \colon e \in s_n) \wedge (\forall (i < n) \colon \neg \exists (e \in X_d) \colon e \in s_i) \\
\lambda \in S(X)^\omega & \Rightarrow & \forall (n) \colon \neg \exists (e \in X_d) \colon e \in s_n
\end{array}$$

Dies drei Bedingungen sichern die folgenden drei erwünschten Einschränkungen an Objektlebensläufe zu:

1. Lebensläufe beginnen mit einem Geburtsereignis.
2. Endliche Lebensläufe enden mit einem Todesereignis. Vor dem letzten Zustand treten keine Todesereignisse auf.
3. Unendliche Lebensläufe enthalten keine Todesereignisse. □

Bis jetzt haben wir nur die Prozeßkomponente unseres Objektmodells betrachtet. Im nächsten Schritt definieren wir nun die *Beobachtungen* von Attributen über einem Prozeß.

Definition 3.3.9 Wir beginnen wieder mit der Definition einiger grundlegender Begriffe.

- A ist eine aufzählbare Menge von Attributen.
- Jedem Attribut $a \in A$ ist ein Datentyp $\mathbf{sort}(a)$ zugeordnet.
- Die Datentypen der Attribute werden durch eine Basisdatenalgebra B interpretiert. Die Trägermenge von $\mathbf{sort}(a)$ wird als $B(\mathbf{sort}(a))$ notiert.
- Mit $\mathbf{obs}(A)$ bezeichnen wir die Menge aller möglichen *Beobachtungen* über Attributen A. Die Menge $\mathbf{obs}(A)$ ist definiert als

$$\mathbf{obs}(A) = \{(a, d) | a \in A \wedge d \in B(\mathbf{sort}(a))\}$$

- Mit $\mathbf{Prefix}(L)$ für ein $L \subseteq \mathcal{LC}_X$ bezeichnen wir die Menge aller endlichen Präfixe von Lebensläufen $\lambda \in L$. Ein gebener Präfix wird oft auch als *Zustand* eines Objektes bzw. Prozesses bezeichnet. □

Die Menge der Attribute A muß aufzählbar, aber nicht unbedingt endlich sein. Analog zu Ereignissen können auch Attributdeklarationen mit Parametern versehen werden, so daß eine unendliche Menge A entstehen kann. Ein Beispiel dafür wäre eine Attributdeklaration

```
Jahresgehalt(integer) : money;
```

bei der das Jahresgehalt für unterschiedliche Jahre gespeichert wird. In diesem Fall wird von syntaktischen *Attributgeneratoren* gesprochen.

Definition 3.3.10 Eine *Beobachtungsstruktur* V über einem Prozeß $P = (X, \Lambda)$ ist ein Paar (A, α) mit

- A ist eine Menge von Attributen, und
- α ist eine *Beobachtungsfunktion* über A,

$$\alpha\colon \mathbf{Prefix}(\Lambda) \to 2^{\mathbf{obs}(A)}$$

mit der zusätzlichen Bedingung $\alpha(\epsilon) = \emptyset$. □

Die Beobachtungsfunktion ordnet einem Lebenszykluspräfix, d.h. einem konkreten Objektzustand, eine Untermenge der möglichen Beobachtungen zu. Die Abbildung α hat als Bildbereich also die Potenzmenge $2^{\mathbf{obs}(A)}$ über der Menge der möglichen Attribut/Werte-Paare.

Mit Hilfe der bisherigen Definitionen kann jetzt die semantische Modellbildung für Objekte definiert werden.

Definition 3.3.11 Ein *Objektmodell ob* $= (P, V)$ besteht aus einem Prozeß P und einer Beobachtungsstruktur V über P. □

Diese semantische Modellbildung erfüllt die aufgestellten Forderungen nach Ausdrückbarkeit der Konzepte gleichzeitig auftretender Ereignisse sowie nicht initialisierter und nicht determinierter Attributwerte. Das formale Modell wird nun anhand einer Erweiterung der Spezifikation des `konto`-Objekts (Beispiel 3.3.5) weiter erläutert.

Beispiel 3.3.12 Die Menge der Ereignisse $X_{\texttt{konto}}$ ist definiert durch

$$\begin{aligned} X_{\texttt{konto}} &= \{\texttt{Eröffnung}\} \cup \{\texttt{Einzahlen}(i) | i \in \mathbf{integer}\} \\ &\quad \cup \{\texttt{Abheben}(i) | i \in \mathbf{integer}\} \cup \{\texttt{SchließeKonto}\} \end{aligned}$$

Wir haben hier zusätzlich ein Sterbeereignis `SchließeKonto` eingeführt, um auch endliche Lebensläufe im Beispiel aufführen zu können. Die Menge $S(X_{\texttt{konto}})$ der Schnappschüsse über $X_{\texttt{konto}}$ enthält u.a. die folgenden Elemente :

$$S(X_{\texttt{konto}}) = \{\{\}, \{\texttt{Einzahlen}(3)\}, \{\texttt{Eröffnung}, \texttt{Einzahlen}(5)\}, \ldots\}$$

Ein möglicher (endlicher) Lebenslauf aus $\Lambda_{\texttt{konto}}$ ist dann

$$\langle\{\texttt{Eröffnung}\}, \{\texttt{Einzahlen}(50)\}, \{\texttt{Abheben}(45), \texttt{SchließeKonto}\}\rangle \in \Lambda_{\texttt{konto}}$$

Die Menge der Attribute $A_{\texttt{konto}}$ ist gegeben durch

$$A_{\texttt{konto}} = \{\texttt{Guthaben}, \texttt{Überzogen}\}$$

mit

$$\mathbf{sort}(\texttt{Guthaben}) = \mathbf{integer}, \mathbf{sort}(\texttt{Überzogen}) = \mathbf{boolean}$$

und den naheliegenden Trägermengen für **integer** und **boolean**. Die einzelnen Präfixe $\mathbf{Prefix}(\Lambda_{\texttt{konto}})$ enthalten aufgrund des obigen Beispiellebenslaufs u.a. den leeren Lebenslauf ϵ und die folgenden beiden Sequenzen

$$\mathbf{Prefix}(\Lambda_{\texttt{konto}}) \supseteq \{\epsilon, \langle\{\texttt{Eröffnung}\}\rangle, \langle\{\texttt{Eröffnung}\}, \{\texttt{Einzahlen}(50)\}\rangle\}$$

die durch $\alpha_{\texttt{konto}}$ wie folgt interpretiert werden :

$$\begin{array}{cccc} & \epsilon & \langle\{\texttt{Eröffnung}\}\rangle & \langle\{\texttt{Eröffnung}\}, \{\texttt{Einzahlen}(50)\}\rangle \\ \alpha_{\texttt{konto}} & \downarrow & \downarrow & \downarrow \\ & \emptyset & \{(\texttt{Guthaben}, 0), & \{(\texttt{Guthaben}, 50), \\ & & (\texttt{Überzogen}, \mathbf{false})\} & (\texttt{Überzogen}, \mathbf{false})\} \end{array}$$

Der leere Lebenslauf ϵ wird hierbei — wie immer — durch die leere Beobachtung $\emptyset$ interpretiert. □

Nach der Diskussion der semantischen Strukturen, die Objekten zu Grunde liegen, werden im nächsten Abschnitt Logikkalküle eingeführt, die Aussagen über derartige semantische Modelle ermöglichen.

3.4 Basiskonzepte der Spezifikationskalküle

Spezifikationskalküle für Objekte basieren auf Teilsprachen formaler Logikkalküle, die Aussagen über Werte, Objektzustände und Objektentwicklungen ermöglichen. In den folgenden Unterabschnitten werden zuerst die Termbildung über Signaturen und die Prädikatenlogik als Grundlage für Aussagen über Werte und Objektzustände eingeführt, auf denen dann *temporale Logiken* zur Spezifikation von Objektentwicklungen aufbauen. Gleichzeitig werden die entsprechenden Interpretationsstrukturen definiert. Der Abschnitt endet mit der Beziehung der logischen Interpretationsstrukturen zu den im vorigen Abschnitt definierten Objektmodellen.

3.4.1 Signaturen und Termbildung

Unter einer *Signatur* (vergl. Abschnitt 3.3.1) versteht man die Menge der Symbole für Sorten, Werte und Funktionen, die in einer logischen Spezifikation neben den 'logischen' Symbolen (wie $\wedge$, $\neg$ oder $\exists$) anwendungsspezifisch eingeführt werden. Eine Signatur definiert somit den 'anwendungsspezifischen' Teil des Alphabets, über dem logische Formeln gebildet werden können.

Über einer gegebenen Signatur σ können als elementare Ausdrücke *Terme* gebildet werden, indem Funktionssymbole auf Teilterme angewendet werden. Die Bildung von Termen und ihre Auswertung entspricht den üblichen mathematischen Konventionen.

Sei zusätzlich zu der Signatur $\Sigma = (S, \Omega, \Pi)$ eine Menge von sortengebundenen Variablen $Y = \{x\colon s_1,\, y\colon s_2, \ldots\}$ mit $s_i \in S$ gegeben. Die Menge aller Terme zu einer Signatur Σ und einer Variablenmenge Y wird als $T_\Sigma(Y)$ bezeichnet.

Definition 3.4.1 Die *Terme* einer Sorte s in $T_\Sigma(Y)$ sind definiert durch :

1. Jede Variable $x\colon s$ ist ein Term der Sorte s.
2. Sind $t_1\colon s_1, \ldots, t_n\colon s_n$ Terme der Sorten $s_1, \ldots, s_n$ und ist $f\colon s_1 \times \ldots \times s_n \to s$ ein Funktionssymbol, so ist $f(t_1\colon s_1, \ldots, t_n\colon s_n)\colon s$ ein Term der Sorte s.
3. Keine anderen Zeichenketten sind Terme in $T_\Sigma(Y)$. □

Für eine gegebene Algebra $A(\Sigma) = (A(S), A(\Omega), A(\Pi))$ wird eine Belegung der Variablen aus Y durch eine Belegungsfunktion β festgelegt.

Definition 3.4.2 Eine *Belegung* β der Variablen in Y ist eine Funktion

$$\beta\colon Y \to \bigcup_{s \in S} A(s)\ ,$$

die jeder Variablen $x\colon s \in Y$ einen Wert $\beta(x\colon s) \in A(s)$ aus der Trägermenge der passenden Sorte zuordnet. □

Ist die Interpretation der Signatur und die Belegung der Variablen bestimmt, kann die Auswertung von Termen rekursiv über den syntaktischen Aufbau definiert werden. Dies entspricht der intuitiven Termauswertung, bei der der Gesamtwert eines Ausdrucks ausgehend von den kleinsten Bestandteilen, deren Wert direkt bestimmt ist, errechnet wird. In den folgenden Definitionen wird der Übersicht halber auf die explizite Sortenangabe bei Variablen und Termen verzichtet.

Definition 3.4.3 Für eine gegebene Algebra A und eine Belegung der Variablen β ist die *Termauswertung* definiert durch

$$\begin{aligned}(A,\beta)[\![x]\!] &= \beta(x) \text{ für } x \in Y \\ (A,\beta)[\![f(t_1,\ldots,t_n)]\!] &= A(f)((A,\beta)[\![t_1]\!],\ldots,(A,\beta)[\![t_n]\!])\end{aligned}$$

Der Ausdruck $(A,\beta)[\![t]\!]$ bezeichnet hierbei den Wert des Termes t bei gegebener Algebra A und Belegung β. Die Auswertung von Konstanten als nullstellige Funktionen ist als Spezialfall enthalten. □

3.4.2 Prädikatenlogik

Die Menge der Prädikatensymbole Π legt die elementaren Aussagen eines Prädikatenkalküls fest. Diese elementaren Aussagen werden als *atomare Formeln* bezeichnet. Die Menge der atomaren Formeln zu einer Signatur Σ und einer Variablenmenge Y wird mit $AF_\Sigma(Y)$ bezeichnet.

Definition 3.4.4 Die *atomaren Formeln* in $AF_\Sigma(Y)$ sind definiert durch

1. Sind $t_1{:}\, s$ und $t_2{:}\, s$ Terme der gleichen Sorte s aus $T_\Sigma(Y)$, so ist $t_1{:}\, s = t_2{:}\, s$ eine atomare Formel.

2. Sind $t_i{:}\, s_i$ Terme der Sorten s_i aus $T_\Sigma(Y)$ und ist $p{:}\, s_1 \times \ldots \times s_n$ ein Prädikatensymbol, so ist $p(t_1{:}\, s_1,\ldots,t_n{:}\, s_n)$ eine atomare Formel.

3. Keine anderen Zeichenreihen sind atomare Formeln in $AF_\Sigma(Y)$. □

Der Punkt 1. ist zwar durch die Bemerkung nach der Definition 3.3.1 bereits im zweiten Punkt mit erfaßt, wird aber aufgrund der besonderen Rolle der Gleichheitsprädikate gesondert aufgeführt.

Basierend auf den atomaren Formeln kann nun die Menge der Formeln der *Prädikatenlogik* definiert werden. Die Menge der Formeln der Prädikatenlogik über einer Variablenmenge Y zu einer Signatur Σ wird mit $P_\Sigma(Y)$ bezeichnet.

Definition 3.4.5 Die *Formeln* der Prädikatenlogik in $P_\Sigma(Y)$ sind definiert durch :

1. Die Symbole **true** und **false** sind Formeln.

2. Jede atomare Formel in $AF_\Sigma(Y)$ ist eine Formel in $P_\Sigma(Y)$.

3. Sind φ, ψ Formeln in $P_\Sigma(Y)$ und $x \in Y$ eine Variable der Sorte $s \in S$, so sind auch $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$, $(\varphi \Rightarrow \psi)$, $(\varphi \iff \psi)$, $\neg\varphi$, $(\exists x{:}\, s)\varphi$ und $(\forall x{:}\, s)\varphi$ Formeln in $P_\Sigma(Y)$.

4. Keine anderen Zeichenreihen sind Formeln in $P_\Sigma(Y)$. □

Um die Lesbarkeit von Formeln zu erhöhen, werden die üblichen logischen Konjektive eingeführt, obwohl sie zum Teil auch als Abkürzungen für andere Formeln definiert werden können, so z.B. $(\varphi \vee \psi)$ als $\neg(\neg\varphi \wedge \neg\psi)$ oder **false** als $\neg$**true**. Eine für den Prädikatenkalkül ausreichende Menge von logischen Symbolen wäre z.B. $\neg$, $\wedge$ und $(\forall x\colon s)$.

Ebenfalls zur Erhöhung der Lesbarkeit von Signaturen und Formeln werden weitere Abkürzungskonventionen eingeführt. Bei Konstanten $k\colon \to s$ in Ω kann das "$\to$"–Symbol in der Signatur weggelassen werden. In Termen werden die Sortenangaben weggelassen, falls keine Mißverständnisse zu erwarten sind. Klammern werden zur Strukturierung gesetzt und zur Verbesserung der Lesbarkeit weggelassen. Hierbei gelten die üblichen Vorrangregeln für die logischen Konnektive (die Operatoren $\neg$, $(\exists x\colon s)$ und $(\forall x\colon s)$ binden stärker als alle binären Operatoren, $\wedge$ bindet stärker als $\vee$, etc.).

Die bisherigen Definitionen legen die Syntax des Prädikatenkalküls fest. Die Zuordnung der Wahrheitswerte zu den Formeln wird als *Semantik der Formeln* bezeichnet. Der Wahrheitswert einer Formel hängt von der Interpretation der Signatur Σ durch eine Algebra (oder auch Σ–Struktur) und der Belegung der Variablen aus Y ab.

Aufbauend auf der Termauswertung kann nun der Wahrheitswert einer Formel definiert werden. Eine Formel heißt *gültig* für eine Algebra A und eine Belegung β, wenn ihr der Wahrheitswert **true** zugeordnet ist. Die Gültigkeit einer Formel φ bezüglich gegebenen A und β wird als $(A,\beta) \models \varphi$ notiert, $\not\models$ ist entsprechend das Symbol für "nicht gültig".

Definition 3.4.6 Für eine gegebene Algebra A und eine Belegung der Variablen β ist die *Gültigkeit einer Formel* $\varphi \in P_\Sigma(Y)$, kurz $(A,\beta) \models \varphi$, definiert durch

$(A,\beta) \models$ **true**		
$(A,\beta) \not\models$ **false**		
$(A,\beta) \models t_1 = t_2$	gdw.	$(A,\beta)[\![t_1]\!] = (A,\beta)[\![t_2]\!]$.
$(A,\beta) \models p(t_1,\ldots,t_n)$	gdw.	$((A,\beta)[\![t_1]\!],\ldots,(A,\beta)[\![t_n]\!]) \in A(p)$.
$(A,\beta) \models \neg\varphi$	gdw.	nicht $(A,\beta) \models \varphi$.
$(A,\beta) \models \varphi \wedge \psi$	gdw.	$(A,\beta) \models \varphi$ und $(A,\beta) \models \psi$.
$(A,\beta) \models \varphi \vee \psi$	gdw.	$(A,\beta) \models \neg(\neg\varphi \wedge \neg\psi)$.
$(A,\beta) \models \varphi \Rightarrow \psi$	gdw.	$(A,\beta) \models \neg\varphi \vee \psi$.
$(A,\beta) \models \varphi \iff \psi$	gdw.	$(A,\beta) \models (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$.
$(A,\beta) \models (\forall x)\varphi$	gdw.	für alle Belegungen β', die sich von β nur im Wert für x unterscheiden, gilt : $(A,\beta') \models \varphi$.
$(A,\beta) \models (\exists x)\varphi$	gdw.	$(A,\beta) \models \neg((\forall x)\neg\varphi)$.

□

Durch diese Definition ist die Gültigkeit einer prädikatenlogischen Formel für den allgemeinen Fall definiert. Es folgen einige Notationen für wichtige Spezialfälle.

Definition 3.4.7 Die Behandlung einer Variablen x in einer Formel hängt davon ab, ob ihre Vorkommen in der Formel durch einen Quantoren, also einem Existenz– ($\exists$) oder Allquantor ($\forall$), lokal gebunden sind. Für diese Vorkommen ist die globale Belegung der Variable x durch eine Belegung β unerheblich.

- Ein Auftreten einer Variable x heißt *gebunden* in einer Formel φ, wenn dieses Auftreten Teil einer Unterformel $(\forall x)\varphi'$ oder$(\exists x)\varphi'$ von φ ist.
- Eine Variable x heißt *frei* in einer Formel φ, wenn sie ein nicht gebundenes Auftreten in φ hat.
- Eine Formel φ heißt *geschlossen*, wenn sie keine freien Variablen enthält. □

Die vorgestellte Prädikatenlogik ermöglicht es, Aussagen über einzelne Objektzustände zu formulieren. In den nächsten Abschnitten wird diese Logik dahingehend erweitert, daß Aussagen über *zeitliche Objektentwicklungen* formuliert werden können.

3.4.3 Modellbildung für lineare temporale Logiken

Formeln der temporalen Logik sind Aussagen über zeitliche Entwicklungen. Für unsere Zwecke können zeitliche Entwicklungen durch endliche bzw. unendliche Folgen von Zuständen modelliert werden. Andere Ansätze, z.B. die Modellierung nichtdeterministischer Entwicklungen durch verzweigende Strukturen, werden u.a. in [Saa88, Eme90] diskutiert.

Die semantische Modellbildung für temporale Logiken sind Folgen von Zuständen, d.h. von Algebren einer gegebenen Signatur. Als Vereinfachung schränken wir die Algebren, die als Zustände auftreten dürfen, weiter ein und fordern feste Trägermengen als Interpretationen für die Sortensymbole für alle Zustände (eine allgemeinere Definition für Folgen mit variierenden Trägermengen wird in [Saa88, Saa89] diskutiert).

Definition 3.4.8 Eine *Zustandsfolge* $\hat{\sigma}$ ist eine indizierte Folge von Zustandsalgebren σ_i zu einer gegebenen Signatur. Endliche Folgen werden als $\hat{\sigma} = \langle \sigma_0, \sigma_1, \ldots, \sigma_n \rangle$ notiert, unendliche Folgen als $\hat{\sigma} = \langle \sigma_0, \sigma_1, \ldots \rangle$.

Alle Zustandsalgebren haben die selben Trägermengen für die Sorten $s \in S$ der gegebenen Signatur, d.h.

$$\forall (s \in S) \forall (i, j) \ \ (\sigma_i(s) = \sigma_j(s)).$$

Die Länge einer Zustandsfolge wird durch die Anzahl der Zustände definiert und wird mit $|\hat{\sigma}|$ notiert, also $|\langle \sigma_0, \sigma_1, \ldots, \sigma_n \rangle| = n + 1$. □

Die Einschränkung, daß die Trägermengen für die ganze Folge invariant sind, ermöglicht die direkte Übernahme der prädikatenlogischen Quantifizierung auch für die temporale Erweiterung. Die zustandsabhängige Information wird durch die erlaubte wechselnde Interpretation der Funktions- und Prädikatensymbole in verschiedenen Zuständen modelliert. Wird z.B. ein Attribut als nullstellige Funktion in die Signatur aufgenommen, so kann es in jedem Zustand einen anderen Wert annehmen.

Analog zu Zustandsfolgen werden auch Folgen von Variablenbelegungen definiert.

Definition 3.4.9 Eine *Belegungsfolge* $\hat{\beta}$ ist eine indizierte Folge von Belegungen β_i zu einer gegebenen Signatur. Endliche Folgen werden als $\hat{\beta} = \langle \beta_0, \beta_1, \ldots, \beta_n \rangle$ notiert, unendliche Folgen als $\hat{\beta} = \langle \beta_0, \beta_1, \ldots \rangle$. Alle Belegungen β_i sind identisch, also

$$\forall (i, j) \ \ (\beta_i = \beta_j).$$

Analog zu Zustandsfolgen, wird die Länge einer Belegungsfolge durch die Anzahl der Zustände definiert und wird mit $|\hat{\beta}|$ notiert. □

Nach der Definition der semantischen Interpretationsstrukturen für zeitliche Entwicklungen sind wir nun bereit, die temporalen Logiken zu definieren, die als logische Grundlage der Spezifikation von Objektdynamik eingesetzt werden sollen.

3.4.4 Zukunftsgerichtete temporale Logik

Die zukunftsgerichtete temporale Logik wird eingesetzt, um Aussagen über die zukünftige zeitliche Entwicklung zu form ulieren. Typische Beispiele für derartige Aussagen sind

- *"Das Gehalt eines Angestellten darf (zukünftig) nur steigen."*
- *"Ein einmal gefeuerter Angestellter darf nicht erneut eingestellt werden."*

Bei der Spezifikation von Objekten wird zukunftsgerichtete temporale Logik explizit eingesetzt, um temporale Integritätsbedingungen für zeitliche Entwicklungen von Attributwerten auszudrücken. Implizit kann sie zusätzlich als semantische Grundlage für eine logikbasierte Semantik von Sprachkonstrukten dienen (siehe Abschnitt 3.5).

Die Prädikatenlogik wird zu diesem Zweck um zusätzliche logische Konnektive erweitert, die Aussagen über zukünftige Entwicklungen ermöglichen :

next ... "ab dem nächsten Zustand gilt (falls dieser existiert)"

existsnext ... "der nächste Zustand existiert, und ab dort gilt"

always ... "ab jetzt gilt immer"

sometime ... "irgendwann (in der Zukunft) gilt"

... until ... "ab jetzt gilt immer solange bis das erstemal eine Endebedingung eintritt"

... before ... "irgendwann gilt mindestens einmal bevor das erstemal eine Endebedingung eintritt"

Diese Operatoren sind nur eine Auswahl möglicher sinnvoller Operatoren für die zukunftsgerichtete temporale Logik. Weitere Operatoren werden in [Krö87, Lip89] beschrieben. Ebendort und in [MP81, MW84, Saa88, SL89, Eme90, MP92] werden Axiomatisierungen und Gesetze dieser Logik diskutiert. Wir werden hier nur die Definition der temporalen Operatoren und einige nützliche Regeln vorstellen.

Beispiel 3.4.10 Beispiele für Formeln der zukunftsgerichteten temporale Logik sind die folgenden Formeln :

1. $\mathbf{always}((\texttt{Salary} = x) \Rightarrow \mathbf{next}(\texttt{Salary} \geq x))$
2. $\mathbf{always}((\texttt{Salary} = x) \Rightarrow \mathbf{always}(\texttt{Salary} \geq x))$
3. $\mathbf{sometime}(\texttt{Salary} > 0)$
4. $\mathbf{always}(((\texttt{Salary} = x) \wedge \mathbf{after}(\texttt{BecomeManager})) \Rightarrow \mathbf{sometime}(\texttt{Salary} \geq 2 * x)))$

Die ersten beiden Formeln drücken beide die selbe Integritätsbedingung aus, daß das Gehalt (Attribut `Salary`) nicht sinken darf. Die Bedingung 1. beschränkt jeweils alle zukünftigen Zustandsübergänge, während Bedingung 2. jeweils die ganze Restfolge einschränkt. Die dritte und vierte Formel sind Beispiele für in temporaler Logik formulierte Lebendigkeitsforderungen, die zukünftige Gehaltserhöhungen erzwingen. So fordert die Bedingung (3), daß ausgehend vom initialen Gehalt, das auf 0 gesetzt sein kann, das Gehalt mindestens einmal einen Wert > 0 erreichen muß. Das Prädikat **after** ist ein spezielles Prädikat der Spezifikationsprache, das genau in dem Zustand nach dem Eintreten des Parameterereignisses den Wert **true** annimmt. □

Die Menge der Formeln der zukunftsgerichteten temporalen Logik über einer Variablenmenge Y zu einer Signatur Σ wird mit $FTL_\Sigma(Y)$ (für *Future tense Temporal Logic*) bezeichnet.

Definition 3.4.11 Die *Formeln* in $FTL_\Sigma(Y)$ sind definiert durch :

1. Jede prädikatenlogische Formel in $P_\Sigma(Y)$ ist eine Formel in $FTL_\Sigma(Y)$.

2. Sind φ, ψ Formeln in $FTL_\Sigma(Y)$ und x eine Variable der Sorte $s \in S$, so sind auch $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$, $(\varphi \Rightarrow \psi)$, $(\varphi \iff \psi)$, $\neg\varphi$, $(\exists x\colon s)\varphi$ und $(\forall x\colon s)\varphi$ Formeln in $FTL_\Sigma(Y)$.

   Diese Formeln sind die temporalen Äquivalente zu den prädikatenlogischen Konnektiven, die aufgrund der geänderten Interpretationsstrukturen notwendig sind.

3. Sind φ, ψ Formeln in $FTL_\Sigma(Y)$, so sind auch $(\textbf{next}\,\varphi)$, $(\textbf{existsnext}\,\varphi)$, $(\textbf{always}\,\varphi)$, $(\textbf{sometime}\,\varphi)$, $(\varphi\,\textbf{until}\,\psi)$ und $(\varphi\,\textbf{before}\,\psi)$ Formeln in $FTL_\Sigma(Y)$.

4. Keine anderen Zeichenreihen sind Formeln in $FTL_\Sigma(Y)$. □

Definition 3.4.12 Die Semantik temporaler Formeln wird bezüglich eines Referenzzeitpunkts definiert. Für eine gebene Zustandsfolge $\hat{\sigma}$, einen Referenzzeitpunkt i und eine Belegungsfolge $\hat{\beta}$ ist die *Gültigkeit einer zukunftsgerichteten temporalen Formel* $\varphi \in FTL_\Sigma(Y)$, kurz $(\hat{\sigma}, i, \hat{\beta}) \models \varphi$, definiert durch

$(\hat{\sigma}, i, \hat{\beta}) \models \varphi'$	gdw.	$(\sigma_i, \beta_i) \models \varphi'$ für eine nichttemporale Formel φ'.	(1)
$(\hat{\sigma}, i, \hat{\beta}) \models \neg\varphi$	gdw.	nicht $(\hat{\sigma}, i, \hat{\beta}) \models \varphi$.	(2)
$(\hat{\sigma}, i, \hat{\beta}) \models \varphi \wedge \psi$	gdw.	$(\hat{\sigma}, i, \hat{\beta}) \models \varphi$ und $(\hat{\sigma}, i, \hat{\beta}) \models \psi$.	(3)
$(\hat{\sigma}, i, \hat{\beta}) \models (\forall x)\varphi$	gdw.	für alle Belegungsfolgen $\hat{\beta}'$ der gleichen Länge wie $\hat{\beta}$, die sich in β_0' von β_0 nur im Wert für x unterscheiden, gilt : $(\hat{\sigma}, i, \hat{\beta}') \models \varphi$.	(4)
$(\hat{\sigma}, i, \hat{\beta}) \models \textbf{existsnext}\,\varphi$	gdw.	$\lvert\hat{\sigma}\rvert > i+1$ und $(\hat{\sigma}, i+1, \hat{\beta}) \models \varphi$.	(5)
$(\hat{\sigma}, i, \hat{\beta}) \models \textbf{next}\,\varphi$	gdw.	$\lvert\hat{\sigma}\rvert > i+1$ impliziert $(\hat{\sigma}, i+1, \hat{\beta}) \models \varphi$.	(6)
$(\hat{\sigma}, i, \hat{\beta}) \models \textbf{always}\,\varphi$	gdw.	$(\hat{\sigma}, j, \hat{\beta}) \models \varphi$ für alle $i \leq j < \lvert\hat{\sigma}\rvert$.	(7)
$(\hat{\sigma}, i, \hat{\beta}) \models \textbf{sometime}\,\varphi$	gdw.	es existiert ein j, $i \leq j < \lvert\hat{\sigma}\rvert$, so daß $(\hat{\sigma}, j, \hat{\beta}) \models \varphi$.	(8)
$(\hat{\sigma}, i, \hat{\beta}) \models \varphi\,\textbf{until}\,\psi$	gdw.	$(\hat{\sigma}, j, \hat{\beta}) \models \varphi$ für alle $i \leq j < l$ wobei l definiert ist durch $l = \min\{\{k \mid (\hat{\sigma}_k, i, \hat{\beta}_k) \models \psi\} \cup \{\lvert\hat{\sigma}\rvert\}\}$.	(9)
$(\hat{\sigma}, i, \hat{\beta}) \models \varphi\,\textbf{before}\,\psi$	gdw.	es existiert ein j, $i \leq j < l$, so daß $(\hat{\sigma}, j, \hat{\beta}) \models \varphi$ wobei l definiert ist wie in (9).	(10)

Wir legen fest, daß eine Formel unabhängig vom Referenzpunkt gültig ist, falls sie zum Referenzzeitpunkt 0 gültig ist.

Die Regel (1) bettet die statischen prädikatenlogischen Formeln in die temporale Logik ein, in dem diese zum Referenzzeitpunkt, dem sogenannten *aktuellen Zustand*, ausgewertet werden. Hier wurden nur die Bedeutung der Konnektive $\neg$, $\wedge$ und $\forall(x)$ definiert, da die anderen auch als von diesen abgeleitete Konnektive definiert werden können. Die folgenden Regeln (2), (3) und (4) definieren die bereits erwähnte notwendige Erweiterung der Semantikdefinition der prädikatenlogischen Konnektive für temporale Interpretationsstrukturen. □

Wie erwähnt, soll an dieser Stelle weder eine vollständige Axiomatisierung noch vollständige Diskussion der temporalen Logik erfolgen. Eine ausführlichere Diskussion kann in [MP81, Krö87, Eme90, MP92] gefunden werden. Um ein Gefühl für die Bedeutung temporaler Formeln zu vermitteln, sollen stattdessen einige Tautologien für temporale Formeln aufgeführt werden.

Lemma 3.4.13 Für temporale Formeln $\varphi, \psi \in FTL$ gelten folgende Äquivalenzen:

$$\begin{array}{rcl}
\neg\,\mathbf{next}\,\varphi & \iff & \mathbf{existsnext}\,\neg\varphi \\
\neg\,\mathbf{always}\,\varphi & \iff & \mathbf{sometime}\,\neg\varphi \\
\neg(\varphi\,\mathbf{until}\,\psi) & \iff & (\neg\varphi)\,\mathbf{before}\,\varphi \\
\mathbf{always}\,\varphi & \iff & \varphi\,\mathbf{until\,false} \\
\mathbf{sometime}\,\varphi & \iff & \varphi\,\mathbf{before\,false}
\end{array}$$

Diese Äquivalenzen zeigen u.a., daß nur zwei temporale Operatoren, etwa **until** und **next**, als Definitionsgrundlage ausreichen, da die Semantik der anderen von ihnen abgeleitet werden kann. □

3.4.5 Vergangenheitsgerichtete temporale Logik

Die vergangenheitsgerichtete temporale Logik wird im Gegensatz zur zukunftsgerichteten eingesetzt, um Aussagen über die bisherige zeitliche Entwicklung zu formulieren. Typische Beispiele für derartige Aussagen sind

- *"Ein Buch wurde (in der Vergangenheit) bereits einmal ausgeliehen."*
- *"Der Kontostand war bisher immer höher als 100.-DM."*

Bei der Spezifikation von Objekten in den Sprachen **Oblog**$^+$ und TROLL wird vergangenheitsgerichtete temporale Logik explizit bei der Spezifikation der Objektlebensläufe eingesetzt, so

- bei der Angabe von Sicherheitsbedingungen für Ereignisse,
- bei den Bedingungen für bedingte Auswerteregeln,
- bedingten Ereignisaufrufen (conditional calling),

- und bedingten Lebendigkeitsforderungen.

Die vergangenheitsgerichtete temporale Logik ist spiegelbildlich zur zukunftsgerichteten aufgebaut [Ser80, Eme90, MP92, Cho92a]. Demnach haben wir die folgenden Entsprechungen zu den bisherigen temporalen Operatoren :

previous ... "bezüglich dem vorigen Zustand galt (falls dieser existierte)"

existsprevious ... "der vorige Zustand existierte, und dort galt"

always$_p$ **...** "bis jetzt galt immer"

sometime$_p$ **...** "irgendwann (in der Vergangenheit) galt"

always ... since last ... "bis jetzt galt immer seit dem letzten Zeitpunkt an dem das letztemal eine Bedingung eintrat"

sometime ... since last ... "irgendwann galt mindestens einmal seitdem das letztemal eine Bedingung eintrat"

Auch diese Operatoren sind nur eine Auswahl möglicher sinnvoller Operatoren für die vergangenheitsgerichtete temporale Logik. In der Spezifikationssprache wird anstatt **always**$_p$ (der Index $_p$ steht für *past*) als Schlüsselwort nur **always** benutzt, da aus dem Kontext eindeutig ist, ob vergangenheitsbezogene oder zukunftsbezogenen Logik verwendet wird (analog für den **sometime**-Operator). Die folgenden Definitionen erfolgen analog zur zukunftsgerichteten Logik.

Die Menge der Formeln der vergangenheitsgerichteten temporalen Logik über einer Variablenmenge Y zu einer Signatur Σ wird mit $PTL_\Sigma(Y)$ (für *Past tense Temporal Logic*) bezeichnet.

Definition 3.4.14 Die *Formeln* in $PTL_\Sigma(Y)$ sind definiert durch :

1. Jede prädikatenlogische Formel in $P_\Sigma(Y)$ ist eine Formel in $PTL_\Sigma(Y)$.

2. Sind φ, ψ Formeln in $PTL_\Sigma(Y)$ und x eine Variable der Sorte $s \in S$, so sind auch $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$, $(\varphi \Rightarrow \psi)$, $(\varphi \iff \psi)$, $\neg\varphi$, $(\exists x\colon s)\varphi$ und $(\forall x\colon s)\varphi$ Formeln in $PTL_\Sigma(Y)$.

   Diese Formeln sind wieder die temporalen Äquivalente zu den prädikatenlogischen Konnektiven, die aufgrund der geänderten Interpretationsstrukturen notwendig sind.

3. Sind φ und ψ Formeln in $PTL_\Sigma(Y)$, so sind auch (**previous** φ), (**existsprevious** φ), (**always**$_p$ φ), (**sometime**$_p$ φ), (**always** φ **since last** ψ) und (**sometime** φ **since last** ψ) Formeln in $PTL_\Sigma(Y)$.

4. Keine anderen Zeichenreihen sind Formeln in $PTL_\Sigma(Y)$. □

Definition 3.4.15 Die Semantik temporaler Formeln in $PTL_\Sigma(Y)$ wird wieder bezüglich eines Referenzzeitpunkts definiert. Für eine gebene Zustandsfolge $\hat{\sigma}$, einen Referenzzeitpunkt i und eine Belegungsfolge $\hat{\beta}$ ist die *Gültigkeit einer vergangenheitsgerichteten temporalen Formel* $\varphi \in PTL_\Sigma(Y)$, kurz $(\hat{\sigma}, i, \hat{\beta}) \models \varphi$, definiert durch folgende Regeln. Die Regeln (1) bis (4) aus Definition 3.4.12 werden unverändert übernommen.

$(\hat{\sigma}, i, \hat{\beta}) \models \textbf{existsprevious}\, \varphi$	gdw.	$i > 0$ und $(\hat{\sigma}, i-1, \hat{\beta}) \models \varphi$.	(5')
$(\hat{\sigma}, i, \hat{\beta}) \models \textbf{previous}\, \varphi$	gdw.	$i = 0$ oder $(\hat{\sigma}, i-1, \hat{\beta}) \models \varphi$.	(6')
$(\hat{\sigma}, i, \hat{\beta}) \models \textbf{always}_p\, \varphi$	gdw.	$(\hat{\sigma}, j, \hat{\beta}) \models \varphi$ für alle $0 \leq j < i$.	(7')
$(\hat{\sigma}, i, \hat{\beta}) \models \textbf{sometime}_p\, \varphi$	gdw.	es existiert ein j, $0 \leq j < i$, so daß $(\hat{\sigma}, j, \hat{\beta}) \models \varphi$.	(8')
$(\hat{\sigma}, i, \hat{\beta}) \models \textbf{always}\, \varphi\, \textbf{since last}\, \psi$	gdw.	$(\hat{\sigma}, j, \hat{\beta}) \models \varphi$ für alle $l < j \leq i$ wobei l definiert ist durch $l = \max\{\{k \mid 0 \leq k \leq i \wedge (\hat{\sigma}_k, i, \hat{\beta}_k) \models \psi\} \cup \{-1\}\}$.	(9')
$(\hat{\sigma}, i, \hat{\beta}) \models \textbf{sometime}\, \varphi\, \textbf{since last}\, \psi$	gdw.	es existiert ein j, $l < j \leq i$, so daß gilt $(\hat{\sigma}, j, \hat{\beta}) \models \varphi$ wobei l definiert ist wie in (9').	(10')

Die Regeln (i') sind genau spiegelbildlich zu den korrespondierenden Regeln (i) aus Definition 3.4.12 definiert. □

Beide vorgestellte temporale Logiken arbeiten auf der selben Klasse von Interpretationsstrukturen. Darum können beide Logiken zu einer allgemeinen temporalen Logik vereinigt werden [MP92].

Definition 3.4.16 Die *allgemeine temporale Logik* $TL_\Sigma(Y)$ ist definiert durch die Vereinigung der Definitionen für Syntax und Semantik der Logiken $FTL_\Sigma(Y)$ und $PTL_\Sigma(Y)$. □

Diese allgemeine temporale Logik bildet die Grundlage für eine einheitliche Semantik der Spezifikationskonzepte für Objekte, wie sie in Abschnitt 3.5 präsentiert wird.

3.4.6 Temporale Logik für Objektmodelle

In diesem Abschnitt soll nun der Bezug zwischen dem vorgestellten formalen Objektmodell und den temporalen Logiken hergestellt werden.

Ein Objekt $ob = (P, V)$ besteht gemäß Definition 3.3.11 aus einem Prozeß $P = (Y, \Lambda)$ und einer Beobachtungsfunktion $V = (A, \alpha)$ über P. Diese Beobachtungsfunktion bildet Anfangsstücke von Lebensläufe aus Λ, also Sequenzen von Ereignissen, auf Attributbeobachtungen über den Attributen in A ab:

$$\alpha\colon \mathbf{Prefix}(\Lambda) \rightarrow 2^{\mathbf{obs}(A)}$$

Diese Abbildung kann äquivalent durch eine Abbildung von kompletten Lebensläufen in *Folgen von Attributbeobachtungen* dargestellt werden.

Definition 3.4.17 Ein Objektmodell definiert für jeden Lebenslauf $\lambda \in \Lambda$, $\lambda \neq \epsilon$, eindeutig eine Folge $\hat{\sigma}(\lambda)$ als *Beobachtungsfolge* der Länge $|\lambda|$.

Der leere Lebenslauf $\epsilon \in \Lambda$ wird in der Konstruktion nicht berücksichtigt. □

Falls in den Beobachtungen die Attribute jeweils einen eindeutigen Wert haben, können Beobachtungsfolgen als Zustandsfolgen zu einer Signatur aufgefaßt werden, die direkt aus der Objektsignatur abgeleitet wird und in der Attribute durch Funktionen modelliert werden. Sind mehrere Werte für ein Objektattribut gleichzeitig möglich, müssen Attribute stattdessen durch eine Relation oder Funktion mit einem mengenwertigen Ergebnis interpretiert werden. Die Signatur für Zustandsalgebren besteht dann im wesentlichen aus

- der Signatur der importierten Datentypen,
- Funktionen für die Objektattribute,
- sowie einer Sorte für die Ereignisse mit einem **after**-Prädikat und den lokalen Ereignisnamen als erzeugende Funktionen.

Das **after**-Prädikat ist für ein Ereignis in genau den Zuständen mit dem Wert **true** belegt, die durch einen Zustandsübergang mit dem betreffenden Ereignis erreicht wurden. Aus einer Zustandsfolge kann also der zugrundeliegende Ereignisprozeß eindeutig rekonstruiert werden — eine Spezifikation in temporaler Logik (die ja eine Menge von Zustandsfolgen als Semantik hat) ist also geeignet Objektmodelle komplett und eindeutig zu spezifizieren.

Bemerkung 3.4.18 Das **after**-Prädikat wurde hier als ein explizites Prädikat eingeführt, um Ereignisterme von dem Prädikat 'ein bestimmtes Ereignis ist eingetreten' zu unterscheiden. Da Ereignisterme in der temporalen Logik (im Gegensatz zu anderen

Teilsprachen objektorientierter Spezifikationssprachen etwa zur Prozeßdefinition) nur im Zusammenhang mit dem **after**-Prädikat auftreten können, kann auf ein explizites Prädikat auch zugunsten einer impliziten Benutzung der Ereignisterme als Prädikate verzichtet werden.

Dieser Ansatz wurde etwa in der auf temporaler Logik basierenden Spezifikationslogik von C. Arapis [Ara91a, Ara91b]. In der Sprache TROLL [JSHS91] wird das **after**-Prädikat explizit notiert. □

Die Interpretation der Attribute als (totale) Funktionen beschränkt diesen Ansatz allerdings auf eindeutige Attributbelegungen. Um auch undefinierte Attributwerte behandeln zu können, müssen die Wertebereiche dieser Attribute dann einen Wert *undef* beinhalten. Die genaue Definition der Zustandssignatur für ein Objekt wird in Abschnitt 3.5.2 beschrieben.

Während für ein gegebenes Objektmodell sich eindeutig eine äquivalente Menge von Zustandsfolgen als Modelle für eine temporale Logik angeben läßt (und sich das Objektmodell wieder rekonstruieren läßt), gilt die Umkehrung nicht. Der Grund ist, daß der Ereignisprozeß aus den Werten des **after**-Prädikats rekonstruiert wird. Eine Menge von Zustandsfolgen als Modell einer temporalen Spezifikation kann nun aber zwei Zustandsfolgen enthalten, die in der Interpretation des **after**-Prädikats übereinstimmen, in den Attributwerten aber differieren ! Dieser Effekt muß bei einer Semantikfestlegung mittels temporaler Logik beachtet werden. Beliebige Mengen von Zustandsfolgen entsprechen somit einem erweiterten Objektmodell, in dem einem Lebenslauf mehrere Beobachtungen zugeordnet sind. Diese Erweiterung entspricht in der Ausdrucksfähigkeit dem in [ES91] diskutierten Modell, in dem Attributbeobachtungen als zu ändernden Ereignissen gleichberechtigte Leseereignisse modelliert werden.

3.5 Semantik von Objektspezifikationen in temporaler Logik

Die Objektbeschreibungen einer objektorientierten Spezifikationssprache wie **Oblog**$^+$ oder TROLL können im wesentlichen als syntaktischer Zucker für einen auf temporaler Logik basierenden Spezifikationskalkül aufgefaßt werden. Wie bereits erwähnt, ist die Art der syntaktischen Darstellung wesentlich an der erhofften Akzeptanz durch spätere Benutzer einer Spezifikationssprache beteiligt, da sie Verständlichkeit und Präsentierung von Entwürfen stark beeinflußt.

In diesem Abschnitt soll skizziert werden, wie aus einer Spezifikation eines Objekts eine Menge von temporalen Kalkülformeln abgeleitet werden kann, die für die zugeordneten Zustandsfolgen $\hat{\sigma}(\lambda)$ Einschränkungen erzwingt, die der Objektspezifikation entsprechen. Hier soll keine komplette formale Semantik in temporaler Logik angegeben werden, sondern nur die Basisideen vermittelt werden.

Hierzu wird im ersten Teilabschnitt die Signatur einer temporalen Logik festgelegt, die den temporalen Kalkül für eine einzelne Objektspezifikation festlegt. Die Signatur wird aus der lokalen Objektsignatur (Attribute und Ereignisse) und den Import-Klauseln bestimmt. Die restlichen Spezifikationsbestandteile werden dann in ein Menge von temporalen Formeln zu dieser Signatur umgewandelt, die als Axiome einer Objekttheorie aufgefaßt werden, die die Semantik einer Objektspezifikation festlegen.

3.5.1 Aufbau einer Objektspezifikation

Der Aufbau von Objektspezifikationen wurde in Abschnitt 3.1 anhand von Beispielen ausführlich vorgestellt. Hier soll dieser Aufbau nur kurz rekapituliert werden. Die Beschreibung eines Objekts in **Oblog**$^{+}$ ist syntaktisch wie folgt aufgebaut :

object `Objektname;`
  template
    data types *importierte Datentypen;*
    attributes *Attributnamen und -typen;*
    events *Ereignisnamen und -parameter;*
    constraints *Integritätsbedingungen;*
    valuation *Attributänderungen durch Ereignisse;*
    safety *Sicherheitsbedingungen für Ereignisse;*
    liveness *Lebendigkeitsforderungen;*
end object `Objektname;`

In den folgenden Abschnitten wird beschrieben, wie die einzelnen Spezifikationsteile in die Signatur und die Axiome einer temporalen Logik überführt werden können und auf diese Weise eine logikbasierte Semantik von Objektspezifikationen festlegen.

3.5.2 Lokale Signatur

Die Spezifikationskonstrukte einer Objektbeschreibung sollen in Formeln der vorgestellten temporalen Logik überführt werden. Um dies tun zu können, muß als erster Schritt die *lokale Signatur* Σ_{ob} für ein Objekt *ob* festgelegt werden.

Der erste Bestandteil der lokalen Signatur ist die Signatur Σ_{DT} der importierten Datentypen. Zusätzlich wird eine Sorte E_{ob} der Ereignisse eingeführt. Attribute werden als parameterlose Funktionen passender Sorte modelliert, und Ereignisse werden als Funktionen mit Zielsorte E_{ob} dargestellt. Als zusätzliches Prädikat wird das **after**-Prädikat über der Ereignissorte definiert.

Definition 3.5.1 Die *lokale Objektsignatur* Σ_{ob} ist definiert durch

$$
\begin{array}{l}
\Sigma_{ob} = \Sigma_{DT} + \\
\quad (\\
\qquad S' = \{E_{ob}\}, \qquad\qquad\qquad \text{(* Sorte der lokalen Ereignisse *)} \\
\qquad \Omega' = \{a :\rightarrow \textbf{sort}(a) \ \mid \text{für alle Attribute } \texttt{a: sort(a)} \ \} \ \cup \\
\qquad\quad \{e : s_1 \times \ldots \times s_n \rightarrow E_{ob} \ \mid \text{für Ereignisse } \texttt{e(s1,..,sn)}\}, \\
\qquad \Pi' = \{\textbf{after} : E_{ob}\} \\
\quad)
\end{array}
$$

Diese Definition kann leicht um die Berücksichtigung von sogenannten *Attributgeneratoren*, also parametrisierten Attributen der Form $a : (s_1 \times \ldots \times s_n) \rightarrow \textbf{sort}(a)$, erweitert werden. □

Beispiel 3.5.2 Ein Teil der Signatur $\Sigma_{\texttt{copy}}$ des `copy`-Objekts (Beispiel 3.2.2 auf Seite 48) kann wie folgt angegeben werden:

$$
\begin{array}{rcl}
S_{\texttt{copy}} & = & \{\texttt{nat},\ \texttt{bool}, |\texttt{BOOK}|, ..., E_{\texttt{copy}}\} \\
\Omega_{\texttt{copy}} & = & \{\texttt{plus:}\,\texttt{nat} \times \texttt{nat} \rightarrow \texttt{nat}, \texttt{times:}\,\texttt{nat} \times \texttt{nat} \rightarrow \texttt{nat}, ..., \\
 & & \texttt{Doc\_No:} \rightarrow \texttt{nat}, \texttt{Of:} \rightarrow |\texttt{BOOK}|, ..., \\
 & & \texttt{Get\_Copy:} \rightarrow E_{\texttt{copy}}, \texttt{Check\_Out:}\, |\texttt{USER}| \times \texttt{date} \times \texttt{nat} \rightarrow E_{\texttt{copy}}, ..., \} \\
\Pi_{\texttt{copy}} & = & \{\texttt{equal:}\,\texttt{nat} \times \texttt{nat}, ... \\
 & & \textbf{after} : E_{\texttt{copy}}\}
\end{array}
$$

Die Teilsignatur der importierten Datentypen wurde natürlich nur teilweise aufgeführt.

Korrekte Terme dieser Signatur sind z.B. `Doc_No`$+2$ oder `Of`, und atomare Formeln sind etwa '**after**(`Check_Out`(`person`(`´Müller´`, 05.10.60), 13.11.90, 4))' oder `Doc_No` $>$ 100 wobei in der ersten Formel `person` eine Funktion zur Konstruktion von `USER`-Identifikatoren ist und 05.10.60 und 13.11.90 Konstanten des importierten Datentyps `date` sind. □

Nach diesen Festlegungen steht die (lokale) Signatur für eine Objektbeschreibung fest. Die Variablen Y_{ob} und deren Typ für eine gegebene Objektspezifikation *ob* werden direkt aus den Variablendeklarationen nach dem Schlüsselwort **variables** übernommen.

In den folgenden Abschnitten werden wir den Index $_{ob}$ bei der Signatur Σ_{ob} und den Variablen Y_{ob} weglassen, wenn das betrachtete Objektmodell *ob* eindeutig aus dem Kontext hervorgeht.

3.5.3 Integritätsbedingungen

Integritätsbedingungen werden eingesetzt um die möglichen Attributwerte eines Zustands bzw. die zeitliche Entwicklungen dieser Werte einzuschränken. Der benutzte

Spezifikationsformalismus ist zukunftsgerichtete temporale Logik. Integritätsbedingungen können somit direkt in Formeln der temporalen Logik $TL_\Sigma(Y)$ transformiert werden.

Bei den Integritätsbedingungen einer Objektspezifikation unterscheiden wir in der Regel vier Klassen von Bedingungen :

1. *Statische Invarianten* φ_s sind prädikatenlogische Formeln aus $P_\Sigma(Y)$, die Einschränkungen an aktuelle Ausprägungen der Attributbelegungen (den beobachtbaren Objektzuständen) spezifizieren.

2. *Temporale Invarianten* φ_t sind temporale Formeln aus $FTL_\Sigma(Y)$, die bezüglich jedem erreichbaren Zustand erfüllt werden müssen. Strenggenommen sind statische Invarianten nur ein Spezialfall der temporalen Invarianten, werden aber in der Regel in der Literatur gesondert aufgeführt, da sie bei Integritätsüberwachung und Konsistenzuntersuchung mit anderen Techniken behandelt werden können.

   Typische Beispiele für temporale Invarianten sind Invarianten der Form

   $$(\texttt{Gehalt} = x) \Rightarrow \textbf{always}(\texttt{Gehalt} \geq x)$$

3. *Initiale Integritätsbedingungen* φ_i werden spezifiziert durch zukunftsgerichtete temporale Formeln aus $FTL_\Sigma(Y)$, die bezüglich dem Geburtszeitpunkt eines Objekts gelten sollen. Im Gegensatz zu Invarianten, beschreiben initiale Integritätsbedingungen oft Lebendigkeitsforderungen, die für korrekte Objektlebensläufe gelten müssen, so z.B.

   $$\textbf{initial} \quad \textbf{sometime}(\texttt{Gehalt} \geq 1.000)$$

   Andere typische initiale Bedingungen betreffen die Initialisierung von Objektattributwerten bzw. Einschränkungen der initialen Attributwerte, so die Einschränkung

   $$\textbf{initial} \quad \texttt{Gehalt} \geq 0$$

   oder die Initialisierung

   $$\textbf{initial} \quad \texttt{BisherigesGesamtgehalt} = 0$$

   Initiale Integritätsbedingungen werden in **Oblog$^+$** mit dem Schlüsselwort **initial** gekennzeichnet. In der Sprache TROLL lautet das entsprechende Schlüsselwort **initially**.

4. Die *Ableitungsregeln* φ_d (derivation rules) können als eingeschränkte Form der statischen Invarianten aufgefaßt werden. In der Objektspezifikation werden sie wie folgt notiert :

   derivation rules
   `AttributTerm =` *Term des passenden Datentyps*;

Definition 3.5.3 Die Umwandlung der Integritätsbedingungen in temporale Formeln als Axiome der zu bildenden Objekttheorie geschieht in zwei Schritten. Als ersten Schritt werden die Formeln wie folgt umgeformt (wir nehmen an, daß die Symbole der Spezifikationssprache bereits in die entsprechenden Symbole der temporalen Logik übersetzt worden, so etwa das syntaktische Symbol `and` in den Operator $\wedge$) :

1. Statische Invarianten φ_s werden zu $\mathbf{always}(\varphi_s)$ erweitert.
2. Temporale Invarianten φ_t werden zu $\mathbf{always}(\varphi_t)$ erweitert.
3. Initiale Bedingungen φ_i werden unverändert übernommen.
4. Ableitungsregeln φ_d werden als statische Invarianten interpretiert und zu der Formel $\mathbf{always}(\varphi_d)$ erweitert.

Als zweiter Schritt werden die einzelnen Formeln mit allen in ihnen frei auftretenden Variablen allquantifiziert. □

Beispiel 3.5.4 Die temporale Invariante

$$(\texttt{Gehalt} = x) \Rightarrow \mathbf{always}(\texttt{Gehalt} \geq x)$$

wird zu

$$\forall(x\colon \mathbf{integer})\ \ \mathbf{always}((\texttt{Gehalt} = x) \Rightarrow \mathbf{always}(\texttt{Gehalt} \geq x))$$

erweitert. □

3.5.4 Auswertungsregeln für Ereignisse

Auswertungsregeln verknüpfen die Prozeßereignisse mit den Belegungen der Attribute, indem sie neue Attributwerte als Nachbedingungen der Ereignisse festlegen. Das allgemeine Muster für eine Auswertungsregel (vergl. Abschnitt 3.2) ist :

`[EreignisTerm] AttributTerm = DatenTerm` *(des passenden Datentyps)*;

Die einzelnen Basisterme für das Ereignis, das Attribut und dem neuen Datenwert können direkt in Terme der Spezifikationslogik umgesetzt werden.

Definition 3.5.5 Eine Auswertungregel des obigen Musters wird überführt in die folgende temporale Formel :

$$\forall(x\colon \mathbf{type}(\texttt{AttributTerm}))$$
$$\quad \mathbf{always}((\texttt{DatenTerm} = x) \Rightarrow \mathbf{next}(\mathbf{after}(\texttt{Ereignisterm}) \Rightarrow (\texttt{AttributTerm} = x))$$

In einem zweiten Schritt werden wiederum alle freien Variablen der Formel jeweils mit einem Allquantor quantifiziert. □

Da der Wert des Attributs im *alten* Zustand berechnet werden muß, und die Belegung des Attributs im neuen Zustand zugesichert werden soll, muß eine transitionale Bedingung aufgestellt werden. Der Formelteil ($\texttt{DatenTerm} = x$) bindet den Wert des im alten Zustand ausgewerteten Datenterms an die Variable x, und im durch **next** gebundenen Formelteil wird dieser Wert dem Attribut zugeordnet falls der Zustandsübergang durch das angegebene Ereignis hervorgerufen wurde. Die Hilfsvariable x wird nur benutzt, um auf den im alten Zustand bestimmten Wert im neuen Zustand zugreifen zu können.

Beispiel 3.5.6 Als Beispiel betrachten wir einige der Auswertungsregeln aus dem `copy`-Objekt (Beispiel 3.2.2, Seite 48) aus Abschnitt 3.2.

```
valuation
   variables U: |USER|, d: date, n: nat;
   [GetCopy] OnLoan = false;
   [CheckOut(U,d,n)] Due =  add_days_to_date(d,n);
   [CheckOut(U,d,n)] Borrowers = append(U,Borrowers);
```

Die erste Regel kann direkt umgeformt werden zu der Formel

$$\forall(x\colon \texttt{bool})$$
$$\quad \mathbf{always}((\mathbf{false} = x) \Rightarrow \mathbf{next}(\mathbf{after}(\texttt{GetCopy}) \Rightarrow (\texttt{OnLoan} = x))$$

die zu

$$\mathbf{always}(\mathbf{after}(\texttt{GetCopy}) \Rightarrow (\texttt{OnLoan} = \mathbf{false}))$$

vereinfacht werden kann, da der neue Attributwert von `OnLoan` zustandsunabhängig bestimmt ist.

Die zweite Regel wird im ersten Schritt zu

$$\begin{array}{l} \forall(x\colon \mathtt{date}) \\ \quad \mathbf{always}((\mathtt{add\_days\_to\_date}(d,n) = x) \\ \qquad \Rightarrow \mathbf{next}(\mathbf{after}(\mathtt{CheckOut}(U,d,n)) \Rightarrow (\mathtt{Due} = x)) \end{array}$$

umgeformt und die resultierende Formel dann im zweiten Schritt zusätzlich mit den folgenden Quantifizierungen gebunden :

$$\forall(U\colon |\mathtt{USER}|)\forall(d\colon \mathtt{date})\forall(n\colon \mathtt{nat})$$

Auch in dieser Regel ist die Verwendung der Hilfsvariable nicht zwingend erforderlich, da bei der Berechnung des neuen Attributwerts keine zustandsabhängige Information benutzt wird. Dies ist allerdings der Fall bei der dritten Regel, die zu folgender Formel führt :

$$\begin{array}{l} \forall(U\colon |\mathtt{USER}|)\forall(d\colon \mathtt{date})\forall(n\colon \mathtt{nat}) \\ \quad \forall(x\colon \mathbf{list}(|\mathtt{USER}|)) \\ \qquad \mathbf{always}((\mathbf{append}(U,\mathtt{Borrowers}) = x) \\ \qquad\quad \Rightarrow \mathbf{next}(\mathbf{after}(\mathtt{CheckOut}(U,d,n)) \Rightarrow (\mathtt{Borrowers} = x)) \end{array}$$

Diese dritte Regel kann nicht vereinfacht werden, da der neue Wert des Attributs im alten Zustand berechnet werden muß. □

Durch die bisherigen Definitionen haben wir die explizit angegebenen Änderungen durch Auswertungsregeln erfaßt. Als letzten Schritt müssen wir die *Rahmenregel* berücksichtigen, die festlegt, daß alle Attributwerte durch ein Ereignis unverändert bleiben falls keine explizite Regel die Änderung erzwingt.

Definition 3.5.7 Die *Rahmenregel* besagt, daß Attribute nur durch explizite Auswertungsregeln geändert werden dürfen. Mit **change**(`Attributterm`) bezeichnen wir die Menge derjenigen Ereignisterme e, für die Auswertungsregeln existieren, die den `Attributterm` betreffen. Die Formel **ChangeAllowed**(`Attributterm`) ist definiert als die Disjunktion

$$\bigvee_{e \in \mathbf{change}(\mathtt{Attributterm})} \mathbf{after}(e)$$

Die Rahmenregel wird nun spezifiziert durch

$$\begin{array}{l} \forall(x\colon \mathbf{type}(\mathtt{Attributterm})) \\ \quad \mathbf{always}((\mathtt{Attributterm} = x) \Rightarrow \\ \qquad \mathbf{next}(\neg(\mathbf{ChangeAllowed}(\mathtt{Attributterm})) \Rightarrow \mathtt{Attributterm} = x)). \end{array}$$

Bei Bedarf müssen Variablen für eventuelle Ereignis- bzw. Attributparameter noch allquantifiziert werden. □

Beispiel 3.5.8 Für das `copy`-Beispiel 3.2.2 erhalten wir u.a. die folgende Rahmenregel für das **Borrowers**-Attribut :

$$
\begin{aligned}
&\forall(x\colon \mathbf{list}(|\mathtt{PERSON}|))\\
&\quad \mathbf{always}((\mathtt{Borrowers} = x) \Rightarrow \mathbf{next}(\neg(\mathbf{after}(\mathtt{Return}) \vee \mathbf{after}(\mathtt{ThrowAway}))\\
&\qquad\qquad \Rightarrow \mathtt{Borrowers} = x))
\end{aligned}
$$

□

Die korrekte Behandlung von *konstanten Attributen* (Schlüsselwort **constant**) kann entweder auf der Spezifikationsebene berücksichtigt werden, indem keine Auswertungregeln für diese Attribute auftreten dürfen, oder analog zur Rahmenregel durch explizite temporale Formeln garantiert werden.

In der Sprache TROLL können auch *bedingte Auswertungsregeln* angegeben werden, die analog etwa zu bedingten Lebendigkeitsforderungen notiert werden. Deren Umsetzung in eine logische Implikation erfolgt in der naheliegenden Weise.

3.5.5 Prozeßbeschreibung

Bei der Beschreibung der Objektprozesse, wie sie im Abschnitt 3.2 eingeführt wurde, werden verschiedene Beschreibungsformalismen eingesetzt. Wir beschränken uns hier auf die Behandlung von Aspekten der **birth**- und **death**-Ereignisse, der Sicherheitsbedingungen und der Lebendigkeitsforderungen.

Geburts- und Todesereignisse

In der Signaturbeschreibung der Objektereignisse werden neben den konkreten Ereignisnamen und -parametertypen auch die Schlüsselwörter **birth** und **death** angegeben, die anzeigen ob es sich um ein Geburts- bzw. Sterbeereignis handelt. Diese Angaben müssen in temporale Formeln umgesetzt werden, die die entsprechende intendierte Bedeutung garantieren.

Definition 3.5.9 Existiert genau ein Geburtsereignis `Ereignis` notiert als

birth `Ereignis(.., Sort_i, ..);`

mit entsprechenden Parametersorten `Sort_i`, so kann direkt die folgende Formel in temporaler Logik angegeben werden, die festlegt, daß im initialen Zustand gelten muß, daß dieser durch das Geburtsereignis erreicht wurde :

$$\ldots \exists(x_i\colon \mathtt{Sort\_i})\ldots\ \mathbf{after}(\mathtt{Ereignis}(..,\mathtt{Sort\_i},..))$$

Als zweite Bedingung haben wir, daß anschließend kein weiterer Übergang mehr durch ein Geburtsereignis hervorgerufen wurde :

$$\ldots \forall(x_i\colon \texttt{Sort\_i}) \ldots \ \textbf{next}(\textbf{always}(\neg\, \textbf{after}(\texttt{Ereignis}(.., x_i, ..))))$$

Im Falle mehrerer Geburtsereignisse muß die erste Formel durch eine Disjunktion von Einzelformeln für die einzelnen Ereignisse ersetzt werden, die zweite hingegen durch eine entsprechende Konjunktion. □

Die Behandlung der Sterbeereignisse gestaltet sich etwas komplizierter, da in der verwendeten temporalen Logik kein Operator explizit vorhanden ist, der auf den *letzten* Zustand eines Lebenslaufs hinweist. Implizit können wir hingegen die Eigenschaft der Logik verwenden, daß die Formel **next**(**false**) genau im letzten Zustand einer Folge wahr wird.

Definition 3.5.10 Existiert genau ein Sterbeereignis `Ereignis` notiert als

death `Ereignis(.., Sort_i, ..);`

mit den entsprechenden Parametersorten `Sort_i`, so wird diese Angabe in die folgende temporale Formel in temporale Logik überführt :

$$\ldots \forall(x_i\colon \texttt{Sort\_i}) \ldots \ \textbf{always}(\textbf{after}(\texttt{Ereignis}(.., \texttt{Sort\_i}, ..)) \Rightarrow \textbf{next}(\textbf{false}))$$

Nach einem Übergang mit einem Sterbeereignis muß die Zustandsfolge also abbrechen. □

Beispiel 3.5.11 Für das `copy`-Beispiel (Beispiel 3.2.2) erhalten wir die Formeln

$$\textbf{after}(\texttt{GetCopy})$$

und

$$\textbf{next}(\textbf{always}(\neg\, \textbf{after}(\texttt{GetCopy})))$$

aufgrund der Festlegung des `GetCopy`-Ereignisses als Geburtsereignis. Für das Sterbeereignis `ThrowAway` erhalten wir die Formel

$$\textbf{always}(\textbf{after}(\texttt{ThrowAway})) \Rightarrow \textbf{next}(\textbf{false}))$$

als temporale Formel, die einen Abbruch der Zustandsfolge nach dem ersten Eintreten von `ThrowAway` erzwingt. □

Sicherheitsbedingungen

Sicherheitsbedingungen stellen Einschränkungen für diejenigen Zustände auf, in denen ein Ereignis eintreten kann. Ihre Gültigkeit wird relativ zu einem Zustand *vor* dem potentiellen Ereigniseintreten bestimmt.

Definition 3.5.12 Die allgemeine Form einer Sicherheitsbedingung in **Oblog$^+$** nach dem Schlüsselwort **safety** ist

$$\{\ \varphi\ \}\ \texttt{EreignisTerm ;}$$

wobei φ eine Formel der vergangenheitsgerichteten temporalen Logik $PTL_\Sigma(Y)$ ist. Diese Angabe wird im ersten Schritt umgewandelt in die Formel

$$\textbf{always}(\neg\varphi \Rightarrow \textbf{next}(\neg\,\textbf{after}(\texttt{EreignisTerm})))$$

und im zweiten Schritt werden alle freien Variablen allquantifiziert. □

Man beachte, daß die aus dieser Umwandlung resultierende Formel sowohl zukunftsgerichtete als auch vergangenheitsbezogene temporale Operatoren enthalten kann.

Beispiel 3.5.13 Als Beispiel betrachten wir die Sicherheitsbedingungen des `copy`-Beispiels 3.2.2 (Seite 48) aus Kapitel 3.2.

```
safety
   variables U: |USER|, d: date, n: nat;
   {OnLoan = false} CheckOut(U,d,n);
   {exists(U: |USER|, d: date, n: nat)
      sometime(after(CheckOut(U,d,n))) since last (after(Return))}
         Return;
```

Die erste Formel wird umgewandelt in

$$\begin{array}{l}\forall(U\colon |\texttt{USER}|)\forall(d\colon \texttt{date})\forall(n\colon \texttt{nat})\\ \quad \textbf{always}(\neg(\texttt{OnLoan} = \textbf{false}) \Rightarrow \textbf{next}(\neg\,\textbf{after}(\texttt{CheckOut}(U,d,n))))\end{array}$$

Als Resultat der Umwandlung der zweiten Sicherheitsbedingung erhalten wir

$$\begin{array}{l}\textbf{always}(\\ \quad \neg(\exists(U' : |\texttt{USER}|)\exists(d' : \texttt{date})\exists(n' : \texttt{nat})\\ \qquad \textbf{sometime}_p(\textbf{after}(\texttt{CheckOut}(U',d',n'))))\\ \qquad \textbf{since last}\ (\textbf{after}(\texttt{Return}))) \quad \Rightarrow \textbf{next}(\neg\,\textbf{after}(\texttt{Return})))\end{array}$$

Wir haben hier die lokal mit $\exists$ gebundenen Variablen umbenannt, um die lokalen Variablen deutlicher hervorzuheben. □

Lebendigkeitsforderungen

Für Lebendigkeitsforderungen gibt es eine Reihe von Notationen insbesondere für Kombinationen möglicher Ereignisse, die in der bisherigen Beschreibung nicht vollständig erwähnt wurden. Wir definieren im folgenden die einzelnen syntaktischen Konstrukte und erläutern ihre Bedeutung direkt anhand der Umwandlung in temporale Formeln.

Definition 3.5.14 In der folgenden Auflistung werden die verschiedenen Kombinationen von Lebendigkeitsforderungen für Ereignisse aufgelistet und ihre Entsprechung in temporaler Logik angegeben. Die Konstruktion von Lebendigkeitsforderungen erfolgt rekursiv aus einfacheren Bedingungen. Analog zu diesem rekursiven Aufbau, werden hier für Konstruktionsmuster für Lebendigkeitsforderungen (syntaktische Kategorie `LivenessTerm`) entsprechende Konstruktionsvorschriften für Formeln angegeben, die rekursiv weiter angewendet werden müssen.

- Eine *atomare Lebendigkeitsforderung* (nach dem Schlüsselwort **liveness**) besteht aus der Angabe eines Ereignisterms und wird notiert als

  ```
  EreignisTerm;
  ```

  Die entsprechende temporale Formel erzwingt mit Hilfe des **sometime**-Operators das Eintreten des Ereignisses :

  $$\textbf{sometime}(\textbf{after}(\texttt{EreignisTerm}))$$

- Da freie Variablen universell quantifiziert werden und somit Lebendigkeitsforderungen *für alle möglichen Belegungen* der Parameter mit Werten bedeuten, muß es auch möglich sein, durch ein Sprachkonstrukt die Parametervariablen eines Ereignisterms *existentiell zu binden*. Dies wird notiert als

  ```
  exists ( n : type ) ... Ereignis(..., n, ...) ;
  ```

  Die entsprechende Formel wird gebildet durch

  $$\exists(n : \texttt{type})...\,\textbf{sometime}(\textbf{after}(\texttt{Ereignis}(..., n, ...)))$$

  Die Punkte in der Notation deuten an, daß für einen Ereignisterm mehrere Parameter existentiell gebunden sein können.

- Eine *Konjunktion* von Lebendigkeitsbedingungen kann durch

```
EreignisTerm1 & EreignisTerm2 ;
```

spezifiziert werden. Ihr entspricht die Formelkonstruktion :

$$\textbf{sometime}(\textbf{after}(\texttt{EreignisTerm1})) \wedge \textbf{sometime}(\textbf{after}(\texttt{EreignisTerm2}))$$

Eine Konjunktion von Lebendigkeitsbedingungen ist äquivalent zu der getrennten Spezifikation der beiden Bedingungen in separaten Zeilen. In der Sprache TROLL wird anstatt `&` das Schlüsselwort **and** verwendet.

Die Konjunktion mit `&` (und die folgenden Konstrukte) können nicht nur auf elementare Ereignisterme angewendet werden, sondern auch beliebige Lebendigkeitsforderungen `LivenessTerm_x` verbinden. Die Konstruktion der Formeln erfolgt dann analog obigem Muster. Die einzelnen Lebendigkeitsforderungen `LivenessTerm_x` müssen dann gegebenenfalls weiter aufgespalten werden.

- Eine *Disjunktion* von Lebendigkeitsbedingungen kann durch

```
EreignisTerm1 | EreignisTerm2 ;
```

spezifiziert werden. Ihr entspricht die Formelkonstruktion :

$$\textbf{sometime}(\textbf{after}(\texttt{EreignisTerm1})) \vee \textbf{sometime}(\textbf{after}(\texttt{EreignisTerm2}))$$

In der Sprache TROLL wird anstelle des Symbols `|` das Schlüsselwort **or** verwendet.

- Als letzten Fall haben wir die *bedingten Lebendigkeitsforderungen*. Sie werden notiert als

$$\{\ \varphi\ \}\ \texttt{==> EreignisTerm ;}$$

wobei φ eine Formel der vergangenheitsgerichteten temporalen Logik $PTL_{\Sigma}(Y)$ ist. Die bedingten Bedingungen werden umgesetzt in

$$\textbf{always}(\varphi \ \Rightarrow \textbf{sometime}(\texttt{EreignisTerm}))$$

Hier ist zu bemerken, daß bedingte Lebendigkeitsforderungen *zustandsabhängig* sind, d.h. daß sie in Abhängigkeit vom Objektzustand neu zu den noch zukünftig zu erfüllenden Bedingungen hinzugenommen werden können. Einfache Bedingungen hingegen entsprechen initialen Integritätsbedingungen.

Als abschließender Schritt müssen auch für Lebendigkeitsforderungen alle freien Variablen allquantifiziert werden. □

Beispiel 3.5.15 Als einfaches Beispiel betrachten wir wiederum die Lebendigkeitsforderung des copy-Beispiels 3.2.2 aus Kapitel 3.2.

```
liveness
  { exists(U: |USER|, d: date, n: nat) after(CheckOut(U,d,n)) }
    ==> Return;
```

Die aus der Umwandlung resultierende Formel lautet

$$\mathbf{always}(\\ (\exists(U : |\mathtt{USER}|)\exists(d : \mathtt{date})\exists(n : \mathtt{nat})\,\mathbf{after}(\mathtt{CheckOut}(U, d, n)))\\ \Rightarrow \mathbf{sometime}(\mathbf{after}(\mathtt{return}))$$

□

Als abschließende Bemerkung dieses Abschnitts sei erwähnt, daß prinzipiell auch andere Prozeßbeschreibungformalismen (explizite z.B. CSP-ähnliche Notationen für Prozesse, Petri-Netze, Übergangsdiagramme etc.) sich in eine Menge von temporalen Formeln umwandeln lassen, die äquivalente Einschränkungen für Objektlebensläufe ausdrücken. Eventuell muß hierbei allerdings ein expliziter Zustand in Form eines Zustandsattributs zusätzlich modelliert werde.

3.6 Objektklassen und Klassentypen

Einzelne Objekte können zu Objektklassen gruppiert werden, in denen die Struktur der enthaltenen Objekte durch einen zugeordneten Klassentyp festgelegt wird. Wie bereits erwähnt, werden Klassentypen in der Literatur auch als Objekttypen oder Objektklassentypen bezeichnet. Zusätzlich zur Beschreibung der Objektstrukturen legt ein Klassentyp einen Identifikationsmechanismus für die Klassenelemente fest, in dem ein Namensraum für Objektidentifikatoren vorgegeben wird.

3.6.1 Semantische Modellbildung *

Ein Klassentyp muß die Strukturen der Objekte einer Objektklasse bestimmen sowie einen Namensraum festlegen. Der allgemeine Fall ermöglicht Objekte *unterschiedlicher Struktur* als Elemente einer Klasse, die dann als *heterogene* Klasse bezeichnet wird. Der entsprechende Klassentyp wird dann ebenfalls als heterogen bezeichnet. Nichtheterogene Klassen und Klassentypen bezeichnen wir als *homogen*. Homogene Klassen sind der Normalfall bei Spezifikationen, während heterogene Klassen in der

Regel hauptsächlich als Ergebnis einer Generalisierung auftreten. Sprachmittel zur Spezifikation heterogener Klassen werden in Abschnitt 4.6 vorgestellt.

In den folgenden Definitionen werden beide Fälle berücksichtigt.

Definition 3.6.1 Das semantische Modell eines *Klassentyps* $ct = (ID, OB, \omega)$ besteht aus einem Wertebereich ID als Namensraum, einer Menge von Objektmodellen OB und einer Zuordnungsfunktion $\omega: ID \to OB$, die jedem Wert des Namenraums ein Objektmodell aus OB zuordnet. Der Wertebereich ID ist definiert als die Trägermenge einer Sorte eines abstrakten Datentyps.

Ein Klassentyp ct ist homogen, falls $ct.OB = \{ob\}$ einelementig ist. In diesem Fall ist $ct.\omega$ eine konstante Abbildung. □

Die Zuordnung von Objektidentifikatoren zu Objektmodellen ist hierbei statisch; ein Klassenelement einer heterogenen Klasse kann also nicht zur Laufzeit seine Struktur wechseln, und die Struktur eines Elements liegt bei seiner Erzeugung für einen konkreten Identifikator bereits fest.

Die intuitive Sicht auf *Klassen* ist die eines 'Behälters' für aktuelle Objekte, dessen Inhalt sich zustandsabhängig ändert. Klassentypen beschreiben die möglichen Ausprägungen intensional, d.h. sie geben alle möglichen Populationen einer Klasse an, die diesen Typ hat. Die erwähnte Sicht auf Klassen selber hingegen ist extensional — Klassen entsprechen Variablen zu einem Klassentyp, deren Extension durch Einfügen und Löschen von Objekten manipuliert wird. Wir folgen zuerst dieser zustandsorientierten Sicht und beginnen damit, den Zustand einer Klasse zu definieren.

Definition 3.6.2 Ein *Zustand einer Objektklasse oc*, notiert als

$$\sigma_{oc} = (ct, \zeta[ct.OB]),$$

wird beschrieben durch den zugeordneten Klassentyp ct und eine Menge von parametrisierten Zustandsfunktionen

$$\zeta[ob]: ct.ID \to \mathbf{Prefix}(ob.P.\Lambda),$$

wobei gilt daß $\zeta[ob] \neq \epsilon$ impliziert $ob = ct.\omega(i)$, d.h. ein Zustand kann einem Objekt nur über diejenige Zustandsfunktion zugeordnet werden die das passende Objektmodell als Parameter hat.

Einem gegebenen Identifikator $i \in ct.ID$ ist somit durch die Funktion $\zeta[ct.\omega(i)]$ ein aktueller Zustand $\zeta[ct.\omega(i)](i)$ zugeordnet, der ein Anfangsstück aus einem korrekten Lebenslauf des zugeordneten Objektmodells darstellt. Ist der Objektzustand $\zeta[ct.\omega(i)](i) = \epsilon$, d.h. die leere Folge, so ist das Objekt mit dem Identifikator i nicht in der aktuellen Ausprägung der Klasse enthalten.

Für homogene Klassen (d.h. $OB = \{ob\}$) ist der Wert $ct.\omega(i)$ konstant identisch zu ob für alle i. In diesem Fall können wir die vereinfachte Schreibweise $\zeta(i)$ als aktuellen Zustand des mit i identifizierten Objekts verwenden. □

Diese etwas komplexe Definition von Objektklassen resultiert aus der zustandsorientierten, extensionalen Sicht auf Objektklassen im Gegensatz zur intensionalen Semantik für Klassentypen und auch für isolierte Objektspezifikationen.

Die rein zustandsorientierte Vorgehensweise kann vermieden werden in dem wir anstatt einzelner Zustände für Objektklassen analog zu Einzelobjekten auch hier *Entwicklungen* von Objektklassen betrachten.

Definition 3.6.3 Ein *Lebenslauf einer Objektklasse* $\hat{\sigma}_{oc}$ ist definiert als eine Folge von Objektklassenzuständen. Die intensionale *Semantik* einer Klasse kann dann durch eine Menge von erlaubten Lebensläufen festgelegt werden.

Als zusätzliche Bedingung fordern wir, daß die Zustandsübergänge der Klasse für die enthaltenen Einzelobjekte elementar und korrekt sein müssen, d.h. aus der Projektion eines Klassenlebenslaufs auf ein Einzelobjekt müssen wir einen korrekten Objektlebenslauf des passenden Objektmodells rekonstruieren können. □

Wenn wir uns diese Definition genauer betrachten, bemerken wir Parallelen zur Modellbildung für Einzelobjekte — eine Klasse ist beschrieben durch Lebensläufe über Klassenzustände, und die Zustandsübergänge entsprechen dem parallelen Ausführen von Objektereignissen der enthaltenen Klassenobjekte. Für eine Klasse können wir zusätzlich eine aktuelle Beobachtung festlegen, die die aktuelle Klassenpopulation und die Zustände der Einzelobjekte anzeigt. Die Semantik einer Klasse entspricht somit der eines einzigen komplexen Objekt !

Definition 3.6.4 Die *Semantik einer Objektklasse* ist ein komplexes Objektmodell, das durch die parallele Komposition der in ihr enthaltenen Objekte gebildet wird. Ereignisse einer Objektklasse sind somit Mengen von gleichzeitig eintretenden Ereignissen der einzelnen Klassenobjekte. Die Basissignatur einer Klasse wird gebildet durch die disjunkte Vereinigung der Signaturen der Einzelobjekte (eindeutige Namen werden erreicht durch Hinzufügung eines objektspezifischen Präfixes gebildet aus dem Identifikator). □

Es liegt nahe, die Modellbildung für Klassentypen und Objektklassen derart zu erweitern, daß wir auch *Klassenattribute und Klassenereignisse* analog zu Einzelobjekten zulassen.

Definition 3.6.5 Die *erweiterte Modellbildung* eines Klassentyps notiert als $ct = (ID, OB, \omega, \Sigma_{ct}, \mathbf{spec}_{ct})$ enthält als weitere Komponenten eine Klassensignatur Σ_{ct} analog zu einer Objektsignatur und eine Spezifikation $\mathbf{spec}_{ct}$ der Klassenprozesse und -beobachtungen.

Die Klassensignatur definiert Klassenereignisse und Klassenattribute, und die Spezifikation $\mathbf{spec}_{ct}$ legt deren Semantik mittels der für Einzelobjekte eingeführten Formalismen fest. □

Klassenattribute und -ereignisse können zum Teil generisch definiert sein, so etwa das Ereignis des Initialisierens einer Klasse oder ein Attribut `AktuelleObjektanzahl`. Zusätzlich können etwa auch klassenspezifische Attribute wie etwa `Durchschnittsgehalt` für eine Klasse `ANGESTELLTE` definiert werden.

Während Klassenattribute und -ereignisse in der formalen Modellbildung bereits integriert sind [ES91], sind sie bislang in die Sprachvorschläge für **Oblog**$^+$ und TROLL nicht integriert worden. Aus diesem Grunde haben wir hier nur die Basiskonzepte dieser Erweiterungen präsentiert und auf eine vollständige Formalisierung verzichtet.

3.6.2 Sprachkonzepte für homogene Objektklassen

Wie bereits erwähnt, werden in diesem Abschnitt nur homogene Objektklassen und Klassentypen behandelt. Die Spezifikation heterogener Klasen und Typen folgt später in Abschnitt 4.6.

Um einen homogenen Klassentyp zu beschreiben, muß ein Objektmodell festgelegt werden sowie der Namensraum spezifiziert werden. Die Spezifikation von Objektmodellen ist in den bisherigen Abschnitten bereits ausführlich behandelt worden. Zur vollständigen Beschreibung einer Objektklasse muß der Klassentyp sowie ein eindeutiger Klassenname festgelegt werden. In der Regel erfolgt die Definition einfacher homogener Klassen integriert mit der Spezifikation des zugehörigen Klassentyps. Sofern keine Verwechslung zu befürchten ist, wird beiden auch der selbe Name zugeordnet.

Der syntaktische Aufbau einer derartigen integrierten Spezifikation von Klasse und Typ wurde bereits in Abschnitt 3.1 anhand von Beispielen eingeführt. Zu beachten ist hierbei, daß die Attributfestlegungen der Namensraumfestlegung in der Objektspezifikation wie andere (konstante !) Attribute verwendet werden dürfen. In dieser integrierten Schreibweise, wird eine Objektklasse wie folgt definiert :

object class `ObjektKlassenName`
  data types *importierte Datentypen (für Namensraumfestlegung);*
  identification *Namensraumfestlegung;*
  template *Objektbeschreibung;*
end object class `ObjektKlassenName;`

Nach dem Schlüsselwort **template** wird ein Objektmodell mit den bereits vorgestellten Spezifikationsmechanismen definiert. Basierend auf importierten Datentypen, wird die Namensraumfestlegung durch das Schlüsselwort **identification** (in anderen Sprachversionen auch **surrogate** oder **key**) eingeleitet. Aufgabe der Namenraumfestlegung ist es, eindeutig die Trägermenge eines abstrakten Datentyps festzulegen, die die Menge *ID* der Objektidentifikatoren bildet.

Die Form der Namensraumfestlegung ist an Schlüsseldefinitionen z.B. semantischer Datenmodelle angelehnt. Der festgelegte Datentyp wird somit als Instanzierung einer parametrisierten Tupelkonstruktion festgelegt.

Definition 3.6.6 Die **identification**-Klausel einer Klassenspezifikation für eine Objektklasse `NAME` legt basierend auf dem Datentypimport die Komponenten der Namensraumfestlegung fest. Syntaktisch wird sie wie folgt notiert :

> **object class** `NAME`
>   **data types** $d_1, \ldots, d_n$;
>   **identification**
> $\texttt{attribut}_1$: $\texttt{sort}_1$;
>   ...
> $\texttt{attribut}_k$: $\texttt{sort}_k$;

Die Wertebereiche $\texttt{sort}_j$ müssen hierbei jeweils Sorten eines der importierten Datentypen d_i sein. Wir unterscheiden Sorten und Datentypen, da eine Datentypdefinition mehrere Sorten als Wertemengen festlegen kann.

Diese Komponenten bestimmen den folgenden Datentyp $|\texttt{NAME}|$ basierend auf einer Instanzierung eines parametrisierten Datentypkonstrukturs für k-wertige Tupel[3] :

> **sort** $|\texttt{NAME}|$:= **tuple**($\texttt{sort}_1, \ldots, \texttt{sort}_k$);
> **based on** $d_1, \ldots, d_n$;
> **operations**
> $\texttt{attribut}_1$: $|\texttt{NAME}| \to \texttt{sort}_1$;
>   ...
> $\texttt{attribut}_k$: $|\texttt{NAME}| \to \texttt{sort}_k$;
>   `NAME`: $\texttt{sort}_1 \times \ldots \times \texttt{sort}_k \to |\texttt{NAME}|$;
> **predicates**
>   =: $|\texttt{NAME}| \times |\texttt{NAME}|$;

Wir geben hier nur die Signatur des Namensraumdatentyps an; auf die naheliegenden Gleichungen zur Festlegung des Zusammenspiels zwischen der Generierungsfunktion `NAME` und den Selektionsfunktionen der Attribute wurde verzichtet. Die Trägermenge der Sorte $|\texttt{NAME}|$ definiert den Namensraum *ID*. □

Diese Festlegungen eines Namensraums für einen allgemeine Klassendefinition sollen nun anhand eine konkreten Beispiels verdeutlicht werden.

Beispiel 3.6.7 Die Namensraumfestlegung der Objektklasse `BOOK` (Beispiel 3.2.3 von Seite 51) wird wie folgt spezifiziert :

[3] Die Diskussion von Spezifikation und Semantik von parametrisierten abstrakten Datentypen kann u.a. in Lehrbüchern über algebraische Spezifikation abstrakter Datentypen wie [EM85, EGL89] gefunden werden.

```
object class BOOK
   data types string;
   identification
      Title: string;
      FirstAuthor: string;
   template ...
end object class BOOK;
```

Als Namensraumdatentyp erhalten wir nun den folgenden Datentyp

```
sort |BOOK| := tuple(string, string);
based on string;
operations
   Title: |BOOK| → string;
   FirstAuthor: |BOOK| → string;
   BOOK: string × string → |BOOK|;
predicates
   =: |BOOK| × |BOOK|;
```

der die Signatur und die Konstruktion des Datentyps |BOOK| festlegt. □

Dieser Festlegung nach ist der Aufbau von Objektidentifikatoren transparent nach außen, d.h., beim Import des Namensraumdatentyps werden gleichzeitig die Selektionsfunktionen und die Generierungsfunktion importiert. Wir werden sehen, daß diese Funktionen oft in Objektspezifikationen benötigt werden. Andererseits ist es oft sinnvoll, die innere Struktur von Objektidentifikatoren zu verbergen, um ihre explizite Manipulation zu verhindern. Dies kann durch eine explizite *Einkapselung* erreicht werden.

Definition 3.6.8 Steht nach dem Schlüsselwort **identification** als weiteres Schlüsselwort das Wort **encapsulated**, wird der Namenraumdatentyp wie oben festgelegt und kann auch mit dieser Signatur *lokal* in der Spezifikation der Objektklasse benutzt werden.

Wird er aber in einer anderen Spezifikation als Datentyp importiert, enthält seine Signatur nur das Prädikat =. Insbesondere die Selektionsfunktionen sind nicht mehr auf Identifikatorwerte anwendbar. □

Im Beispiel 3.2.4 der Spezifikation der COPY-Objektklasse wurde das konstante Attribut Of syntaktisch als Teil der Namensraumfestlegung aufgeführt. Auch wenn diese Festlegung primär in der Unterscheidung zwischen konstanten und dynamischen Objektattributen für die Festlegung z.B. expliziter Objektinklusion begründet ist (vergl. Abschnitt 4.5.1), kann die Behandlung konstanter Attribute auch in die Namensraumdefinition integriert werden, in dem konstante Attribute als weitere Funktionen der Form

KonstantesAttribut: |NAME| → **sort**(KonstantesAttribut) zu der Definition hinzugefügt werden [SSE87].

Eine weitere sinnvolle Erweiterung ist es, sogenannte *Schlüsselbedingungen* (engl. *key constraints*) zuzulassen, die als zusätzliche Bedingungen bei der Konstruktion des Namensraums berücksichtigt werden müssen. Ein Beispiel hierfür wäre "Die Werte des Schlüsselattributs `FirstAuthor` müssen mit einem Großbuchstaben beginnen und höchstens 30 Zeichen lang sein".

Literaturhinweise

Eine gut verständliche Einführung in die Grundlagen der Prädikatenlogik basierend auf der Idee abstrakter Datentypen bietet das Buch von H.-J. Kreowski [Kre91]. Weitere empfehlenswerte deutsche Bücher neueren Datums zu diesem Thema sind [Sch89] und [Sie90].

Die vorgestellte lineare temporale Logik ist an die temporalen Logiken angelehnt, die von Z. Manna und A. Pnueli in [MP81, MP92] vorgestellt wurden. Die Definition der semantischen Strukturen für temporale Logik ist an die Präsentation in [Saa88, SL89] angelehnt.

Verschiedene semantische Modelle für Objekte werden in mehreren Arbeiten von H.-D. Ehrich, A. Sernadas und anderen vorgestellt [ESS88, ESS89, EGS90, ESS90, SEC90, ES91, SE91, ESS92, SJE92]. Die Rolle von Objekten in der Spezifikation von Informationssystemen wird in [SFSE89] diskutiert.

Die verwendeten Beispiele für Objektspezifikationen sind an der Beispielmodellierung in [JSS91, SJ91] orientiert, die in der Sprache **Oblog$^+$** spezifiziert ist. Die verwendete Sprache TROLL ist ausführlich in [JSHS91] sowie in [JHSS91, JSH91, SJ92] beschrieben. Beide Sprachen basieren auf den in [SSE87] vorgestellten Konzepten.

Kapitel 4

Beziehungen zwischen Objekten

Bislang haben wir uns mit der Beschreibung isolierter Objekte beschäftigt. In einem Informationssystem stehen aber Objekte in vielfältiger Beziehung zueinander — sie kommunizieren über gemeinsame Ereignisse, sie sind Teil anderer, komplexer Objekte, sie sind spezielle, eventuell nur zeitweilige Rollen anderer Objekte, und anderes mehr.

Dieses Kapitel beschäftigt sich mit den verschiedenen Arten, in denen Objekte und Objektklassen zueinander in Beziehung stehen können, und deren formalen Spezifikation. Im Gegensatz zu isoliert stehenden Objektspezifikationen, werden wir keine umfassende formale Semantik für alle Konzepte geben können und uns statt dessen auf einige ausgewählte Konstruktionen beschränken.

4.1 Arten von Objektbeziehungen

Wir beginnen die Diskussion der verschiedenartigen Objektbeziehungen mit einer informellen Auflistung der zu betrachtenden Abstraktionskonzepte. Viele dieser Konzepte wurden bereits in Kapitel 2 diskutiert.

- Die insbesondere in objektorientierten Ansätzen wohl verbreiteteste Beziehung zwischen Objektklassen und deren Instanzen ist der Aufbau einer *Klassenhierarchie mit Vererbung.* Mit diesem Begriff sind eine Reihe von semantischen Konstruktionen verbunden, die genauer behandelt werden müssen :

  - Auf der Ebene der Objektbeschreibungen und Klassentypen beschreibt eine Klassenhierarchie eine (optionale) Erweiterung der Signatur (mehr Attribute oder Ereignisse) sowie eine Verfeinerung der Objektspezifikation (eingeschränktere Lebensläufe, mehr Integritätsbedingungen, etc.). Wir bezeichnen diese Erweiterung unter Beibehaltung bisheriger Spezifikationsbestandteile als *syntaktische Vererbung* in einem *Typverband.*

  - Auf der Ebene der Klassenzustände und der aktuellen Instanzen haben wir eine Untermengenbeziehung zwischen den aktuellen Klassenpopulationen. Ein Objekt einer Basisklasse in einer spezielleren Klasse behält seine Identität und seine Historie und somit den aktuellen Zustand als Teil des spezielleren Objekts. Wir sprechen in diesem Zusammenhang von *semantischer Vererbung* in einem *Klassenverband.*
  - Die Einordnung in eine Klassenhierarchie kann auf zwei Arten gesteuert werden. Erfolgt der Wechsel in eine Klasse aufgrund eines Ereignisses dynamisch zur Laufzeit, so sprechen wir von *Phasen* oder *Rollen* eines Objekts. Erfolgt die Einordnung aufgrund eines (statischen) Prädikats über dem Objektzustand, so sprechen wir von *Spezialisierungen.* In einem Klassenverband können beide Arten beliebig miteinander kombiniert werden.

- Die *Generalisierung* erzeugt ebenfalls eine neue Klasse in einem Klassenverband, indem jeweils mehrere existierende Klassen zu einer neuen Klasse generalisiert werden. Im Gegensatz zu Phasen und Spezialisierungen erfolgt die Konstruktion aber entgegengesetzt zur Untermengenbeziehung auf den Klassenpopulationen. Das Ergebnisse einer Generalisierung ist in der Regel eine *heterogene Objektklasse.*

- Aus semantischen Datenmodellen und den sogenannten Nicht-Standard-Datenbankanwendungen insbesondere im Büro und technischen Bereich kommt der Begriff der *komplexen* Objekte. Wir unterscheiden wieder mehrere semantische Konzepte, die unter diesem Begriff subsumiert werden können.

  - Direkte *Objektinklusion* bestimmt zur Spezifikationszeit daß ein Objekt als eingekapselter Teil eines anderen Objekts aufzufassen ist. Objektinklusion ist das semantische Primitiv zur Konstruktion allgemeiner komplexer Objektstrukturen.
  - Die Bildung von *Komponentenobjekten* als zustandsabhängige Konstruktion komplexer Objektstrukturen ermöglicht die Manipulation komplexer Objekte zur Laufzeit. Wir unterscheiden *disjunkte und nichtdisjunkte* Komponentenbildung je nachdem, ob die Komponenten nur lokal existieren oder aus einer global definierten Klasse entnommen wurden.

    Die Manipulation komplexer Komponenten erfolgt über implizit generierte zusätzliche Attribute und Ereignisse. Die Aggregationskonzepte fortschrittlicher Datenmodelle — etwa Mengenbildung oder Listenbildung — werden durch parametrisierte Konstruktionen unterstützt.

- Objekte *kommunizieren* miteinander, indem sie an gemeinsamen Ereignissen teilhaben oder gegenseitig Ereignisse aufrufen.

In den folgenden Abschnitten werden diese Konzepte anhand von Beispielen erläutert, die korrespondierenden Sprachkonzepte vorgestellt und die Semantik der Konstruktionen diskutiert. Die Reihenfolge folgt dabei dem logischen Zusammenspiel der semantischen Konzepte und nicht der obigen Auflistung.

4.2 Ereignisaufruf als Kommunikationsprimitiv

Wir beginnen unsere Diskussion der Beziehungen zwischen Objekten mit den Kommunikationsprimitiven des *Ereignisaufrufs* (engl. *event calling*) und der Definition *gemeinsamer* oder auch *identifizierter Ereignisse* (engl. *event sharing*), da diese beiden Konzepte auch lokal in isolierten Objekten definiert werden können und diese Definition sich in natürlicher Weise auf zusammengesetzte Objekte übertragen lassen.

Ereignisse können aufgefaßt werden als Abstraktionen von Methodenausführungen objektorientierter Programmiersprachen bzw. elementarer Transaktionen in Datenbanksystemen. Die Kommunikation zwischen Objekten sollte entsprechend von den konkreten Konzepten des Methodenaufrufs der objektorientierten Programmierung abstrahieren. Diese Konzepte basieren auf operationalen Kontrollflußprinzipien angelehnt an den Prozeduraufruf imperativer Programmiersprachen bzw. konkreten Prozeßkommunikationsprotokollen.

4.2.1 Identifizierte Ereignisse und Ereignisaufruf

Wir abstrahieren zuerst von implementierten Konzepten indem wir als *synchrone symmetrische Kommunikation* die Definition gemeinsamer Ereignisse einführen. Die Basisidee hinter gemeinsamen Ereignissen ist, daß zwei Ereignisnamen (innerhalb eines Objektes oder auch verschiedener Objekte) semantisch ein und dasselbe Ereignis bezeichnen — etwa als Beispiel, daß das Ereignis `KaufeAuto(A)` einer Person mit Identifikator P das selbe Ereignis ist wie `WirdGekauftVon(P)` des Autoobjekts mit dem Identifikator A.

Definition 4.2.1 Zwei Ereignisse `Ereignis1` und `Ereignis2` werden als *identisch* deklariert durch die folgende syntaktische Notation

```
sharing
    Ereignis1 == Ereignis2
```

Sind beide Ereignisse in verschiedenen Objekten definiert (zum Beispiel in verschiedenen Komponentenobjekten innerhalb eines komplexen Objekts), so sprechen wir von *gemeinsamen Ereignissen* (engl. *shared events*). Allgemein sprechen wir vom Konzept der *Ereignisidentifikation*. □

Um das Konzept identischer Ereignisse zu formalisieren, können wir verschiedene Ansätze wählen, die hier nur kurz skizziert werden sollen :

- Da die Menge der Ereignisse in der formalen Semantik durch die Trägermenge eines abstrakten Datentyps E_{ob} modelliert werden, können wir eine Definition eines gemeinsamen Ereignisses direkt in eine *Gleichung einer algebraischen Spezifikation* für den Datentyp E_{ob} umwandeln. Als Ergebnis erhalten wir für verschiedene Ereignisnamen den selben Wert in E_{ob}.

- Auf der Ebene der Objektmodelle können wir eine Bedingung (e_1 `==` e_2) als zusätzliche Einschränkung für das Bilden von korrekten Lebensläufen $\lambda \in \Lambda$ auffassen :

  $$\forall(\lambda \in \Lambda)\,\forall(s_i \in \lambda)\ \ (e_1 \in s_i \Leftrightarrow e_2 \in s_i)$$

  Die Formel sagt aus, daß jeder Schnappschuß s_i eines Lebenslaufs $\lambda \in \Lambda$ entweder keines oder aber beide Ereignisse gleichzeitig enthalten muß.

- Der selben Idee folgend, kann die Angabe (e_1 `==` e_2) auch als temporale Formel formuliert werden :

  $$\mathbf{always}((\mathbf{after}(e_1) \Rightarrow \mathbf{after}(e_2)) \wedge (\mathbf{after}(e_2) \Rightarrow \mathbf{after}(e_1)))$$

  Eventuelle Parametervariablen der Ereignisse müssen dann noch allquantifiziert werden.

Die Identifikation von Ereignissen modelliert eine symmetrische und synchrone Kommunikation. Viele Anwendungen erfordern aber eine *asymmetrische* Kommunikation, d.h. einen gerichteten Kontrollfluß.

Beispiel 4.2.2 Betrachten wir ein einfaches Objekt `GiroKonto`, das unter anderen die folgenden Ereignisse habe :

```
events
   Einzahlen(money);
   Abheben(money);
   BuchungsZähler;
```

Bei jeder Abhebung oder Einzahlung soll das Ereignis `BuchungsZähler` aufgerufen werden, das die Attribute `AnzahlBuchungen` und `DatumDerLetztenBuchung` aktualisiert. Diese Aktualisierung wird durch ein spezielles Ereignis durchgeführt, um eine sonst notwendige doppelte, redundante und dadurch fehlerträchtige Spezifikation zu vermeiden.

Sollte die Kommunikation zwischen den Ereignissen durch Ereignisidentifikation erfolgen, wären alle drei Ereignisse gleichgesetzt, da die durch `==` induzierte Relation auf Ereignissen symmetrisch und transitiv ist. Zusätzlich würden mit der selben Begründung auch die Einzahlungen für alle konkreten Parameterwerte miteinander identifiziert. □

Die asynchrone Kommunikation kann durch das Konzept des *Ereignisaufrufs* (engl. *event calling*) beschrieben werden [SE91, JSHS91]. Ereignisaufruf ist eine Abstraktion der Konzepte Methoden- bzw. Prozeduraufruf imperativer Programmiersprachen. Ein Ereignis kann mehrere andere Ereignisse gleichzeitig aufrufen, die dann konzeptionell synchron ausgeführt werden.

Definition 4.2.3 Der *Aufruf* eines Ereignisses `Ereignis2` eines Objekts durch ein anderes Ereignis `Ereignis1` wird deklariert durch die folgende syntaktische Notation :

```
calling
   Ereignis1 >> Ereignis2;
```

Der Begriff "Ereignis" wird hier im Sinne eine Ereignis*terms* verwendet, d.h. die Ereignisparameter werden durch Variablen oder Datenterme besetzt. Variablen sind hierbei wie üblich implizit allquantifiziert. □

Beispiel 4.2.4 Die in Beispiel 4.2.2 beabsichtigte asymmetrische Kommunikation kann wie folgt deklariert werden :

```
calling
   variables m: money;
      Einzahlen(m) >> BuchungsZähler;
      Abheben(m) >> BuchungsZähler;
```

Für ein Beispiel mit einem Datenterm anstatt einer Variable formalisieren wir die Bedingung aus dem erweiterten `konto`-Beispiel 3.3.6 von Seite 56, daß das Einrichten eines Kontos immer mit einer gleichzeitigen Einzahlung von 5.- DM verbunden ist. Dies kann deklariert werden durch die folgende Definition.

```
calling
   Eröffnung >> Einzahlen(5);
```

Nehmen wir diese beiden Beispiele zusammen, haben wir ein Beispiel für die transitive Fortpflanzung des Ereignisaufrufs — `Eröffnung` ruft das Ereignis `Einzahlen`(5) auf, das wiederum den `Buchungszähler` aktiviert. □

Auch bei der Formalisierung des Ereignisaufrufs können wir die unterschiedlichen Ansätze benutzen, die bei der Ereignisidentifikation diskutiert wurden.

- Im Gegensatz zur Identifikation, können wir Gleichungen auf dem Ereignisdatentyp E_{ob} nicht direkt benutzen, da Gleichungen symmetrisch und transitiv sind.

  Eine mögliche *algebraische* Semantik kann allerdings dadurch ereicht werden, in dem statt atomarer Ereignisse ganze Mengen von parallel eintretenden Ereignissen als Trägermenge für E_{ob} gewählt werden. Dies kann durch die Aufnahme eines Operators "parallele Komposition" $||: E_{ob} \times E_{ob} \to E_{ob}$ in den Ereignisdatentyp E_{ob} realisiert werden. Ereignisaufruf kann dann wiederum durch Gleichungen auf mit $||$ gebildeten Ereignistermen ausgedrückt werden.

- Auf der Ebene der Objektmodelle können wir eine Bedingung (e_1 **>>** e_2) wieder als zusätzliche Einschränkung für das Bilden von korrekten Lebensläufen $\lambda \in \Lambda$ auffassen :

  $$\forall(\lambda \in \Lambda)\ \forall(s_i \in \lambda)\ \ e_1 \in s_i \Rightarrow e_2 \in s_i$$

  Die Formel sagt aus, daß jeder Schnappschuß s_i eines Lebenslaufs $\lambda \in \Lambda$ auch e_2 enthalten muß, falls er e_1 enthält.

- Der selben Idee folgend, kann die Angabe (e_1 **>>** e_2) auch als temporale Formel formuliert werden :

  $$\mathbf{always}(\mathbf{after}(e_1) \Rightarrow \mathbf{after}(e_2))$$

  Eventuelle Parametervariablen der Ereignisse müssen auch hier allquantifiziert werden.

In den Sprachen **Oblog**$^+$ und TROLL besteht die Möglichkeit, Identifikations- und Aufrufdeklaration mit einer zusätzlichen Bedingung zu versehen. Wir sprechen dann von *bedingter Ereignisidentifikation* bzw. *bedingtem Ereignisaufruf* (engl. *conditional sharing* und *conditional calling*).

Definition 4.2.5 Sowohl Identifikations- als auch Ereignisaufrufe können mit einer zusätzlichen *Bedingung* versehen werden. Die Bedingung ist eine Formel der vergangenheitsgerichteten temporalen Logik.

Eine *bedingte Ereignisidentifikation* wird notiert als :

sharing
```
{ Bedingung } ==> Ereignis1 == Ereignis2
```

Ein *bedingter Ereignisaufruf* wird notiert als :

```
calling
  { Bedingung } ==> Ereignis1 >> Ereignis2
```

Auch hier stehen `Ereignis1` und `Ereignis2` für Ereignis*terme*. Die erweiterte Semantikfestlegung z.B. in temporaler Logik erfolgt in naheliegender Weise durch eine zusätzliche Implikation. □

Abschließend ist eine Bemerkung zur asynchronen Kommunikation und ihre Integration mit den vorgestellten Konzepten sicherlich angebracht. Basierend auf synchroner Kommunikation, kann asynchrone Kommunikation durch zusätzliche *Kanalobjekte* realisiert werden, die ein vorgegebenes Kommunikationsprotokoll als internen Prozeß realisieren und eventuell einen Nachrichtenpuffer mit internen Attributen verwirklichen [HJ92]. Da eine derartige Simulation immer möglich ist, wurden bisher keine expliziten Sprachprimitive für asynchrone Kommunikation in die Sprachen **Oblog**$^+$ und Troll aufgenommen.

4.2.2 Ereignisaufruf in Objektspezifikationen

Die bisherigen Beispiele und deren Diskussion zeigten den Einsatz von Ereignisaufruf analog zur Verwendung des Funktionsaufrufs in Programmiersprachen. Die deklarative Semantik des Ereignisaufrufs ermöglicht aber einen Einsatz dieses Konzepts in Spezifikationen, der weit über die Möglichkeiten des prozeduralen Funktions- oder Prozeduraufrufs hinausgeht. Im folgenden Abschnitt sollen die Einsatzmöglichkeiten des synchronen Ereignisaufrufs anhand einfacher Beispiele vorgeführt werden.

Das erste betrachtete Anwendungsbeispiel ist die Spezifikation von Ketten von Aufrufen, bei der ein aufgerufenes Ereignis wiederum andere Ereignisse aufruft. Im allgemeinen Fall sind sogar rekursive Kette von Aufrufen möglich — und diese lassen sich für elegante Spezifikationen etwa von kaskadierenden Ereignissen einsetzen. *Rekursiver Ereignisaufruf*, d.h. ein Ereignis, das sich selber mit geänderten Parametern aufruft, ist in der Regel mit bedingtem Ereignisaufruf verbunden um die Rekursion abbrechen zu lassen.

Beispiel 4.2.6 Als Beispiel für eine rekursive Kette von Ereignisaufrufen und deren sinnvollem Einsatz in Spezifikationen betrachten wir die folgende Spezifikation eines `DezimalZählers`. Ein Dezimalzähler zählt das Auftreten eines Ereignisses `Zähle` in Dezimaldarstellung, wobei die einzelnen Dezimalstellen durch einen Attributgenerator `Stelle(nat)` modelliert werden.

```
object DezimalZähler
  template
    attributes
      Stelle(nat):nat;
```

```
    events
       birth ErzeugeZähler;
       Zähle;
       SetzeStelleWeiter(nat);
    valuation
       [ ErzeugeZähler ] Stelle(n) = 0;
       { Stelle(n) < 9 } ==>
          [ SetzeStelleWeiter(n) ] Stelle(n) = Stelle(n) + 1;
       { Stelle(n) = 9 } ==>
          [ SetzeStelleWeiter(n) ] Stelle(n) = 0;
    calling
       Zähle >> SetzeStelleWeiter(1);
       { Stelle(n) = 9 } ==>
          SetzeStelleWeiter(n) >> SetzeStelleWeiter(n+1);
end object DezimalZähler;
```

Das vorgestellte Beispiel realisiert einen Dezimalzähler mit unendlich vielen Stellen[1] modelliert durch den Attributgenerator `Stelle`. Für jede einzelne Stelle des Dezimalzählers ist ein Ereignis `SetzeStelleWeiter` definiert, das diese Stelle um 1 weitersetzt (modulo 10). Der Überlauf einer Stelle führt zum Aufruf vom Ereignis `SetzeStelleWeiter` für die nächsthöhere Stelle. Der Zähler zählt das Auftreten des Ereignis `Zähle` dadurch, daß das Ereignis `Zähle` die erste Stelle des Zählers durch den Aufruf von des Ereignisses `SetzeStelleWeiter(1)` weitersetzt. □

Ein weiterer interessanter Aspekt des obigen Beispiels ist, daß die Anzahl der aufgrund von Ereignisaufruf gleichzeitig eintretender Ereignisse erstens nicht beschränkt ist (beliebig viele Ziffern 9 müssen synchron auf 0 weitergeschaltet werden) und daß zweitens die konkrete Anzahl der gleichzeitig eintretenden Ereignisse zustandsabhängig ist, diese Ereignismenge also nicht zur Spezifikationszeit explizit in ein Einzelereignis, das den Effekt der parallelen Komposition simuliert, transformiert werden kann.

Bemerkung 4.2.7 Da Ereignisaufruf auch den Aufruf von Geburtsereignissen bedeuten kann, kann (neben der Initialisierung unendlich vieler Attribute) das Ergebnis eines Aufrufs auch die simultane Erzeugung *unendlicher Objektstrukturen* sein. Ein sinnvolles Beispiel hierfür wäre eine Initialisierung eines unendlichen Gittes aus parallel arbeitenden Automaten.

Während unendliche Objektstrukturen zur Spezifikationszeit durchaus Sinn machen können, bereiten sie zur Ausführungszeit natürlich Probleme. Neben einer Trans-

[1] Die Initialisierung unendlich vieler Attribute (die Variable `n` ist allquantifiziert!) in der ersten Auswertungregel ist in der Spezifikationsphase als Konstrukt erlaubt, muß aber in einer späteren Implementierung natürlich anders realisiert werden.

formation in endliche Strukturen während des Verfeinerungsprozesses kommt als Alternative auch ein Ausführungkonzept in Frage, das eine Art 'lazy evaluation' anbietet, wie sie etwa aus funktionalen Programmiersprachen mit unendlichen Objekten bekannt ist, in Frage. □

Das obige Beispiel zeigte den Einsatz von rekursivem Ereignisaufruf, um zustandsabhängig durch die Kumulation von Einzelereignissen einen Effekt zu erreichen, der durch Einzelereignisse nur sehr umständlich erreichbar wäre (in diesem Beispiel etwa durch bedingte Auswertungsregeln mit einer Allquantifizierung über niedrigere Zählerstellen in der Bedingung). Da Ereignisaufruf nicht auf den Aufruf eines *einzelnen* Ereignisses beschränkt ist, können ähnliche Effekte durch den *simultanen* Aufruf vieler Ereignisse gleichzeitig erreicht werden.

Beispiel 4.2.8 Wir modifizieren das Beispiel 4.2.6 dahingehend, daß wir die Initialisierung der Stellen nicht explizit durch eine Auswertungsregel für das Ereignis `ErzeugeZähler` sondern durch den simultanen Aufruf von expliziten Ereignissen `SetzeStelle` ersetzen.

```
object DezimalZähler2
   template
      attributes
         Stelle(nat):nat;
      events
         birth ErzeugeZähler;
         Zähle;
         SetzeStelle(nat,nat);
         SetzeStelleWeiter(nat);
      valuation
         [ SetzeStelle(n,m) ] Stelle(n) = m;
      calling
         ErzeugeZähler >> SetzeStelle(n,0):
         Zähle >> SetzeStelleWeiter(1);
         { Stelle(n) < 9 } ==>
            SetzeStelleWeiter(n) >> SetzeStelle(n, Stelle(n) + 1);
         { Stelle(n) = 9 } ==>
            SetzeStelleWeiter(n) >> SetzeStelle(n,0);
         { Stelle(n) = 9 } ==>
            SetzeStelleWeiter(n) >> SetzeStelleWeiter(n+1);
end object DezimalZähler2;
```

Ein weiterer interessanter Aspekt dieses Beispiels ist, daß in der neuen Spezifikation des Dezimalzählers alle expliziten Änderungen des `Stelle`attributs nun durch ein einzelnes Ereignis `SetzeStelle` realisiert werden können. Diese Lösung hat den

Vorteil einer lokalen Kontrolle aller Änderungen dieses Attributs durch ein einzelnes Ereignis. □

Bisher wurde der Einsatz von simultanen Ereignisaufrufen innerhalb eines Objektes diskutiert. In den folgenden Abschnitten wird dieses Konzept primär zur *Kommunikation zwischen Objekten* eingesetzt. In diesem Zusammenhang entspricht simultaner Ereignisaufruf einer Art *Multicasting-Kommunikationsbeziehung*, d.h. ein Sender sendet seine Nachricht simultan an mehrere Empfänger. Ein typisches Beispiel ist ein `GehaltsAbteilung`-Objekt, daß bei allen `Angestellten` eine `GehaltsErhöhung` durchführen will.

Zu beachten hierbei ist, daß weiterhin eine synchrone Kommunikation modelliert wird, d.h. ein simultaner Aufruf kann nur erfolgen, wenn alle beteiligten Objekte existieren und auch bereit zum Ausführen des Ereignisses sind. Vorstellbar ist hier eine Spracherweiterung um einen *schwachen* Ereignisaufrufmechanismus (engl. *weak calling*), der einer Nachricht an alle Objekte weitergibt die gerade empfangsbereit sind.

Der schwache Ereignisaufruf kann durch den vorgestellten Aufrufmechanismus und zusätzliche zustandsabhängige Prädikate simuliert werden, etwa nach dem folgenden Muster:

```
{ obj.Exists and obj.EnabledEvent } ==>
      Calling.Event >> obj.Event;
```

Der Term `obj` sei hier ein objektbezeichnender Term, eventuell mit einer freien Variable `O`. Das Prädikat **exists** wird genau dann wahr falls der Term `obj` ein bereits kreiertes Objekt bezeichnet. Ein derartiges Prädikat kann implizit in der Spezifikationssprache definiert oder explizit als Klassenattribut spezifiziert werden.[2] Das Attribut **EnabledEvent** muß derart spezifiziert sein, daß es genau dann wahr ist wenn alle Vorbedingungen des Ereignisses `Event` erfüllt sind.

4.2.3 Kontrollfluß versus Datenfluß

Ein weiterer interessanter Aspekt des vorgestellten Aufrufkonzepts ist die Behandlung von *Datenfluß bei der Parameterübergabe*. Da Ereignisaufruf eine synchrone Kommunikation modelliert, ist ein Datenfluß prinzipiell in beide Richtungen möglich — Variablen als Parameter der Ereignisterme können sowohl eine Wertübergabe als Aufrufparameter als auch ein vom aufgerufenen Objekt zurückgeliefertes Ergebnis bezeichnen. Mit anderen Worten, der Datenfluß kann entgegengesetzt zum Kontrollfluß erfolgen.

[2] Die syntaktische Schreibweise müßte bei beiden Realisierungen allerdings anders lauten — etwa als Klassenprädikat mit einen Objektidentifikator als Parameter. Zur Erläuterung des Prinzips wurde hier obige intuitive Schreibweise gewählt.

Ein typisches Beispiel dafür ist eine Spezifikation etwa eines endlichen Kellerspeichers `FiniteStack`, bei dem das `Push`-Ereignis zwei Parameter hat, ein Parameter für das zu speichernde Ereignis und ein **bool**sches Attribut das einen Overflow anzeigt. Ruft nun ein Objekt `Ob` das `Push`-Ereignis auf, so wird der Wert des ersten Parameter durch den Aufruf bestimmt während der Wert des zweiten Parameters durch den Kellerspeicher gesetzt wird, um eine eventuelle Fehlersituation anzuzeigen.

Soll ein derartiger Aufruf mit den bisher vorgestellten Sprachkonstrukten formuliert werden, so könnte dies wie folgt formuliert werden:

```
calling
   Ob.Speichere(n) >> FiniteStack.Push(n,true);
```

Dieser Ansatz aber versagt, falls wir einen Wert, etwa die Anzahl aktuell gespeicherter Werte, als Ergebnis zurückgeben wollen. Die Parameterrückgabe kann nur als Parameter des aufrufenden Ereignisses modelliert werden, wenn wir synchrone Kommunikation als Kommunikationsparadigma verwenden. Die folgende Modellierung formuliert nun nicht unbedingt vollständig die gewünschte Kommunikationsbeziehung:

```
calling
   Ob.Speichere(n,anz) >> FiniteStack.Push(n,anz);
```

Wie gefordert, haben wir eine synchrone Kommunikation der beiden parametrisierten Ereignisse — aber die Information über den *Datenfluß* ist in der Spezifikation nicht enthalten, obwohl diese Information sowohl in der Spezifikationsanalyse als auch in der Implementierungsphase eminent wichtig ist.

Betrachten wir gar eine Aufrufbeziehung, in der ein als Wert zurückgebener Parameter kein Parameter eines aufrufenden Ereignisses ist. Hierzu betrachten wir eine Erweiterung des obigen Beispiels um ein Ereignis `AuftragsEingang`, das neben anderen Parametern als Parameter die Auftragsnummer `n` hat, die auf dem `FiniteStack` abgelegt werden sollen.

Die naheliegende Modellierung durch

```
calling
   Ob.AuftragsEingang(x,y,n) >> Ob.Speichere(n,anz);
   Ob.Speichere(n,anz) >> FiniteStack.Push(n,anz);
```

ist nicht korrekt, da alle Variablen allquantifiziert sind ! Die erste **calling**-Klausel ruft somit unendlich viele Ereignisse auf, da die Variable `anz` allquantifiziert ist und nicht auf der linken Seite der Klausel durch einem konkreten Wert substitutiert wurde.

Eine erste Lösung des Problems bietet auch hier der schwache Ereignisaufruf, so daß nur diejenigen Aufrufe instantiiert werden, die aktuell möglich sind (d.h. in diesem Fall derjenige Aufruf, der den korrekten Rückgabewert hat). Der Nachteil hierbei ist allerdings, daß in dem Fall, daß das aufgerufene Objekt aus anderen Gründen nicht kommunikationsbereit ist bzw. gar nicht existiert, gar kein Aufruf stattfindet ohne daß dabei eine Fehlersituation auftritt.

Die aufgeführten Beispiel legen es nahe, die besondere Situation des Datenflusses *entgegen* der Kontrollflußrichtung syntaktisch besonders zu notieren. Ein derartiges syntaktisches Sprachkonzept folgt auch dem Entwurfsprinzip, alle für eine Anwendungssituation relevanten Informationen im konzeptionellen Entwurfsprozeß zu erfassen.

Prinzipiell gibt es mehrere Möglichkeiten der syntaktische Behandlung von zum Kontrollfluß inversem Datenfluß, die als Sprachkonstrukt in eine objektorientierte Spezifikationssprache eingebaut werden können:

- In der Spezifikationssprache Troll [JSHS91] werden die Ereignisparameter in der Objektsignatur durch die Schlüsselworte **in** und **out** markiert, um die Richtung des Datenflusses zu kennzeichnen. Das Schlüsselwort **in** bezeichnet Parameter, die als *Ein*gabe dem Objekt von außen mitgeteilt werden, während **out**-Parameter vom Objekt selber bestimmt werden (also einen potentiellen Datenfluß entgegen die Aufrufrichtung bedeuten können).

- Ein alternatives Sprachkonzept kennzeichnet den Datenfluß explizit in einer Aufrufdefinition, etwa (in der Notation an CSP [Hoa85] angelehnt) durch die Symbole ! und ?.

  Das obige Beispiel kann dann folgendermaßen formuliert werden

  ```
  calling
     Ob.AuftragsEingang(x,y,!n) >> Ob.Speichere(?n,?anz);
     Ob.Speichere(!n,?anz) >> FiniteStack.Push(?n,!anz);
  ```

  Bemerkenswert ist daß die Markierung des ersten Parameters des Ereignisses `Speichere` unterschiedlich markiert ist, je nach dem dieses in der Rolle als aufrufendes oder aufgerufenes Ereignis auftritt. Dies zeigt daß dieser Ansatz ausdrucksfähiger als die zuerst diskutierte Signaturmarkierung ist.

- In der Terminologie der Prozeßtheorie entspricht ein Datenfluß von außen einer externen Wahl (engl. *external choice*). Das obige Problem kann demnach durch einen *Prozeßaufruf* anstatt einem Ereignisaufruf gelöst werden:

  ```
  calling
     Ob.AuftragsEingang(x,y,n) >>
        choice(anz:nat) Ob.Speichere(n,anz);
     Ob.Speichere(n,anz) >> FiniteStack.Push(n,anz);
  ```

Hierbei wird ein Operator **choice** benutzt, der die externe Wahl der Variable `anz` anzeigt.

Externe Wahl wird als Sprachkonzept in der Prozeßsprache angeboten, die als Teilsprache in TROLL integriert ist [JSHS91]. Der **choice**-Operator wird in TROLL syntaktisch als **case on** notiert.

Eine interessante Fragestellung ist die Analyse von Datenfluß in komplexen Ereignisaufrufen zur Spezifikationszeit und zur Laufzeit. Für einen gerichteten Graphen aus Ereignisaufrufbeziehungen muß analysiert werden, ob der Datenfluß zyklische Beziehungen aufbauen kann. Während zur Laufzeit bekannte Techniken der Datenflußanalyse eingesetzt werden können, kann dies zur Spezifikation (aufgrund bedingtem Ereignisaufruf !) oft nicht durchgeführt werden, da allein die Fragestellung *"Terminiert eine rekursive bedingte Ereignisaufrufkette ?"* unentscheidbar ist[3].

4.3 Sichere Objektinklusion als semantisches Basiskonzept

Bisher haben wir uns nur mit isolierten Objekten und der Kommunikation innerhalb eines Objektes beschäftigt. Die elementare Art Objekte in Bezug zueinander zu setzen ist die *Objektinklusion* unter Berücksichtigung der lokalen Objekteinkapselung. Diese Einkapselung bedeutet in unserem Ansatz insbesondere, daß lokale Attribute nur durch lokale Ereignisse verändert werden dürfen. Um diese Einkapselung hervorzuheben, sprechen wir auch von *sicherer Objektinklusion*.

Objektinklusion bedeutet konzeptionell, daß ein Objekt a ein anders Objekt b als Teilobjekt enthält. Diese 'Enthaltenseinbeziehung' hat sowohl eine syntaktische als auch eine semantische Dimension :

- Auf der syntaktischen Ebene haben wir eine Inklusion auf den Signaturen, wobei eventuell Umbenennungen nötig sind um Namenskonflikte zu vermeiden (z.B. können alle syntaktische Namen von b durch den Namen von b als Präfix ergänzt werden). Es gilt also
$$\Sigma'_b \subseteq \Sigma_a$$
wobei gilt
$$\Sigma'_b \equiv \Sigma_b$$

[3]Das Beispiel 4.2.6 zeigt, wie ein Ereignis im Zusammenspiel mit einem Attributgenerator zu beliebig langen Berechnungsfolgen führen kann. Die Unentscheidbarkeit kann anhand folgender Argumentation deutlich gemacht werden. Ein Attributgenerator eines komplexen Datentyps kann eine unendliche Zustandsfolge modellieren, während der zugehörige Ereignisgenerator einen allgemeinen Berechnungsschritt auf Zuständen ausführen kann. Der Beweis der Terminierung der Aufrufkette entspricht dann dem Beweis des Halteproblems.

nach eventuellen Umbenennungen, d.h. Σ_b' ist isomorph zu Σ_b. In den folgenden Notationen benutzen wir die Notation Σ_b auch für Σ_b' und ignorieren die eindeutigen Umbenennungen in Notationen und Definitionen, um diese nicht unnötig komplex werden zu lassen.

- Auf der semantischen Ebene haben wir eine sichere Inklusion der Objektmodelle, d.h.
  - jeder Teillebenslauf von b, der in a beobachtet werden kann, ist ein korrekter Lebenslauf des b-Objekts,
  - und interne Attribute von b werden nicht durch Ereignisse von a korrumpiert.

  Diese Bedingungen stellen sicher, daß ein Objekt als Teilobjekt eines anderen Objekts sich so verhält, wie es — als isoliertes Objekt ! — auch spezifiziert wurde.

Die Formalisierung der syntaktischen Dimension ist kein Problem, aber die semantische Komponente erfordert doch einigen formalen Aufwand. Wir beginnen mit der Formalisierung der 'Beobachtung von Teillebensläufen eines Komponentenobjekts innerhalb eines komplexen Objekts'.

Definition 4.3.1 Gegeben seien zwei Mengen X_1 und X_2 von Ereignissen mit $X_1 \subseteq X_2$. Ferner sei λ ein Lebenslauf über X_2 und s ein Schnappschuß über X_2. Die *Reduktion* des Schnappschusses s auf das Alphabet X_1, notiert als $s \downarrow X_1$, ist definiert als

$$s \downarrow X_1 \quad := \quad \{e | e \in s \wedge e \in X_1\}$$

Die *Einschränkung von λ auf X_1*, notiert als $\lambda \downarrow\downarrow X_1$, ist wie folgt definiert. Für einen nichtleeren Schnappschuß $\lambda = s \circ \lambda'$ gilt

$$(s \circ \lambda') \downarrow\downarrow X_1 \quad := \quad \begin{cases} \lambda' \downarrow\downarrow X_1 & \text{falls } s \downarrow X_1 = \{\} \\ (s \downarrow X_1) \circ (\lambda' \downarrow\downarrow X_1) & \text{andernfalls} \end{cases}$$

Diese rekursive Definition muß für endliche Lebensläufe um die folgende Bedingung erweitert werden, die eine Terminierung für den leeren Lebenslauf ϵ garantiert :

$$\epsilon \downarrow\downarrow X_1 = \epsilon$$

Die Einschränkung auf X_1 streicht also Ereignisse, die nicht in X_1 enthalten sind, aus dem ursprünglichen Lebenslauf über X_2 heraus und entfernt zusätzlich leere Schnappschüsse aus dem Ergebnis. □

Basierend auf diesen Notationen, können wir jetzt die semantischen Einschränkungen bei der sicheren Objektinklusion formalisieren.

Definition 4.3.2 Ein Objektmodell b ist *eingebettet* in ein Objektmodell a, wenn die folgenden vier Bedingungen erfüllt sind :

1. Die Ereignismengen von b sind in denen von a enthalten, also
$$X_b \subseteq X_a$$

2. Die Einschränkungen von a-Lebensläufen auf X_b sind korrekte b-Lebensläufe :
$$\forall(\lambda_a \in \Lambda_a)\ \exists(\lambda_b \in \Lambda_b)\ \ (\lambda_a \downarrow\downarrow X_b) = \lambda_b$$

3. Auch die entsprechenden Attributmengen sind ineinander enthalten :
$$A_b \subseteq A_a$$

4. Eine Beobachtung der Attribute von b innerhalb von a ergibt dasselbe Ergebnis als würde man b isoliert beobachten :
$$\forall(\lambda_a \in \Lambda_a)\ \forall(\tau_a \in \mathbf{Prefix}(\lambda_a))\ \ \alpha_a(\tau_a) \downarrow A_b = \alpha_b(\tau_a \downarrow\downarrow X_b)$$
wobei die Notation $\downarrow A_b$ die Restriktion der Beobachtungsmenge auf Attribute in A_b bedeutet. Der Anteil $\alpha_a(\tau_a) \downarrow A_b$ beschreibt den "b-Anteil" in einer a-Beobachtung, und $\alpha_b(\tau_a \downarrow\downarrow X_b)$ den Effekt der b-Ereignisse auf die b-Attribute als ob b isoliert betrachtet würde.

Die Einbettung wird auch als *sichere Objektinklusion* bezeichnet. □

Sichere Objektinklusion ist ein Spezialfall von strukturerhaltenden Abbildungen zwischen Objektmodellen, sogenannten *Objektmorphismen*. Objektmorphismen spielen eine wichtige Rolle in der direkten Übertragung von objektorientierten Konzepten auf semantische Objektmodelle. Als vertiefende Literatur hierzu seien insbesondere die Veröffentlichungen von H.-D. Ehrich und A. Sernadas zu diesem Thema empfohlen [SE91, ES91].

Die sichere Objektinklusion spielt als semantische Grundlage eine wichtige Rolle in den folgenden Abschnitten. Explizite Sprachkonstrukte für Objektinklusion werden in Abschnitt 4.5.1 diskutiert.

4.4 Klassenhierarchien mit Vererbung

Der Aufbau einer Klassenhierachie mit Vererbung ist eine weitverbreitete Konstruktion sowohl in (semantischen) Datenmodellen als auch objektorientierten Ansätzen.

Die Basisidee ist dabei, daß ausgehend von einer Basisklasse eine neue, spezialisiertere Klasse konstruiert wird. Die spezialisiertere Klasse *erbt* dabei Eigenschaften von der Basisklasse. Die verschiedenen Ansätze in Datenmodellen und objektorientierter Programmierung differieren allerdings sehr stark bezüglich dem eingesetzten Vererbungskonzept. Im ersten Abschnitt 4.4.1 werden verschiedene Ansätze für Vererbung diskutiert und die in dem hier vorgestellten Ansatz realisierten Vererbungskonzepte vorgestellt.

Es folgen zwei Abschnitte, die die zwei in den Sprachen **Oblog$^+$** und TROLL durch explizite Sprachmittel unterstützten Konzepte Rollen- bzw. Phasenbildung und Spezialisierung vorstellen.

4.4.1 Syntaktische und semantische Vererbung

Vererbung kann als eine gerichtete, azyklische Beziehung zwischen Objekten, Klassen oder zugehörigen Typen aufgefaßt werden. Der verbreiteteste Fall ist der Aufbau einer *Vererbungshierarchie* bei dem jede Klasse von höchstens einer anderen Klasse erbt. Wird diese Einschränkung aufgegeben, sprechen wir von einem *Vererbungsverband mit Mehrfachvererbung*. Wir fordern hierbei die Verbandeigenschaft, daß Mehrfachvererbungen im Graphen wieder auf eine gemeinsame Basisklasse zurückführen. Im Gegensatz zum hier vorgestellten Ansatz, ist diese Eigenschaft in vielen anderen objektorientierten Ansätzen immer durch die Existenz eine Basisklasse `Object` gewährleistet, die transitiv als Basisklasse aller anderen Klassen fungiert.

In der objektorientierten Programmierung ist das Hauptziel der Vererbung die *Wiederverwendung von Schnittstellen und ausprogrammierten Methoden.* Schnittstellendefinitionen werden hierbei in der Regel unverändert übernommen, während bei Methoden eine Neuimplementierung stattfinden kann. Mit anderen Worten, die Schnittstelle (oder Objektsignatur in unserer Sprechweise) wird unverändert vererbt, während die Methoden (die Semantik) geändert werden können.

Das zugrundeliegende Vererbungskonzept dieser Art der Wiederverwendung bezeichnen wir als *syntaktische Vererbung.* Syntaktische Vererbung für Objektspezifikationen beinhaltet

- eine Signatureinbettung der Basisklasse in die konstruierte Klasse sowie
- eine Einbettung der Spezifikations*texte* mit der Möglichkeit des Überschreibens.

Syntaktische Vererbung ist auf der Spezifikationsebene und nicht auf der Ebene der Instanzen angesiedelt. Syntaktische Vererbung setzt somit also Objektbeschreibungen oder Klassentypen miteinander in Verbindung. Oft wird in diesem Zusammenhang von einem *Typverband mit syntaktischer Vererbung* gesprochen. Konzepte wie Objektinstanzen oder Klassenpopulationen werden durch syntaktische Vererbung nicht berührt, wie das folgende Beispiel andeutet.

Beispiel 4.4.1 Wir betrachten ein Beispiel aus der objektorientierten Programmierung und benutzen auch die dort verwendete Terminologie. Gegeben sei eine Klasse `Quadrat` mit einem Attribut `Seitenlänge_a` und zwei Methoden `BerechneUmfang` und `Zeichne`. Der Umfang wird als $4 *$ `Seitenlänge_a` berechnet.

Eine zweite Klasse `Rechteck` kann jetzt die Attribute und Methoden von `Quadrat` erben und zusätzlich ein Attribut `Seitenlänge_b` neu definieren. Die ererbte Methode `BerechneUmfang` wird neu definiert zu $2 *$ `Seitenlänge_a` $+ 2 *$ `Seitenlänge_b`. □

In semantischen Datenmodellen hingegen wird beim Aufbau einer Klassenhierarchie (durch die **is-a**- Beziehung) zwar auch die syntaktische Vererbung von Attributen unterstützt, aber die Klassenhierarchie hat zusätzlich eine Bedeutung für die Instanzen in den Klassen — eine Unterklasse hat als aktuelle Ausprägung eine Teilmenge der aktuellen Population der Oberklasse. Vererbung wird also auch auf der Ebene der Instanzen und Klassen definiert und nicht allein auf der (syntaktischen) Typebene. Wir bezeichnen dieses Konzept darum als *semantische Vererbung in einem Klassenverband.*

Bei der semantischen Vererbung in einem Klassenverband haben wir

- syntaktische Vererbung der Signatur,
- semantische Vererbung der Objektinstanzen in Form einer sichern Objektinklusion,
- und eine Teilmengenbeziehung auf den Klassenpopulationen.

Eine Unterklasse enthält also eine Untermenge der Instanzen der Oberklasse als jeweilige Population, und jede enthaltenen Objektinstanz erbt syntaktisch die Schnittstellendefinition und semantisch die Oberklasseninstanz als sicher eingebettetes Teilobjekt. Insbesondere werden bei semantischer Vererbung somit auch — im Gegensatz zu syntaktischer Vererbung — Attribut*werte* und der interne Objektzustand vererbt.

Beispiel 4.4.2 Betrachten wir das Beispiel 4.4.1 aus der Sicht der semantischen Vererbung, so stellen wir fest, daß auf der Instanzenebene die umgekehrte Beziehung gilt — Quadrate sind eine spezielle Untermenge der Rechtecke ! Die semantische Vererbung läuft also im Klassenverband in die umgekehrte Richtung wie in Beispiel 4.4.1. Als Erweiterung der Unterklasse haben wir dann etwa die Bedingung `Seitenlänge_a` $=$ `Seitenlänge_b`. □

Auch wenn wir bisher hauptsächlich von Typ- und Klassenverbänden mit Vererbung gesprochen haben, können diese Konzepte natürlich auch auf einzelne isolierte Objektbeschreibungen und Objektinstanzen angewendet werden.

Mehrfachvererbung (engl. *multiple inheritance*) kann in die vorgestellten Konzepte integriert werden. Namenskonflikte müssen dabei z.B. durch explizite Umbenennung gelöst werden. Auf der Ebene der Klassenpopulationen haben wir bei Mehrfachvererbung eine Schnittmengenbildung.

Die Sprachen **Oblog$^+$** und TROLL unterstützen semantische Vererbung durch die Abstraktionskonzepte Rolle und Spezialisierung, die in den folgenden beiden Abschnitten eingeführt werden. Syntaktische Vererbung spielt eine Rolle bei der Benutzung von Spezifikationsbibliotheken und der syntaktischen Wiederverwendung von Objektbeschreibungen und Klassentypen, die als Erweiterungen dieser Sprachen vorgesehen sind.

Die hier diskutierte semantische Vererbung basiert auf der Vererbung semantischer Ereignisse und Attributbeobachtungen, also auf einer modelltheoretischen Semantik. S. Braß und U. Lipeck stellen in [BL91] einen Ansatz zur semantischen Vererbung vor, der auf Vererbungsbeziehungen zwischen Objekt*theorien* basiert und somit einer primär logikbasierten Semantik zuzuordenen ist. Dieser Ansatz ermöglicht insbesondere eine elegante Integration des Überschreibens von Ereignissen und Attributen (engl. *overriding*) im Zusammenspiel mit semantischer Vererbung.

4.4.2 Objektphasen und Rollen von Objekten

Ein Objekt einer Oberklasse spielt eine speziellere *Rolle* in einer Unterklasse, wenn es zeitweilig in dieser anderen Klasse enthalten ist und dort eine verfeinertes Verhalten zeigt oder eine erweiterte Schnittstelle aufweist. Anstatt von dem Begriff der Rolle zu sprechen, wird oft auch von *Phasen* von Objekten oder auch (zeitweiligen) Sichten (engl. *views*) auf Objekte gesprochen [SSE87], die allerdings nicht mit der Sichtdefinition in Datenbanksprachen verwechselt werden sollten. Von einer Phase wird insbesondere dann gesprochen, wenn das spezialisierte Objekt zwar dieselbe Schnittstelle aber ein verfeinertes Verhalten aufweist.

Der Übergang in eine speziellere Rolle erfolgt durch ein ausgezeichnetes Ereignis, das den Phasenübergang einleitet. Dieses Ereignis ist entweder in der Basisklasse definiert oder als Geburtsereignis der Rolle in der Rollenklasse deklariert. Diese beiden Möglichkeiten spiegeln die unterschiedlichen Modellierungen wider ob der Phasenübergang unter lokaler Kontrolle des Basisobjekts oder unter globaler Kontrolle von außerhalb (z.B. durch Klassenereignisse oder globale Kommunikation) erfolgt.

Zusammengefaßt umfaßt die Idee der Deklaration von Objekten in einer zeitweiligen Rolle oder Phase

- die Deklaration des Phasenübergangs durch Angabe von Geburtsereignissen für die Rolle bzw. Phase,
- eine optionale Erweiterung der Objektsignatur bei der Rollendefinition,
- optionale weitere Einschränkungen des Verhaltens des Basisobjekts,

- sowie eine semantische Inklusion der Basisobjekte im erweiterten Objekt.

Bei der Betrachtung von Klassen kommt zusätzlich noch eine Untermengenbeziehung auf den aktuellen Populationen hinzu.

Aus dem Bereich der semantischen Datenmodellierung sind eine Reihe von Standardbeispielen bekannt, so `ANGESTELLTER` als Rolle von `PERSON` und `MANAGER` als Rolle von Objekten der Klasse `ANGESTELLTER`. Haben wir `PATIENT` als weitere Rolle von `PERSON`, können wir als einen Fall mit Mehrfachvererbung die Rolle `PATIENT-MANAGER` als gemeinsame Rolle von `MANAGER` und `PATIENT` definieren.

Definition 4.4.3 Die syntaktische Notation einer *Rollendefinition* in der Sprache **Oblog$^+$** wird für Klasssen durch die Deklaration

```
object class Klassenname
   view of Basisklasse / Basisklassen
   template ...
```

und eine zusätzliche Objektbeschreibung als Erweiterung vorgenommen. In dieser Erweiterung dürfen die Signaturelemente (Attributnamen, Datentypen, Ereignisnamen) der Basisklasse benutzt werden. Ausnahmen sind hierbei zusätzliche Auswertungregeln für geerbte Ereignisse, die verboten sind — derartige Regeln würden die Einkapselung der Basisobjekte durch sichere Objektinklusion verletzen.

Mit Basisklasse wird hier nicht die Wurzel der Rollenhierarchie bezeichnet, sondern der direkte Vorgänger im Vererbungsgraphen. In der Sprache TROLL wird eine Rollendefinition analog mit dem Schlüsselwort **role of** anstatt **view of** notiert.

Analog zu Klassen, kann eine Rollendefinition auch für Einzelobjekte vorgenommen werden :

```
object Objektname
   view of Basisobjekt
   template ...
```

Hier gelten dieselben Einschränkungen wie für Klassen. □

Im folgenden werden die Sprachmittel zur Definition von Rollen und ihre Semantik anhand mehrerer Beispiele erläutert.

Beispiel 4.4.4 Wir beginnen als Einstieg mit einem Fragment aus dem oben erwähnten Szenario mit Personen, Angestellten und Managern. Die Klasse `MANAGER` kann als Rolle von Angestellten wie folgt deklariert werden :

```
object class MANAGER
   view of ANGESTELLTER;
   template
      attributes
         OfficialCar : |CAR|;
      events
         birth BecomeManager;
         ...
      constraints
         Salary >= 5.000;
      ...
end object class MANAGER;
```

In diesem Beispiel fehlen noch Ereignisse zur Manipulation der Daten über den dem Manager zustehenden Dienstwagen (Attribut `OfficialCar`). Das Attribut `Salary` wird von `ANGESTELLTER` geerbt und kann zur Formulierung einer Integritätsbedingung benutzt werden. Alternativ kann die Notation `ANGESTELLTER.Salary` benutzt werden. □

Bei einer Rollendefinition wird keine neue Identifikation angegeben, da der Identifikationsmechanismus geerbt wird. Die Notationen |PERSON|, |ANGESTELLTER| und |MANAGER| sind also Namen für ein und denselben Datentyp als Namensraum.

Das folgende Beispiel ist dem im dritten Kapitel begonnenen Bibliotheksbeispiel entnommen. Das Beispiel wird im Abschnitt 4.5.1 fortgeführt.

Beispiel 4.4.5 Die Objektklasse `LIB_BOOK` modelliert die Klasse der Bücher, die in der Bibliothek tatsächlich mit Exemplaren vorhanden sind. Dazu wird die Klasse `LIB_BOOK` als Rolle der insgesamt erfaßten Bücher `BOOK` definiert :

```
object class LIB_BOOK
   view of BOOK;
   template
      data types nat;
      attributes
         no_available: nat;
      events
         birth acquire;
         dec_copies;
         inc_copies;
      valuation
         {no_available> 0} ==> [dec_copies] no_available = no_available - 1;
         [inc_copies] no_available = no_available + 1;
         [aquire] no_available = 0;
end object class LIB_BOOK;
```

Hier wird nur eine unvollständige Spezifikation von `LIB_BOOK` vorgestellt. Die vollständige Spezifikation findet sich in Beispiel 4.5.4. Die erste Auswertungsregel ist ein Beispiel für eine etwa in der Sprache TROLL erlaubte *bedingte Auswertungsregel*, eine Erweiterung der im vorigen Kapitel vorgestellten Sprachkonstrukte für Auswertungregeln bei der das Attribut nur geändert wird, falls eine gegebene Bedingung erfüllt ist.

Die Objekte der Klasse `LIB_BOOK` haben als zusätzliches Attribut die Anzahl aktuell verfügbarer Exemplare `no_available`, und mehrere zusätzliche Ereignisse die die Ausleihvorgänge verwalten. Das Ereignis `aquire` initialisiert ein Bibliotheksbuch, und die Ereignisse `dec_copies` und `inc_copies` zählen die Anzahl der verfügbaren Exemplare herunter bzw. herauf. □

Die bisherigen Beispiel waren alle mit einer Signaturerweiterung verbundene Rollendeklarationen. Da neben Signaturerweiterungen auch weitere *Einschränkungen* des Verhaltens der Basisobjekte möglich sind, machen auch Deklarationen ohne Signaturerweiterung einen Sinn, die oft zur Unterscheidung von Rollen als *Phasen* bezeichnet werden. Phasen definieren in gewissem Sinne zeitweilig zusätzliche Integritätsbedingungen für Objekte.

Beispiel 4.4.6 Wir betrachten die Klasse `LOANED_COPY` als Beispiel für eine Phase der Basisklasse `COPY` (Beispiel 3.2.4 von Seite 51) ohne zusätzliche Signaturerweiterung. Als Erweiterung haben wir eine zusätzliche Einschränkungen der korrekten Lebenszyklen : ausgeliehene Exemplare müssen auch wieder zurückgegeben werden[4].

```
object class LOANED_COPY
  view of COPY
  template
    events
      birth COPY.CheckOut(|USER|,date,nat);
      death COPY.Return;
    liveness
      COPY.Return;
end object class LOANED_COPY;
```

Die Klasse `LOANED_COPY` ist auch ein Beispiel für die lokale Kontrolle des Übergangs in die Phase — der Phasenübergang wird durch die Ereignisse `COPY.CheckOut`(U,d,n) und `COPY.Return` der `COPY`-Klasse hervorgerufen. □

Bei *Mehrfachvererbung* bei der Rollendeklaration mit mehreren Eingangsklassen haben wir auf der Populationenebene eine Untermenge der Schnittmenge der beteiligten Eingangsklassenpopulationen, was natürlich nur Sinn macht, da die Eingangsklassen selber direkt oder indirekt Rollen (oder Spezialisierungen) ein und derselben

[4] Dieses Beispiel kann natürlich auch mit einer bedingten Lebendigkeitsbedingung modelliert werden.

Basisklasse sein müssen. Aus diesem Grund ist auch der Namensraumdatentyp der neuen Klasse wohldefiniert. Auf der Signatur- und Instanzenebene haben wir — nach Auflösung etwaiger Namenskonflikte — die Vereinigung der Objekteigenschaften.

Beispiel 4.4.7 Als Beispiel für Mehrfachvererbung bei Rollen hatten wir bereits die Klasse `PATIENT-MANAGER` als gemeinsame Rolle der Klassen `MANAGER` und `PATIENT` erwähnt. Die Klasse `PATIENT-MANAGER` erhält die Vereinigung der Erweiterungen für `MANAGER` und `PATIENT` als geerbte Eigenschaften. Instanzen in `PATIENT-MANAGER` müssen sowohl in `MANAGER` als auch in `PATIENT` enthalten sein. Die gemeinsame Rolle wird deklariert durch

```
object class PATIENT-MANAGER
   view of PATIENT, MANAGER;
   ...
end object class PATIENT-MANAGER;
```

Wird zum Beispiel in `PATIENT` und in `MANAGER` jeweils ein neues Attribut `Status` mit unterschiedlicher Bedeutung neu definiert, muß der Namenskonflikt durch die ausschließliche Verwendung der Namen `PATIENT.Status` und `MANAGER.Status` aufgelöst werden. □

Der vorgestellte Ansatz unterscheidet sich insbesondere durch die semantische Vererbung der Instanzen von der Philosophie anderer objektorientierter Ansätze. Das in objektorientierten Programmiersprachen populäre Überschreiben von Methoden (engl. *overriding*) widerspricht der semantischen Vererbung so wie sie hier benutzt wird. Eine mögliche Teillösung dafür ist es in der Sprache zuzulassen, daß ein neues Ereignis mit gleichem Namen definiert werden kann wobei auf das geerbte Ereignis mittels des Klassenpräfix der Basisklasse zugegriffen werden kann.

Wie bereits erwähnt, bietet der in [BL91] vorgestellte Ansatz eventuell eine auf der Ebene der logischen Beschreibungen angesiedelte allgemeinere Lösung dieser Fragestellung.

4.4.3 Spezialisierung

In semantischen Datenmodellen ist eine 'automatische Klassifizierung' anhand von Attributwerten ein verbreitetes Modellierungskonzept. Ein Standardbeispiel ist die Klassifizierung von `PERSON`-Objekten in die Klassen `MANN` und `FRAU` anhand des Werts eines Attributs `Geschlecht`. Wir bezeichnen diese Art der Klassifizierung als *Spezialisierung*. Diese Klassifizierung ist konzeptionell Attribut-gesteuert im Gegensatz zur Ereignis-gesteuerten Klassifizierung der Rollenkonstruktion.

In semantischen Datenmodellen gibt es keine Einschränkung für die Art der Attribute die die Klassifizierung steuern. Wir beschränken die Definition der Spezialisierungsbedingung hingegen auf Formeln über *konstanten Attributen* (inklusive Schlüsselattributen). Der Grund ist daß bei allgemeinen Formeln jede Attributänderung als (beabsichtigten oder unbeabsichtigten) Effekt einen Wechsel der Klassifizierung erzeugen kann. Aus methodischen Gründen fordern wir, daß derartige dynamische Klassifizierungen direkt durch Ereignisse modelliert werden müssen.

Definition 4.4.8 Eine *Spezialisierung* wird definiert durch die Angabe einer Basisklasse und eine Spezialisierungsbedingung, die eine nichttemporale Formel über den konstanten Attribute (inklusive Schlüsselattributen) der Basisklasse ist. Zusätzlich wird die Erweiterung des Objekttemplates angegeben.

Syntaktisch wird eine Spezialisierung in **Oblog$^+$** wie folgt notiert :

```
object class Klassenname
   specializing from Basisklasse where  Formel über konstante Attribute;
   template
      ...
end object class Klassenname;
```

Semantisch werden Spezialisierungen durch eine äquivalente Rollenkonstruktion definiert. Dazu wird zusätzlich ein künstliches **birth**-Ereignis für die Rolle eingeführt, das vom Geburtsereignis der Basisklasse genau dann aufgerufen wird wenn die Spezialisierungsbedingung gilt. In der Sprache TROLL fehlt das Wort **from** in der syntaktischen Notation. □

Wir erläutern die Spezialisierungskonstruktion anhand einer weiteren Erweiterung unseres Bibliotheksbeispiels.

Beispiel 4.4.9 Bibliotheksbenutzer `USER` werden definiert als Basisklasse deren Objekte die Tätigkeiten von Bibliotheksbenutzern modellieren: `USER` melden sich bei der Bibliothek an (`subscribe`) und wieder ab (`unsubscribe`), leihen Bücher aus (`borrow`) und geben sie zurück (`return`), und fordern Bücher an (`request`).

Ein konstantes Attribut `copies_allowed` gibt die Höchstzahl von Büchern an, die der Benutzer ausleihen kann, und ein Attribut `borrowed` enthält die Identifikatoren aller aktuell ausgeliehener Bücher.

```
object class USER
   data types string;
   identification
      uid: string;
   constants
```

```
        copies_allowed: {3, 10};
    template
        data types |COPY|, date, nat, |LIB_BOOK|;
        attributes
            borrowed: set(|COPY|);
        events
            birth subscribe;
            death unsubscribe;
            borrow(|COPY|, date, nat);
            active return(|COPY|);
            active request(|LIB_BOOK|);
        valuation
            variables C: |COPY|, d: date, n: nat;
            [borrow(C,d,n)] borrowed = insert(C, borrowed);
            [return(C)] borrowed = remove(C, borrowed);
        safety
            variables C: |COPY|, d: date, n: nat;
            {not(in(C, borrowed)) and card(borrowed) < copies_allowed}
                borrow(C,d,n);
            {sometime(after(borrow(C,d,n)))} return(C);
            {empty(borrowed)} unsubscribe;
        initiatives
            variables C: |COPY|, d: date, n: nat;
            {after(borrow(C,d,n))} ==> return(C);
end object class USER;
```

Das Schlüsselwort **active** markiert in der Sprache TROLL diejenigen Ereignisse, die ein USER-Objekt aus eigener Initiative ausführen kann.

Anhand der Werte des konstanten Attributs copies_allowed können wir jetzt zwei Klassen von Benutzern unterscheiden : Institutsangehörige (STAFF_USER) dürfen maximal 10 Bücher ausleihen, studentische Ausleiher (STUDENT_USER) nur 3.

```
object class STAFF_USER
    specializing from USER where copies_allowed = 10;
    template
        data types |LIB_BOOK|;
        events
            get_message(|LIB_BOOK|);
        liveness
            variables LB: |LIB_BOOK|, C: |COPY|;
            { after(get_message(LB))
                and in(C,borrowed) and C.of = LB }  ==>  return(C);
end object class STAFF_USER;
```

Institutsbenutzer haben keine Beschränkung der Ausleihdauer, müssen aber darum auf eine erhaltenen Anforderungsmitteilung (`get_message`) reagieren. Studentische Benutzer hingegen dürfen Bücher höchstens für 21 Tage ausleihen :

```
object class STUDENT_USER
   specializing from USER where copies_allowed = 3;
   template
      safety
         variables C: |COPY|, d: date, n: nat;
         { n ≤ 21 } borrow(C,d,n);
end object class STUDENT_USER;
```

Die letzte Klassifizierung ist ein Beispiel für eine Klassifizierung ohne Signaturerweiterung analog zur Phasenkonstruktion. □

Spezialisierungen und Rollenkonstruktionen können in einem Klassenverband beliebig gemischt auftreten. Mehrfache Spezialisierungen analog zur Rollenkonstruktion sind ebenfalls möglich, da sie äquivalent durch eine Rolle über mehreren Basisklassen plus eine anschließende Spezialisierung modelliert werden können.

Als abschließende Bemerkung sei erwähnt, daß der Semantikansatz der Definition von Spezialisierungen über bedingten Ereignisaufruf sich natürlich auch auf Spezialisierungsbedingungen über nichtkonstanten Eigenschaften anwenden läßt (*dynamische Spezialisierung*). Die Einschränkung auf konstante Attribute hat also tatsächlich hauptsächlich methodische Gründe — Seiteneffekte wie Klassifizierungsänderungen sollten immer explizit und nichtredundant deklariert werden. Auch können Initialisierungen von neuen Attributen der Spezialisierungsklassen bei dynamischer Spezialisierung extrem unübersichtlich und fehlerträchtig werden.

4.4.4 Erweiterungen des Rollenkonzepts um Instanzengeneratoren *

Eine interessante Erweiterung des bisher vorgestellten Rollenkonzepts ist die Möglichkeit der *Mehrfachinstanziierung* in einer Spezialisierungsklasse, bei dem einem Objekt der Basisklasse mehrere Objekte der Spezialisierung entsprechen können. Dies widerspricht dem bisherigen Spezialisierungs- und Rollenkonzept. Ein Beispiel ist die Klasse `ANGESTELLTER` als Spezialisierung von `PERSON`, wobei eine `PERSON` mehrfach in der Rolle eines Angestellten auftreten darf — etwa in Form von Teilzeitbeschäftigten bei mehreren Firmen.

Im bisher diskutierten Ansatz übernimmt die spezialisierte Klasse implizit den Identifikationsmechanismus der Basisklasse. Um Mehrfachinstanziierung unterstützen zu können, muß eine erweiterte Identifikation eingeführt werden. Wie auch in den bisherigen Sprachkonzepten, wird ein explizite Identifikationsmechanismus einer impliziten Generierung von Objektidentifikatoren vorgezogen, wie sie etwa in vielen objektorientierten Datenmodellen vorgezogen wird.

Definition 4.4.10 Eine Rollendeklaration mit *Instanzengenerator* ermöglicht Mehrfachinstanziierung in einer Unterklasse. Ein Instanzengenerator ist ein zusätzliches identifizierendes Attribut (oder auch mehrere), das nach dem Schlüsselwort **with identification** deklariert wird.

Der Instanzengenerator muß beim **birth**-Ereignis mit einem konkreten Wert besetzt werden, also in der Regel ein Parameter dieses Ereignisses sein. Der Instanzengenerator ist — als Teil des Schlüssels — ein konstantes Attribut. □

Prinzipiell sind auch Instanzengeneratoren bei Spezialisierungen denkbar — der Wert des Generatorattributs kann etwa aus den Elementen eines konstanten mengenwertigen Attributs gewählt werden. Wir verzichten hier auf einen konkreten Sprachvorschlag.

Beispiel 4.4.11 Als Beispiel für eine Rollendefinition mit Instanzengenerator geben wir das folgende Spezifikationsfragment einer Angestelltenrolle an, bei der eine Person bei mehreren Fimen gleichzeitig als Teilzeitkraft arbeiten kann. Der Einfachheit halber gehen wir davon aus, daß alle Datentypen bereits in `PERSON` importiert wurden.

```
object class ANGESTELLTER
   view of PERSON
   with identification Firma: |FIRMA|;
   template
      attributes
         ProzentDerArbeitszeit: integer;
      events
         birth Einstellung(Firma: |FIRMA|);
         ...
end object class ;
```

In diesem Beispiel wurde die Kurzschreibweise der Initialisierung von konstanten Attributen durch das Geburtsereignis übernommen, bei der die Initialisierung über benannte Parameter des Geburtsereignisses erfolgt. □

Streng genommen sind Unterklassen mit Instanzengeneratoren keine Unterklassen im eigentlichen Sinne mehr. Die Deklaration einer derartigen Klasse trägt aber mehr Modellierungssemantik als etwa eine unabhängige Klassendefinition mit explizitem neuen Namensraum und expliziter Objektinklusion, die eine äquivalente Beziehung ausdrücken kann.

Unabhängig von der Modellierungsintention, kann die Semantik einer Unterklasse mit Instanzengenerator durch andere diskutierte Sprachmittel auf verschiedene Arten ausgedrückt werden:

- Mehrfachinstanzen können zu Einzelinstanzen zusammengefaßt werden, etwa in dem alle neuen Signaturelemente mit einem zusätzlichen Parameter (dem Wert des Instanzengenerators) versehen werden.

  Dieser Ansatz hat gravierende Nachteile:

  - Die Lokalität der Einzelinstanzen geht verloren. Insbesondere entsprechen modellierte Objekte nicht den realisierten Objekten.
  - Weitere Spezialisierungen und Rollendefinitionen der Einzelinstanzen sind nicht direkt möglich.

- Eine Unterklasse mit Instanzengenerator kann durch eine unabhängige neue Klasse mit neuem Namensraum und expliziter Objektinklusion der Basisobjekte ausgedrückt werden (vergl. Abschnitt 4.5.1). Die zusätzliche Semantik muß durch explizite Integritätsbedingungen und Kommunikation ausgedrückt werden.

In den Sprachen TROLL und **Oblog$^+$** ist das Konzept von Unterklassen mit Instanzengeneratoren bisher nicht integriert worden. Die diesem Sprachvorschlag zugrundeliegende konzeptionelle Idee wurde durch die Ausführungen von R. Wieringa und W. de Jonge in [WdJ92] motiviert.

4.5 Komplexe Objekte

Komplexe Objekte beschreiben die Aggregation von Teilobjekten zu größeren Gesamtobjekten, wie sie insbesondere in semantischen Datenmodellen und Nicht-Standard-Datenbankanwendungen eine große Rolle spielen.

Semantische Grundlage für die Konstruktion komplexer Objekte ist die *sichere Objektinklusion* aus Definition 4.3.2, d.h. eine Objekt- / Teilobjektbeziehung die auf strukturerhaltenden Abbildungen zwischen kompletten Objektmodellen beruht. Komplette Objektmodelle aber bedeuten komplette Objektlebensläufe; und sichere Inklusion modelliert somit eine *statische* Beziehung zwischen Objekten die nicht dynamisch zur Laufzeit verändert werden kann.

Im folgenden ersten Teilabschnitt 4.5.1 werden Sprachmittel der Sprache TROLL diskutiert, die die direkte Deklaration statischer komplexer Objektstrukturen mittels sicherer Objektinklusion ermöglichen [JSHS91, HJS92]. Es folgen zwei weitere Teilabschnitte die sich mit Sprachkonzepten und semantischen Grundlagen dynamischer komplexer Objekte beschäftigen.

4.5.1 Explizite Objektinklusion

Die explizite Inklusion entspricht dem Basisprinzip der semantischen Vererbung, d.h. neben der semantischen Vererbung von Signaturanteilen werden auch Objektmodelle an ein anderes Objekt 'vererbt'. Diese Auffassung ist, wie bereits erwähnt, weiter gefaßt als die Vererbungsidee objektorientierter Programmiersprachen. Aufgrund des zugrundeliegenden Basiskonzepts der semantischen Vererbung durch sichere Objektinklusion wurde in den Sprachen TROLL und **Oblog**$^+$ die Schlüsselworte **including** bzw. **inheriting** zur Kennzeichnung expliziter Inklusion gewählt. Wir verwenden die TROLL-Syntax, um Konfusion mit anderen Vererbungskonzepten, die bei dem Schlüsselwort **inheriting** assoziiert werden könnten, zu vermeiden.

Definition 4.5.1 Der einfache Fall der *expliziten Inklusion* ist, daß ein bestimmtes Objekt ein einzelnes anderes Objekt als Teilobjekt importiert. Dies wird syntaktisch mit dem Sprachmittel

including object `Objektspezifikation [as Komponentenname ];`

ausgedrückt. In TROLL lautet die Schlüsselwortfolge **including instance**. Die Objektspezifikation identifiziert eindeutig das zu importierende Objekt und hat eine der folgenden Formen :

- Direkte Angabe des `Objektnamens` eines isolierten Objekts.
- Spezifikation *eines einzelnen* Objekts aus einer Objektklasse in der Form

  `Identifikationsterm` **in** `Klasse`

  wobei `Identifikationsterm` ein Term des Namensraumdatentyps |`Klasse`| der Klasse `Klasse` ist, der ausschließlich über konstanten Objekteigenschaften gebildet ist.

Optional kann nach dem Schlüsselwort **as** eine Umbenennung bzw. eindeutige Namenszuordnung des importierten Objekts vorgenommen werden. □

Die Beschränkung auf konstante Objekteigenschaften beim Identifikationsterm ist notwendig, da nur *statische* Objektzusammensetzung deklariert werden darf.

Beispiel 4.5.2 Angenommen wir haben zwei isolierte Objekte `copy` und `book`, so können wir direkt spezifizieren daß `book` Teilobjekt von `copy` sein soll :

```
object copy
template
      including object book;
...
end object copy;
```

Die Signatur von `copy` wird hierbei erweitert um die Signatur von `book`, und das Objektmodell von `book` wird von dem von `copy` mittels sicherer Inklusion als Teilobjekt importiert.

Als zweites (etwas anwendungsnäheres) Beispiel betrachten wir den Fall, daß wir für jedes Bibliotheksexemplar in der Klasse `COPY` den zugehörigen Bucheintrag aus `BOOK` als Teilobjekt importieren wollen.

```
object class COPY
   data types nat, |BOOK|;
   identification
      DocNo: nat;
   constants
      Of: |BOOK|;
   template
      including object Of in BOOK
         as BookEntry;
   ...
end object class COPY;
```

Über den eindeutigen Namen `BookEntry` kann hierbei auch bei Namenskonflikten eindeutig auf die Signaturkomponenten des Teilobjekts zugegriffen werden. □

Der allgemeinere Fall der expliziten Objektinklusion erlaubt es, die gleichzeitige Inklusion ganzer Mengen von Objekten aus einer Klasse zu spezifizieren.

Definition 4.5.3 Der mengenwertige Fall der *expliziten Inklusion* bedeutet, daß ein Objekt eine Teilpopulation einer Objektklasse als Teilobjekte importiert. Dies wird syntaktisch mittels der Konstruktion

```
including Objektvariable in Klassenname
   [ where statische Selektionsbedingung ]
   [ as Komponentenname ];
```

ausgedrückt. Die statische Selektionsbedingung verwendet die `Objektvariable` um auf die (statischen) Eigenschaften der zu importierenden Objekte zuzugreifen. Auch statische Eigenschaften des importierenden Objekts können in dieser Formel auftauchen. Statische Eigenschaften sind im wesentlichen Werte von konstanten Attributen und Schlüsselattributen.

Optional kann wieder nach dem Schlüsselwort **as** eine eindeutige Namenszuordnung der Menge der importierten Objekte vorgenommen werden. □

Beispiel 4.5.4 Als Beispiel für den mengenwertige Fall der *expliziten Inklusion* betrachten wir eine Erweiterung von Beispiel 4.4.5, bei dem die Bibliotheksbücher der Klasse `LIB_BOOK` alle zugehörigen Exemplare aus `COPY` importieren.

```
object class LIB_BOOK
   view of BOOK;
   template
      data types nat;
      including C in COPY where C.of = SELF.id as COPIES;
      attributes
         no_available: nat;
      events
         birth acquire;
         dec_copies;
         inc_copies;
         hand_out(|USER|, |COPY|);
         request(|USER|);
         send_message(|STAFF_USER|);
      valuation
         {no_available > 0} ⇒ [dec_copies] no_available = no_available − 1;
         [inc_copies] no_available = no_available + 1;
         [aquire] no_available = 0;
      safety
         variables U: |USER|, C, C1: |COPIES|, SU: |STAFF_USER|
         {no_available > 0 and sometime(after(request(U)))} hand_out(U,C);
         {not(sometime(after(hand_out(U,C)))
            since last after(COPIES(C).return))}
               hand_out(U,C1);
         {no_available = 0 and
            sometime(after(request(U))) since last after(hand_out(U,C))}
               send_message(SU);
      interaction
         variables U: |USER|, C: |COPIES|, d: date, n: nat;
         calling
            COPIES(C).check_out(U,d,n) >> dec_copies;
            COPIES(C).return >> inc_copies;
            COPIES(C).get_copy >> inc_copies;
            hand_out(U,C) >> COPIES(C).check_out(U,d,n);
end object class LIB_BOOK;
```

Die Objektvariable **SELF** ist implizit innerhalb einer Objekt(klassen)spezifikation definiert und kann benutzt werden um als Präfix unmißverständlich auf die Signaturbestandteile des gerade spezifizierten Objekts zu verweisen. Das Attribut **id** ist als abkürzende Schreibweise definiert und bezeichnet den Identifikator des aktuellen Objekts. Hier ist der Term **SELF.id** also äquivalent zu `BOOK(Title, FirstAuthor)`.

Der neu definierte Name `COPIES` kann lokal in der Spezifikation wie ein globaler Objektklassenname benutzt werden, wobei implizit immer die aktuell importierte Objektpopulation angesprochen wird. Insbesondere ist auch der Datentyp |`COPIES`| als Teilnamensraum von |`COPY`| definiert. □

Wie bei allen Sprachkonzepten, die auf sicherer Objektinklusion beruhen, sind Auswertungsregeln für importierte Attribute verboten, aber zusätzliche Einschränkungen für Lebensläufe durchaus erlaubt. Dies basiert auf dem Grundkonzept, daß ein Objekt in Beziehung zu anderen Objekten sich allgemein so verhält wie spezifiziert, aber z.B. aufgrund von Kommunikation innerhalb komplexer Objekte eventuell nicht mehr alle seine Lebensläufe wirklich auftreten können.

4.5.2 Disjunkte komplexe Objekte

Viele neuartige Datenmodelle für strukturierte Informationen, so etwa das NF^2- oder das eNF^2-Datenmodell, unterstützen nur *disjunkte Datenstrukturen* als Modellierungskonzept, also baumartige Zugriffstrukturen mit Zugang über Datenbank'objekte' als Wurzel. Auf der konzeptionellen Ebene, auf der die semantischen Datenmodelle und auch die hier vorgestellte Objektspezifikation angesiedelt sind, entspricht dieser Idee das Konzept von *abhängigen Unterobjekten mit lokaler Identifikation*, also Objektstrukturen die ausschließlich als Teilobjekt existieren können und eine Objektidentität nur lokal innerhalb des Oberobjekts aufweisen.

Beim Aufbau disjunkter Objektstrukturen werden die Teilobjekte also nicht aus globalen Objektklassen entnommen, sondern sie werden lokal definiert. Ebenfalls lokal wird ein Identifikationsmechanismus definiert.

Die lokale Definition von von Objekten geschieht durch die Deklaration von *Teilobjektbeschreibungen* (auch engl. *subtemplates*) innerhalb von Objektbeschreibungen. Ein Subtemplate ist eine komplette Objektbeschreibung, die entweder importiert werden kann falls sie bereits definiert wurde, oder explizit spezifiziert wird.

Die lokale Objektidentifikation wird analog zu Ereignisparametern bzw. Attributgeneratoren deklariert. Subtemplates werden zusammen mit den Attributen und Ereignissen definiert und erhalten wie sie einen eindeutigen lokalen Namen. Dieser Name kann mit einen oder mehreren Parametern versehen werden, die mehrere Teilobjekte gleicher Struktur unterscheiden.

Beispiel 4.5.5 Als Beispiel für disjunkte Objektstrukturen betrachten wir den Dokumentenaufbau, bei dem ein Dokument unter anderem aus mehreren Kapiteln aufgebaut ist. Kapitel werden — lokal ! — durch eine Kapitelnummer identifiziert.

Wir beginnen mit der Spezifikation von Objektbeschreibungen für einzelne Kapitel, indem wir ein Template `KapitelTemplate` deklarieren.

```
template KapitelTemplate
   data types string, text ;
   attributes
      Überschrift : string ;
      Inhalt : text;
      Autor : |PERSON|;
   events
      birth InitialisiereKapitel (Überschrift: string);
      ...
 end template KapitelTemplate;
```

Für dieses Beispiel haben wir hier die TROLL-Syntax benutzt, die ausführlicher in [JHSS91] eingeführt wird. Wollen wir nun fragmentarisch eine Objektklasse `DOKUMENT` spezifizieren, die einzelne Kapitel als Unterobjekte besitzt, so können wir diese wie folgt deklarieren.

```
object class DOKUMENT
...
   subtemplates
      Kapitel(nat): KapitelTemplate;
...
end object class DOKUMENT;
```

Auf die einzelnen Ereignisse und Attribute der Kapitel können wir nun über den Namen der Unterstrukturen und Parameterwerte zugreifen, etwa auf die `Überschrift` des ersten Kapitels mittels

```
Kapitel(1).Überschrift
```

Als Beispiel für eine nicht parametrisierte Unterstruktur können wir zusätzlich ein besonderes Kapitel `Kurzfassung` einführen, für das wir eine zusätzliche Integritätsbedingung (außerhalb des Subtemplates !) deklarieren können.

```
object class DOKUMENT_2
...
   subtemplates
      Kurzfassung: KapitelTemplate;
      Kapitel(nat): KapitelTemplate;
   ...
   constraints
      Kurzfassung.Überschrift = ´Kurzfassung´
            or Kurzfassung.Überschrift = ´Abstract´;
...
end object class DOKUMENT_2;
```

Alle Unterobjekte müssen mit den expliziten lokalen **birth**-Ereignissen initialisiert werden. Vor diesen lokalen Geburtsereignissen werden alle zugehörigen lokalen Attribute als uninitialisiert behandelt. □

Syntaktisch bedeuten Subtemplates eine Erweiterung der Objektsignatur um neue Ereignisse und Attribute, deren Namen aus dem Subtemplate-Namen und dem Attribut- oder Ereignisnamen zusammengesetzt wird und eine Parameterliste besitzen, die sich aus der lokalen Identifikation der Unterobjektstrukturen und den eigentlichen Ereignis- und Attributparametern zusammensetzt.

Semantisch können wir disjunkte Objektstrukturen auf zweierlei Weise erklären :

- Die Definition von Subtemplates kann als eine strukturierte Beschreibung *eines* komplexen Templates mit strukturierten Namen aufgefaßt werden.

  Dieser Ansatz hat den Nachteil, daß die Strukturierung in Teilobjekte in der semantischen Modellbildung verloren geht. Auch muß die erwünschte sichere Einkapselung der Unterobjekte auf der syntaktischen Ebene garantiert werden.

- Eine mit lokalen Subtemplates strukturierte Objektbeschreibung kann in eine Beschreibung mit globalen Objektklassen umgewandelt werden [JHSS91]. Dafür werden für Unterstrukturen neue künstliche globale Klassen definiert, deren Identifikation sich aus der Identifikation der betrachteten Objektklasse und den Parametern der Subtemplates zusammensetzt. Die Unterstrukturen werden dann mittels expliziter Objektinklusion zum Gesamtobjekt zusammengesetzt.

Wir geben nur ein Beispiel für den zweiten Semantikansatz, da die Namensexpansion des ersten Ansatzes naheliegend durch Präfixbildung und Parameterverschieben erfolgen kann.

Beispiel 4.5.6 Für das obige `DOKUMENT`-Beispiel 4.5.5 können wir eine neue globale Klasse `DOKUMENT_KAPITEL` definieren, die die folgende Identifikation besitzt :

```
object class DOKUMENT_KAPITEL
   data types nat, |DOKUMENT|;
   identification
      DOKUMENT_id: |DOKUMENT|;
      Kapitel: nat;
...
end object class DOKUMENT_KAPITEL;
```

Über explizite semantische Inklusion werden dann diejenigen `DOKUMENT_KAPITEL`-Objekte in die `DOKUMENT`-Objekte importiert, die den gleichen Identifikator wie das Dokument als Wert des Schlüsselattributs `DOKUMENT_id` haben. □

Der vorgestellte Ansatz der Subtemplatedeklarationen ist auch dazu geeignet, *rekursive* disjunkte Strukturen zu modellieren. Als Beispiel betrachten wir eine Erweiterung der Dokumentmodellierung aus Beispiel 4.5.5.

Beispiel 4.5.7 Statt nur Kapitel als oberste Strukturierungsstufe zu modellieren, können wir das allgemeine Konzept eines `Abschnitts` eines Dokuments spezifizieren, der rekursiv Unterabschnitte enthalten kann.

```
template Abschnitt
   data types string, text ;
   subtemplates
      Unterabschnitt(nat) : Abschnitt;
   attributes
      Überschrift : string ;
      Inhalt : text;
      Autor : |PERSON|;
   events
      birth InitialisiereAbschnitt (Überschrift: string);
   ...
 end template Abschnitt;
```

Die rekursive Objektstruktur eines Dokuments hat als Blätter `Abschnitte`, für die keine `Unterabschnitte` durch ein Geburtsereignis explizit initialisiert wurden. □

Auch die oben angeführten Semantikfestlegungen lassen sich auf rekursive Strukturen erweitern, in dem für den identifizierenden Parameterwert bzw. den neue Objektnamensraum ein Datentyp definiert wird, der als Wertebereich die Menge der *endlichen Pfade durch einen attributierten Baum* hat.

Disjunkte Objektstrukturen können auch als lokale Objekte oder Objektklassen aufgefaßt werden, die in einem Objekt eingekapselt sind. Denkba wäre somit auch eine syntaktische Notation, die sich an dieser Sicht anlehnt und eventuell zu einem orthogonaleren Sprachvorschlag führen könnte, der sich enger an bereits eingeführte syntaktische Noatationen hält.

4.5.3 Konstruktoren für nichtdisjunkte komplexe Objekte

Die explizite Objektinklusion aus Abschnitt 4.5.1 erlaubt die Zusammensetzung komplexer Objektstrukturen aus Objekten globaler Klassen. Sie modelliert somit das Konzept beliebig verzeigerter *nicht-disjunkter* komplexer Objekte.

Im Vergleich mit Modellierungskonzepten vergleichbarer Datenmodelle für komplexe nicht-disjunkte Objekte, z.B. dem MAD-Modell [Mit88], hat dieser Vorschlag einige gravierende Nachteile in der Ausdrucksfähigkeit der Modellierungskonzepte :

- Die Zusammensetzung der komplexen Objekte ist *statisch* festgelegt, kann sich also zur Laufzeit nicht ändern.
- Die Zusammensetzung ist *wertegesteuert* im Gegensatz zu der ereignisgesteuerten Modellierung wichtiger Anwendungsgebiete — ein komplexer Bauteil in Ingenieuranwendungen z.B. wird aktiv zusammengesetzt.
- Die semantische Inklusion unterstützt als einzige Strukturierungskonzepte die (heterogene) Aggregation fester Kardinalität ("Tupelbildung") und die homogene Mengenabstraktion.

Ziel ist ein Konstruktionsmechanismus für nichtdisjunkte komplexe Objektstrukturen, der diese Nachteile nicht aufweist aber in den bisher vorgestellten Rahmen paßt. Ein derartiger Mechanismus sollte also

- es erlauben, Objektstrukturen *ereignisgesteuert* zu manipulieren, und
- die unterschiedlichen verbreiteten *Strukturierungskonzepte* unterstützen.

Als Konstruktionsmechanismus kann die *ereignisgesteuerte Komponentenbildung* eingesetzt werden, wie sie etwa in der Sprache TROLL angeboten wird [JHSS91]. Die Basisidee ist angelehnt an die Idee der parametrisierter Datentypkonstruktoren auf dem Gebiet der algebraischen Spezifikation abstrakter Datentypen [EM85]. Die Signatur einer durch Ereignisse veränderbaren komplexen Objektstruktur wird bei Verwendung eines parametrisierten Komponentenkonstruktors implizit vorgegeben, und die Semantik wird z.B. durch eine feste Spezifikationstransformation festgelegt.

Ein parametrisierter Komponentenkonstruktor ist z.B. **LIST** der als Parameter einen Klassennamen hat und als Komponenten Listen von Objektinstanzen dieser Klasse erzeugt. Mit der Verwendung dieses Konstruktors werden implizit Ereignisse und Attribute generiert, die die Manipulation und Beobachtung der Komponente ermöglichen.

Wir beginnen die Diskussion der parametrisierten Komponentenkonstruktion mit einem einführenden Beispiel.

Beispiel 4.5.8 Als Beispiel modellieren wir die komplexe Objektklasse `TEAM`, deren Objekte jeweils einen `Trainer` und mehrere `Spieler` als Instanzen aus der Klasse `PERSON` als Komponenten enthält. Die Klasse `PERSON` ist entsprechend dem folgenden Fragment spezifiziert.

```
object class PERSON
   data types ...;
   identification
      Name: string;
```

```
      Geburtstag: date;
   ...
end object class PERSON;
```

Die Klasse `TEAM` wird nun wie folgt deklariert, wobei die Spieler als *Liste* von `PERSON`en modelliert sind.

```
object class TEAM
   data types string;
   identification
      TeamName: string;
   template
      components
         Trainer : PERSON;
         Spieler : LIST(PERSON);
...
end object class TEAM;
```

Komponenten sind bei der Geburt eines Objekts nicht initialisiert. Initialisierung und Manipulation erfolgt ausschließlich über implizit generierte Ereignisse (und Attribute).

Für die Komponenten `Trainer` und `Spieler` haben wir u.a. die folgenden abgeleiteten Ereignisse :

`Trainer.ASSIGN(|PERSON|)` Setzen der `Trainer` Komponente.

`Spieler.APPEND(|PERSON|)` Eine neue `PERSON` als `Spieler` aufnehmen.

`Spieler.REMOVE(|PERSON|)` Eine `PERSON` aus der `Spieler`-Liste entfernen.

Als Parameter von Ereignissen sind natürlich nur Objekt*identifikatoren*, nicht aber die Objekte selber, erlaubt.

Als generierte Attribute für die Komponenten `Trainer` und `Spieler` haben wir u.a. die folgenden Attribute :

`Trainer.DEFINED`: boolean. Ist die `Trainer`-Komponente initialisiert ?

`Spieler.COUNT`: integer. Aktuelle Anzahl von `Spieler`n.

`Spieler.ID_SET`: **set**(|PERSON|). Menge der Identifikatoren der aktuellen `Spieler`.

Nicht nur Attribute und Ereignisse können als abgeleitete Konzepte zur Verfügung gestellt werden, sondern auch *Komponenten* (also Unterobjekte) selber, so etwa der erste `Spieler` unter dem Namen `Spieler.FIRST`.

Auf die Signaturbestandteile der Komponenten kann wie üblich durch die Verwendung von Pfadausdrücken zugegriffen werden. □

Nach diesem einführenden Beispiel werden die Sprachkonstrukte für die Instanziierung parametrisierter Komponentenkonstruktoren beispielhaft anhand dreier Konstruktoren und deren generierten Ereignisse, Attribute und Unterobjekte definiert.

Definition 4.5.9 Die Deklaration von *Komponenten* wird durch das Schlüsselwort **components** eingeleitet. Es folgt die Deklaration der einzelnen Komponenten :

components
`Componentname : Komponententyp;`

Der `Komponententyp` beschreibt die Instanziierung eines parametrisierten Komponentenkonstruktors mit aktuellen Klassen, so etwa **LIST**(`KlassenName`). Der elementare Konstruktor **SINGLETON** beschreibt Komponenten, die genau eine Instanz beinhalten, und wird syntaktisch vereinfacht als `KlassenName` an Stelle von **SINGLETON**(`KlassenName`) notiert.

Die folgende Aufstellung präsentiert Beispiele für parametrisierte Komponentenkonstruktoren (angelehnt an den TROLL-Sprachvorschlag [JHSS91]) zusammen mit einer Auswahl der generierten Attribute, Ereignisse und abgeleiteten Unterobjekten.

1. Wir beginnen mit elementaren Komponenten definiert durch den Komponentenkonstruktor **SINGLETON**(`Klasse`) bzw. dessen syntaktischer Kurzform '`Klasse`'. Der Komponente sei der Name `KompName` zugewiesen.

   Als generierte Attribute haben wir die folgenden Attribute :

   `KompName.DEFINED`: boolean. Abfrage ob die `KompName`-Komponente initialisiert ist.

   `KompName.ID`: |`Klasse`|. Identifikator der aktuellen `KompName`-Komponente.

   Als neues Ereignis wird das Ereignis `Name.ASSIGN` zur Initialisierung und Neusetzen der Komponente definiert :

   `KompName.ASSIGN`(|`Klasse`|). Setzen der `KompName`-Komponente.

   Unter dem Namen `KompName` kann auf das aktuelle Komponentenobjekt als Unterobjekt zugegriffen werden.

2. Mengenwertige Komponenten werden definiert durch die Instanzierung des Komponentenkonstruktors **SET** notiert als **SET**(`Klasse`). Der Komponente sei wieder der Name `KompName` zugewiesen.

   Als generierte Attribute haben wir die folgenden Attribute :

   `KompName.EMPTY`: boolean. Abfrage ob die Komponente Elemente enthält.

`KompName.COUNT`: integer. Anzahl der Elemente der Menge.

`KompName.ID_SET`: **set**(|`Klasse`|). Identifikatoren der Elemente der aktuellen Komponente.

Zur Manipulation der Menge werden folgende Ereignisse generiert :

`KompName.INSERT`(|`Klasse`|). Einfügen eines Elements in die `KompName`-Komponente.

`KompName.REMOVE`(|`Klasse`|). Löschen eines Elements aus der `KompName`-Komponente.

Unter dem Namen `KompName` kann auf das aktuelle Komponentenobjekt als Menge von Unterobjekten zugegriffen werden. Die Syntax des Zugriffs entspricht dem Zugriff auf eine Menge explizit importierter Unterobjekte (Abschnitt 4.5.1).

3. Listenwertige Komponenten können durch den Konstruktor **LIST**(`Klasse`) definiert werden. Der Komponente sei wieder der Name `KompName` zugewiesen.

   Als generierte Attribute haben wir die folgenden Attribute :

   `KompName.EMPTY`: boolean. Abfrage ob die Komponente Elemente enthält.

   `KompName.LENGTH`: integer. Länge der Liste.

   `KompName.ID`(nat): |`Klasse`|. Identifikator des i-ten Elements der aktuellen Liste.

   `KompName.ID_LIST`: **list**(|`Klasse`|). Liste der Identifikatoren der Elemente der aktuellen Komponente in der Reihenfolge der Elemente in dieser Liste.

   `KompName.ID_SET`: **set**(|`Klasse`|). Menge der Identifikatoren der Elemente der aktuellen Komponente. Dieses Attribut wird insbesondere zur Quantifizierung über alle Elemente einer listenwertigen Komponenete eingesetzt.

   Zur Manipulation der Liste werden folgende Ereignisse generiert :

   `KompName.APPEND`(|`Klasse`|). Anhängen eines Elements an die `KompName`-Liste.

   `KompName.ASSIGN`(integer,|`Klasse`|). Setzen des i-ten Elements der `KompName`-Liste auf ein neues Element (nur erlaubt falls $0 < i \leq$ `KompName.LENGTH`).

   `KompName.REMOVE_FIRST`. Löschen des ersten Elements aus der `KompName`-Komponente.

   `KompName.REMOVE`(|`Klasse`|). Löschen eines durch einen Identifikator identifiziertes Elements aus der `KompName`-Komponente.

Unter dem Namen `KompName.INSTANCE_SET` kann auf Instanzen in der aktuellen Komponentenliste als Menge von importierten Unterobjekten zugegriffen werden.

Als weitere generierte Unterobjekte haben wir die folgenden generierten Komponenten, auf die wieder mittels der Syntax aus Abschnitt 4.5.1 zugegriffen werden kann.

`KompName.FIRST`: (`Klasse`). Erstes Element der `KompName`-Liste.

`KompName.ELEMENTS`(integer): (`Klasse`). i-tes Element der `KompName`-Liste als Unterobjekt (nur für $0 < i \leq$ `KompName.LENGTH` initialisiert).

Sinnvoll sind sicherlich noch weitere Konstruktoren zur Beschreibung etwa von Multimengen, gerichteten Graphen oder Bäumen von Objektinstanzen, auf die hier nicht weiter eingegangen wird. □

Die Komponentenkonstruktion kann konzeptionell *nicht direkt* auf sichere Objektinklusion zurückgeführt werden, da diese prinzipiell statisch ist und keine Änderung der komplexen Objektstruktur zur Laufzeit erlaubt. Die Semantikfestlegung kann jedoch trotzdem auf diese Konzept zurückgeführt werde, indem eine zusätzliche generierte Objektklasse eingeführt wird.

Definition 4.5.10 Die *Semantik dynamischer Komponentenbildung* wird durch eine zweistufige statische sichere Objektinklusion definiert. Eine neue implizite Objektklasse wird definiert, die für *jede mögliche* Komponentestruktur statisch das komplexe Objekt über Inklusion festlegt. Der Identifikator dieser Klasse wird aus der Komponentendeklaration generiert, wobei etwa mengenwertige Komponenten der Klasse `Klasse` zu einem Schlüsselattribut des Typs **set**(|`Klasse`|) führen.

All diese neuen statischen komplexen Objekte werden nun statisch in das komplexe Objekt importiert. Zustandsabhängig ist nun genau dasjenige Objekt existent, das die (dynamische) aktuelle Komponentenstruktur widerspiegelt. Die generierten Ereignisse des Komponentobjekts zur Manipulation der komplexen Objektstruktur rufen jeweils ein Geburts- und ein Sterbeereignis der entsprechenden Instanzen dieser Klasse auf. □

Auch wenn wir diese Semantikfestlegung nicht vertieft diskutieren wollen, soll die Basisidee doch anhand eines Beispiels erläutert werden.

Beispiel 4.5.11 Wir betrachten hierzu das Beispiel 4.5.8, in dem ein `TEAM`-Objekt eine mengenwertige Komponente `Spieler` und eine elementare Komponente `Trainer` hat.

Als implizite neue Klasse wird die Klasse `STATIC_TEAM` wie folgt definiert :

```
object class STATIC_TEAM
   data types |TEAM|, |PERSON|, set(|PERSON|);
   identification
      TEAM_id: |TEAM|;
      Trainer_id: |PERSON|;
      Spieler_id_list: list(|TEAM|);
   template
      including object SELF.Trainer_id in PERSON as Trainer;
      including P in PERSON
         where is_element(P.id, Spieler_id_list)
         as Spieler;
...
end object class STATIC_TEAM;
```

Die Operationen **is_element** testet, ob ein Wert in einer Liste des passenden Typs enthalten ist.

Gemäß dieser Spezifikation sind jetzt alle Unterobjekte mittels sicherer Inklusion importiert, deren Identifikatoren im Schlüssel auftauchen. Der listenwertige Schlüssel wird dabei konzeptionell in eine Menge umgewandelt, da sichere Inklusion für Instanzen*mengen* definiert ist. Die Verwaltung der Listeneigenschaften (Ordnung, Duplikate) bei Zugriffen auf Unterobjekte muß zusätzlich mit expliziten Attributen verwaltet werden.

Als letzter Schritt ist die Interpretation der generierten Ereignisse, Attribute und Unterobjekte durch die zugehörigen Konstrukte der Klasse STATIC_TEAM festzulegen, auf deren Angabe an dieser Stelle verzichtet wird. □

Dieses Beispiel soll nur das *Grundprinzip* verdeutlichen, mit dem dynamische Aggregation mit parametrisierten Komponentenkonstruktoren semantisch auf statische Objektinklusion zurückgeführt werden kann. Die hier weggelassene Interpretation der abgeleiteten Konzepte kann zum Teil mit den bereits vorgestellten Sprachmitteln Ereignisaufruf und abgeleiteten Attributen erfolgen. Als mächtigeres Konzept hierfür bietet sich die *formale Implementierung von Objekten über Basisobjekten* an, die in [ES90, SE91] diskutiert wird.

Es soll hier noch einmal betont werden, daß es wesentliche Unterschiede gibt zwischen Komponenten und zeigerwertigen Attributen. Zeigerwertige Attribute erlauben nur den Zugriff auf die *Werte* von Objektidentifikatoren, sie unterstützen also *nicht* den Zugriff auf die zugehörigen Instanzen. Insbesondere ist die Existenz eines Indentifikatorwerts *unabhängig von der Existenz der Instanz*, d.h. es gibt im Gegensatz zu einigen implementierten objektorientierten Ansätzen konzeptionell keine implizite Änderungsfortpflanzung (engl. *update propagation*) für Zeiger beim Löschen von Objekten !

Die benötigte Auswahl der konkreten parametrisierten Komponentenkonstruktoren und der jeweils generierten neuen Signaturbestandteile hängt natürlich stark vom

Anwendungsgebiet ab. Eine Spezifikationssprache sollte also neben einigen Standardkonstruktoren die Möglichkeit der *Erweiterung um neue Konstruktoren* prinzipiell vorsehen.

Als letzte Bemerkung zu nicht-disjunkten komplexen Objektstrukturen sei erwähnt, daß *rekursive* Strukturen — etwa die bekannte Teilehierarchie (*part - subpart - Beziehung*) — ohne syntaktische oder semantische Probleme modelliert werden können.

4.6 Heterogene Klassen und Generalisierung *

Im Abschnitt über Semantik von Objektklassen und Klassentypen wurden *heterogene Klassen und Typen* eingeführt. Heterogene Klassen und Typen sind mit Konzepten wie variante Strukturen oder Vereinigungsdatentypen und zugehörigen Variablen verwandt, die aus moderneren Programmiersprachen bekannt sind. Die dort verwendeten Beschreibungs- und Zugriffskonzepte können zum Teil auch auf Objektspezifikationen mit heterogenen Klassen übertragen werden.

Während in den Sprachvorschlägen von OBL89 [CSS89] und **Oblog$^+$** die explizite Konstruktion heterogener Klassen vorgesehen ist, ist in der aktuellen Version der Sprache TROLL nur die Konstruktion mittels der *Generalisierungsabstraktion* vorgesehen. In diesem Abschnitt werden beide Konzepte diskutiert und anhand von Beispielen vorgestellt.

4.6.1 Explizite Spezifikation heterogener Objektklassen *

Die wesentliche Idee heterogener Klassen und Klassentypen wurde bereits in Abschnitt 3.6 besprochen. Im Unterschied zu homogenen Klassen, können in *heterogenen Objektklassen bzw. Klassentypen* unterschiedlichen Objektidentifikatoren verschiedene Objektmodelle zugeordnet werden. Wie bei homogenen Klassen und Typen, erfolgt die Deklaration eines Klassentyps in der Regel integriert mit der einer Klasse.

Bei der Deklaration einer heterogener Klasse müssen somit mehrere Einzelpunkte spezifiziert werden :

- Der Namensraum der Klasse (d.h., des Klassentyps).
- Die verschiedenen Objektbeschreibungen.
- Eine *eindeutige Zuordnung* von Objektbeschreibungen zu Objektidentifikatoren.

Als Schnittstelle nach außen haben wir in der Regel die *Vereinigung* der Einzelsignaturen der verschiedenen Objektbeschreibungen, wobei ein syntaktisches Konzept (Attribut, Ereignis, Unterobjekt) bei einer Klasseninstanz nur ansprechbar bzw. anwendbar ist, wenn die Instanz das passende Template hat.

Als Erweiterung dieser Basisidee führen wir eine explizite *gemeinsame homogene Teilsignatur* inklusive die Spezifikation deren Semantik ein, die allen Objektbeschreibungen gemeinsam ist. Diese wird gesondert deklariert, um Redundanz in der Beschreibung möglichst zu vermeiden. Eine wesentliche Aufgabe dieser gemeinsamen Konzepte ist die Bereitstellung einer einheitlichen Schnittstelle für die heterogenen Objekte. Dies kann durch geeignete abgeleitete Attribute und gemeinsame Ereignisse erfolgen.

Definition 4.6.1 Die Deklaration einer *heterogenen Objektklasse* kann syntaktisch nach folgenden Muster erfolgen :

```
heterogeneous object class KlassenName
   Namensraumdefinition und gemeinsame konstante Attribute
   template
   case
      Bedingung_1 : Templatedeklaration_1;
      Bedingung_2 : Templatedeklaration_2;
      ...
   otherwise Templatedeklaration_o;
   end case;
   common template
      Templatedeklaration_c;
end object class KlassenName;
```

Der **otherwise**-Fall kann weggelassen werden, wenn gewährleistet ist, daß für jeden Objektidentifikator mindestens eine `Bedingung_i` wahr ist. Für jeden Identifikator darf höchstens eine `Bedingung_i` erfüllt sein. `Bedingung_i` ist jeweils eine Formel über den konstanten Attributen und den Schlüsselattributen. `Templatedeklaration`en sind entweder explizit deklarierte Templates oder Verweise auf bereits spezifizierte Templates.

In der gemeinsamen Objektbeschreibung `Templatedeklaration_c` kann durch die Verwendung von **case**-Anweisungen oder äquivalenten Sprachmitteln auf die syntaktischen Konzepte der heterogenen Templatedeklarationen zugegriffen werden, um z.B. abgeleitete Attribute zu definieren. □

Die Semantik einer heterogenen Klassendefinition kann direkt durch die in Abschnitt 3.6.1 vorgestellten semantischen Modelle ausgedrückt werden.

Beispiel 4.6.2 Wir betrachten eine heterogene Klasse `Veröffentlichung`, deren Instanzen Bücher, Tagungsbände, Zeitschriftenartikel, etc. sein können.

```
heterogeneous object class Veröffentlichung
   data types ..
```

```
  identification
    Nummer: integer;
  constants
    Typ: enumeration(Buch,Tagungsband,...);
  template
  case
    if Typ = Buch then book.template;
    if Typ = Tagungsband then
      template
        attributes
          data types ..
          Herausgeber: list(|PERSON|);
        ...
      end template
    ...
  common template
    template
      data types ..
      attributes
        derived AutorenOderHerausgeber: list(|PERSON|);
        ...
      constraints
        derivation rules
          AutorenOderHerausgeber =
            case
              if Typ = Buch then Autoren,
              if Typ = Tagungsband then Herausgeber,
              ...
            end case;
      ...
    end template
end object class Veröffentlichung;
```

Die eindeutige Zuordnung zwischen Objektidentifikatoren und Templates wird anhand des konstanten Attributs `Typ` definiert.

Das in der Ableitungsregel benutzte **case**-Konstrukt ist im wesentlichen eine syntaktische Abkürzung für geschachtelte **if-then-else**-Anweisungen[5], die syntaktisch in Anlehnung an die **case**-Fallunterscheidung in Deklarationen von heterogenen Klassentypen gewählt wurden. □

Dieses Beispiel zeigt nur eine kleinen Ausschnitt aus der Modellierung einer he-

[5] Da keine der zugrundeliegenden Spezifikationssprachen derartige Konstruktionen anbietet, wurde eine einfache intuitive Syntax gewählt, die in konkreten Sprachvorschlägen sicher überarbeitet werden muß.

terogenen Klasse. Bereits hierbei wird deutlich wie komplex derartige Spezifikationen werden können. Aus diesem Grunde werden heterogene Klassen in der Regel eher als Resultat von *Generalisierungen* auftreten, wie sie im folgenden Abschnitt behandelt werden.

4.6.2 Generalisierung *

Die *Generalisierung* ist eine aus semantischen Datenmodellen wohlbekannte Konstruktion. Die grundlegende Idee der Generalisierung ist unterschiedliche Objektklassen konzeptionell zu einer gemeinsamen Klasse zu vereinigen (zu *generalisieren*). Im Gegensatz zur Mehrfachvererbung bei Rollen und Spezialisierungen basiert die Generalisierung auf einer *disjunkten* Vereinigung der Klassenpopulationen.

Eine Generalisierung besteht im vorgestellten Spezifikationsansatz somit

- aus der Definition eines heterogenen Klassentyps, dessen Namensraum aus der disjunkten Vereinigung der Namensräume der beteiligten Klasse besteht, und
- der Definition einer Vereinigungsklasse analog zur Rollenkonstruktion und Spezialisierung.

Auch hier haben wir also wieder einerseits eine syntaktische Kontruktion, nämlich die Festlegung eines heterogenen Klassentyps, und andererseits eine semantische Dimension, hier die Festlegung der Klassenausprägung als Vereinigung der Klassenausprägungen der beteiligten Basisklassen.

Analog zur expliziten Deklaration heterogener Klassen erlauben wir auch bei der Generalisierung die Definition einer gemeinsamen Teilbeschreibung. Semantisch ist diese Erweiterung aber auch durch eine Basisgeneralisierung ohne Signaturerweiterung und einer anschließenden Spezialisierung mit Signaturerweiterung ausdrückbar.

Definition 4.6.3 Syntaktisch wird die Deklaration einer *Generalisierungsklasse* wie folgt notiert.

object class `KlassenName`
  generalizing *Liste von disjunkten Klassen*
  template
    gemeinsames Template;
end object class `KlassenName;`

Klassen sind als *disjunkt* zu betrachten, wenn sie nicht in einem Rollen- bzw. Spezialisierungsverhältnis zur selben Basisklasse stehen. Die Angabe eines gemeinsamen Templates ist optional. □

Ein bekanntes Beispiel für Generalisierungen ist der Begriff der juristischen Person, unter dem Personen und Firmen rechtlich zusammengefaßt werden, wenn es um Begriffe wie Besitzverhältnisse etc. geht.

Beispiel 4.6.4 Die Klasse der juristischen Personen `LEGAL_PERSON` kann nach obigem Muster als Generalisierung der Klassen `PERSON` und `COMPANY` deklariert werden.

```
object class LEGAL_PERSON
   generalizing PERSON, COMPANY
   template
      data types set(|CAR|), ...;
      attributes
         OwnedCars: set(|CAR|);
         derived LegalName: string;
      ...
      contraints
         derivation rules
            LegalName =
               case
                  if SELF isa PERSON then Name,
                  if SELF isa COMPANY then CompanyName
               end case;
         ...
end object class LEGAL_PERSON;
```

Die atomare Formel '**SELF isa** `PERSON`' testet die Zugehörigkeit einer Instanz der Generalisierungsklasse zu einer Basisklasse. □

In der Sprache TROLL wird die Generalisiserungsdeklaration weiter eingeschränkt, indem statt Ableitungsbedingungen für die homogene Schnittstelle nur Umbenennung existierender Signaturbestandteile erlaubt ist [JSHS91].

Die gemeinsame Objektbeschreibung kann wie bei expliziten heterogenen Klassendeklarationen zur Festlegung einer einheitlichen gemeinsamen Zugriffsschnittstelle (hier Attribut `LegalName`) oder auch zu echten Erweiterungen (hier das Attribut `OwnedCars`) benutzt werden. Die Semantik der gemeinsamen Schnittstelle wird durch Ableitungsregeln, gemeinsame Ereignisse und Ereignisaufruf festgelegt. Die Semantik echter Signaturerweiterungen werden wie üblich im gemeinsamen Template explizit spezifiziert.

Definition 4.6.5 Die *Semantik* einer Generalisierungsklasse ist durch eine heterogene Objektklasse definiert, für die gilt :

- Der *Namensraum* der heterogenen Klasse ist durch die disjunkte Vereinigung der Namensräume der Basisklassen gegeben. Diese Vereinigung erfolgt durch die Instanziierung eines parametrisierten Datentypkonstruktors **union**.

- Für jede Basisklasse gibt es ein Template, das als einzige Spezifikationsbestandteile die *semantische Inklusion der zugehörigen Basisinstanz* sowie jeweils ein generiertes Geburtsereignis und Todesereignis enthält.

- Geburtsereignisse der Basisklassen rufen die korrespondierenden Geburtsereignisse der Generalisierungsklasse auf. Dies gilt ebenfalls für Todesereignisse.

- Die gemeinsame Objektbeschreibung wird behandelt wie bei expliziten heterogenen Klassen.

Die Bedeutung des **isa**-Prädikates kann auf ein Prädikat auf dem durch disjunkte Vereinigung erzeugten Namensraum definiert werden. □

4.7 Globale Beziehungen

In den meisten objektorientierten Programmiersprachen und Datenmodellen sind Objekte das einzige Modellierungskonzept und Beziehungen zwischen Objekten werden durch ein Zeigerkonzept dargestellt. Ein Zeigerkonzept ist vom Prinzip her eine gerichtete Beziehung — auch wenn in vielen objektorientierten Datenmodellen die inverse Beziehung vom System bereitgestellt werden kann. Auch die Komponentenbildung des hier vorgestellten Ansatzes ist eine gerichtete Beziehung zwischen Objekten und kann als Abstraktion eines Zeigerkonzepts aufgefaßt werden.

Im Bereich der konzeptionellen Modellierung hat sich der Nutzen ungerichteter Beziehungen als Modellierungskonzept allein durch den Erfolg des *Entity-Relationship-Modells* [Che76] in so unterschiedlichen Bereichen wie Datenstrukturentwurf oder Problemanalyse herausgestellt. Auch Ansätze der objektorientierten Modellierung erweitern den rein objektorientierten Ansatz um ungerichtete Beziehungen [RBP+90].

Im Bereich der objektorientierten Spezifikation bieten sich allgemeine Beziehungen als Konzept zur strukturierten Spezifikation globaler Zusammenhänge an. Insbesondere die *globale Kommunikation* zwischen Objekten und *globale Integritätsbedingungen* können auf diese Weise auf natürliche und strukturierte Art deklariert werden. Globale Beziehungen und ihre Integration in die Sprache TROLL werden in [JHS92] ausführlich diskutiert.

4.7.1 Globale Beziehungen als Objektaggregationen

Allgemeine globale Beziehungen können als Paare oder n-stellige Tupel von Objekten aufgefaßt werden. Eine Beziehungsinstanz wird identifiziert durch die an ihr beteiligten Objektinstanzen, und ihre Eigenschaften werden durch die Eigenschaften der beteiligten Objekte festgelegt. Diese Eigenschaften legen es nahe, allgemeine Beziehungen semantisch als *Spezialfälle von Objektaggregation mittels sicherer Objektinklusion* aufzufassen.

Definition 4.7.1 Eine *Beziehung* zwischen Objektklassen ist eine abkürzende Schreibweise für eine Objektklasse,

- deren Instanzen durch Tupel über den beteiligten Objektklassennamensräumen identifiziert werden, und
- deren Template die zugehörigen Objektinstanzen mittels sicherer Inklusion importiert.

Instanzen der Beziehung werden jeweils zusammen mit den Instanzen der beteiligten Klassen erzeugt.

Analog zu Erweiterungen des Entity-Relationship-Models [HNSE87, EGH+90, EGH+92] kann diese Klasse eine Signaturerweiterung beinhalten (Stichwort *Beziehungsattribute*). In der Regel beinhaltet das Template allerdings nur diejenigen globalen Eigenschaften, die durch die Beziehung ausgedrückt werden sollen, also etwa Integritätsbedingungen und globale Kommunikation. □

Diese Definition legt nahe, auf Beziehungen als Modellierungskonzept zugunsten weniger Spezifikationsprimitive ganz zu verzichten und sie explizit durch Objektspezifikationen auszudrücken. Viele objektorientierte Ansätze verzichten darum auf ein zusätzliches Beziehungskonzept. Von den vorgestellten Sprachen zur Objektspezifikation besitzt nur die Sprache TROLL ein explizites Sprachkonzept für Beziehungen.

Für die Aufnahme des Beziehungskonzepts in eine objektorientierte Spezifikationssprache sprechen folgende Überlegungen :

- Das Beziehungskonzept verwirklicht eine ungeordnete Assoziation zwischen Objekten verschiedener Klassen. Diese müßte andernfalls durch gerichtete Beziehungen (Zeiger, Objekthierarchien) modelliert werden, wie es z.B. in objektorientierten Datenmodellen üblich ist. Für den konzeptionellen Entwurf sollten aber auch Modellierungskonzepte bereit gestellt werden, die von diesen 'implementierungsnahen' Konzepten abstrahieren können falls die Beziehung konzeptionell ungerichtet ist.
- Globale Eigenschaften wie Integritätsbedingungen oder globale Kommunikation können mit expliziten Beziehungen direkt modelliert werden, ohne daß künstliche zusätzliche Objektklassen explizit spezifiziert werden müssen.
- Die Aufnahme des Beziehungskonzepts ermöglicht die direkte Integration etablierter Modellierungsmethoden, z.B. den ER-Ansatz, in die objektorientierte Spezifikation.

Der folgende Abschnitt präsentiert eine Integration von Sprachkonzepten für Beziehungen wie sie etwa in der Sprache TROLL realisiert wurden. Um die Unterscheidung zwischen komplexen Objekten und Beziehungen auf der methodischen Ebene klar zu trennen, sind in diesen Sprachen keine Signaturerweiterungen (also Beziehungsereignisse und -attribute) in Beziehungsdeklarationen vorgesehen — Beziehungen mit Attributen und Ereignissen sollen auch konzeptionell explizit als Objektklassen spezifiziert werden.

4.7.2 Beziehungsdefinition für globale Kommunikation und Integritätsbedingungen

Globale Beziehungen werden in Objektspezifikationen insbesondere dafür eingesetzt, globale Eigenschaften wie Objektkommunikation und klassenübergreifende Integritätsbedingungen direkt modellieren zu können, ohne explizit Aggregationsobjekte definieren zu müssen. Die bisherigen Sprachkonstrukte erlauben die Angabe von Kommunikations- und Integritätsbedingungen nur *innerhalb* von Objektbeschreibungen, die allerdings ihrerseits durch semantische Inklusion mehrere andere Objekte als Unterobjekte importieren können. Globale Bedingungen lassen sich also bis jetzt nur dadurch modellieren, daß ein Objekt die betreffenden Klasseninstanzen als Unterobjekte importiert und Einschränkungen für deren Ausprägungen spezifiziert.

Die Definition einer *expliziten Beziehung* ist eine abkürzende Schreibweise für die Definition von Aggregationsobjekten zur Spezifikation globaler Eigenschaften.

Definition 4.7.2 Eine allgemeine *Beziehung* zwischen Objektklassen wird syntaktisch notiert als

relationship `Beziehungsname`
  between `Klasse_1 [Variable_1], ..., Klasse_n [Variable_n]`
    Templatebestandteile ohne *Signaturerweiterung*;
end relationship `Beziehungsname;`

Ähnlich der Konvention etwa in der SQL-Sprache können optionale Klassenvariablen definiert werden, um die Lesbarkeit zu erhöhen oder wenn die Beziehung zwischen Instanzen ein und derselben Klasse definiert ist.

Auch Beziehungen zwischen singulären Objekten und Klassen sind möglich — einzelstehende Objekte werden dazu wie einelementige Klassen behandelt. □

Wie bereits erwähnt, können Beziehungsdefinitionen als abkürzende Schreibweise für die Deklaration einer Klasse von Aggregationsobjekten aufgefaßt werden. Die Semantik einer Beziehungsdefinition wird folgerichtig durch die Definition einer äquivalenten Objektklasse festgelegt.

Definition 4.7.3 Die *Semantik einer Beziehungsdefinition* gemäß Definition 4.7.2 ist definiert durch die folgende äquivalente Objektklassendefinition :

object class `Beziehungsname`
  data types `|Klasse_1|, ..., |Klasse_n|;`
  identification
    `Klasse_1_Id: |Klasse_1|;`
    `...`
    `Klasse_n_Id: |Klasse_n|;`

template
  including object `Klasse_1_Id` **in** `Klasse_1` **as** `Variable_1`;
  ...
  including object `Klasse_n_Id` **in** `Klasse_n` **as** `Variable_n`;
  Templatebestandteile unverändert übernommen;
  ...
end object class `Beziehungsname`; ;

Zusätzlich muß durch Ereignisaufruf und ein generiertes Geburtsereignis garantiert werden, daß das Existenzintervall eines Beziehungsobjekts dem Intervall entspricht, in dem alle beteiligten Objekte existieren. □

Ein wichtiger Spezialfall globaler Beziehungen sind *Kommunikationsbeziehungen* zwischen Objekten. In Abschnitt 4.2 wurden die syntaktischen Primitive für die Deklaration symmetrischer und asymmetrischer Kommunikation vorgestellt, wie sie bisher innerhalb komplexer Objekte verwendet werden konnten.

Definition 4.7.4 Eine *Kommunikationsbeziehung* ist eine binäre Beziehung, in der nur die Deklaration gemeinsamer Ereignisse und von Ereignisaufruf erfolgt. □

In der Sprache **Oblog**$^+$ werden globale Kommunikationsbeziehungen eingeleitet mit dem Schlüsselwort **global interaction** gesondert deklariert [SJ91, JSS91]. In der Sprache TROLL wird die hier vorgestellte Notation verwendet [JSH91, JHSS91, JHS92].

Beispiel 4.7.5 Als Beispiel betrachten wir eine Ausschnitt aus den Kommunikationsbeziehungen in dem bereits zum Teil modellierten Bibliotheksbeispiel. Modelliert wird die Kommunikation zwischen `COPY`-Objekten (Beispiel 3.2.4 auf Seite 51) und `USER`-Objekten (Beispiel 4.4.9 auf Seite 113).

```
relationship USER_COPY
  between USER, COPY;
  variables U: |USER|, C: |COPY|, d: date, n: nat;
  sharing
    USER(U).borrow(C,d,n) == COPY(C).CheckOut(U,d,n);
  calling
    USER.return(C) >> COPY(C).Return;
end relationship USER_COPY;
```

Diese Kommunikation modelliert das Ausleihen und Zurückgeben von Buchexemplaren. Während das Ausleihen durch eine gemeinsames Ereignis ausgedrückt werden kann, muß die Rückgabe durch Ereignisaufruf modelliert werden, um nicht eine Synchronisation zwischen allen `USER`-Objekten über das `Return`-Ereignis zu erzwingen.

Die Notation `USER`(U) wird benutzt um der Variable U den Objektidentifikator des aktuell betrachteten `USER` zuzuweisen. □

Ein weiterer Spezialfall ist die Deklaration *globaler Integritätsbedingungen* mittels globaler Beziehungen. Viele objektorientierte Absätze verbieten globale Integritätsbedingungen mit dem Argument, daß diese Bedingungen und ihre Überwachung lokal durch Methoden erfolgen muß. Diese Argument trifft für objektorientierte Programmiersprachen und Datenbanken sicherlich zu, aber im konzeptionellen Entwurf sprechen starke Argumente für globale Integritätsbedingungen :

- Der konzeptionelle Entwurf startet mit der Analyse der (informalen) Anforderungen, die im wesentlichen globale Aussagen über das zu entwerfende System enthalten. Insbesondere in frühen Entwurfsschritten des konzeptionellen Entwurfs lassen sich derartige Aussagen oft direkt in globale Integritätsbedingungen überführen.

- Der konzeptionelle Entwurf soll vom zu implementierenden System abstrahieren. Die Entscheidung, wie und von welcher Methode eine Integritätsbedingung erzwungen wird, ist aber eine Implementierungsentscheidung.

Obwohl in gewissen Sinne alle Spezifikationsbestandteile zur Deklaration einer Art 'globaler Bedingung' benutzt werden könnten, beschränken wir uns bei globalen Integritätsbedingungen auf statische und temporale Integritätsbedingungen über den Attributwerten.

Beispiel 4.7.6 Als Beispiel betrachten wir eine globale Integritätsbedingung zwischen COPY-Objekten (Beispiel 3.2.4 auf Seite 51) und USER-Objekten (Beispiel 4.4.9 auf Seite 113).

```
relationship USER_COPY_IC
  between USER, COPY;
  variables U: |USER|, C: |COPY|;
  constraints
    in(C,USER(U).borrowed) implies
          (COPY(C).OnLoan and U = last(COPY(C).Borrowers));
end relationship USER_COPY_IC;
```

Die Spezifikation der Ereignisse der Objektklassen USER und COPY erzwingt bereits lokal die Einhaltung dieser Integritätsbedingung. □

Denkbar ist auch die Deklaration globaler Ablaufprozesse durch entsprechende globale Beziehungen. Aus methodischen Gründen sollten derartige Abläufe allerding explizit in Objekten eingekapselt werden, da sie zusätzlich lokale Hilfsattribute und Synchronisationsereignisse haben können und auch eine zeitliche Existenz mit Geburts- und Todeszeitpunkt besitzen. Mit anderen Worten, aus methodischen Gründen sollten globale Beziehungen keinen eigenen internen Zustand in Form eines Gedächtnisses haben.

4.8 Aufbau von Objektgesellschaften

In den bisherigen Abschnitten haben wir uns mit der Spezifikation von Objekten und Objektklassen sowie verschiedener Arten von Beziehungen zwischen Objekten beschäftigt. Das eigentliche Ziel des konzeptionellen Entwurfs ist allerdings die Beschreibung ganzer *Systeme*, die aus Objekten als Basiseinheiten aufgebaut sind. Allgemein bezeichnen wir im folgenden ein System aufgebaut aus kommunizierenden Objekten als *Objektgesellschaft*.

Motiviert ist die Benutzung des Begriffs *Gesellschaft* in diesem Zusammenhang durch folgende Beobachtung : Das Verhalten von Objekten wird lokal beschrieben und erlaubt einzelnen Objekten in der Regel ein breites Spektrum an möglichen Lebensläufen. Eingebunden in einem System vieler Objekte wird dieses Verhalten eingeschränkt durch die Kommunikation und die Beziehungen zu anderen Objekten — genau so, wie Menschen in einer Gesellschaft sich an Regeln des Zusammenlebens halten müssen und somit einen Teil ihrer Autonomie aufgeben.

Eine Objektgesellschaft entspricht einem vollständigen und abgeschlossenen System kommunizierender Objekte. Die Spezifikation einer Objektgesellschaft ist somit eine Beschreibung aller Aspekte eines Objektsystems, die auf Vollständigkeit und Konsistenz überprüft werden kann. Neben den vorgestellten Spezifikationen von Objekten und Objektklassen beinhaltet sie somit auch die formale Spezifikation von abstrakten Datentypen als Wertebereiche für Attribute und Parameter.

In einer ersten Annäherung kann eine Objektgesellschaft wie folgt beschrieben werden :

object society `SocietyName`
  Import von Bibliotheken;
  Datentypspezifikationen;
  Template- und Klassentypspezifikationen;
  Objekt- und Klassendeklarationen;
  Beziehungsdeklarationen;
end object society `SocietyName`.

Die einzelnen Bestandteile einer derartigen Spezifikation einer Objektgesellschaft werden wie folgt aufgeführt :

- Der *Name* der Objektgesellschaft `SocietyName` kann beliebig gewählt werden, sofern kein Konflikt zu anderen Objektgesellschaften auftritt.

- Die Beschreibung der Objektgesellschaft beginnt mit der Angabe des *Imports von Bibliotheken*. Bibliotheken können Bibliotheken von Datentypspezifikationen (hier nicht behandelt) oder Bibliotheken von — eventuell parametrisierten — Template- und Klassentypspezifikationen sein.

- *Datentypspezifikationen* dienen der formalen Beschreibung von Datentypen als Wertebereiche für Attribute und Parameter. Die konkrete Spezifikationssprache ist unabhängig vom Formalismus der Objektbeschreibung und kann im Prinzip frei gewählt werden [EM85, EGL89].

- Explizite *Template- und Klassentypspezifikationen* legen Typbeschreibungen für Objekte und Klassen fest, die bei konkreten Objekt bzw. Klassendeklarationen wiederverwendet werden können.

- *Objekt- und Klassendeklarationen* definieren die eigentlichen Objektsysteme und deren möglichen Objektpopulationen.

- *Beziehungsdeklarationen* (Abschnitt 4.7) etablieren die Kommunikationskanäle zwischen autonomen Objekten und ermöglichen die Angabe globaler Integritätsbedingungen.

Entgegen der obigen linearen Auflistung der Spezifikationsbestandteile erlauben die Sprachen TROLL und **Oblog$^+$** eine beliebige Reihenfolge der drei letzteren Punkte, also der Typbeschreibungen für Objekte und Klassen, der Objekt- und Klassendeklarationen sowie der Beziehungsdeklarationen.

Literaturhinweise

Die Darstellung der verschiedenen Objektbeziehungen ist an deren Realisierung in der Sprache TROLL orientiert [JSHS91, JHSS91].

Ereignisaufruf und identifizierte Ereignisse haben ihre Wurzeln in synchronen Kommunikationsmodellen für abstrakte Prozeßmodelle [Hen88, Mil80, Hoa85].

Das Konzept der sicheren Objektinklusion basiert auf dem Konzept eines *Objektmorphismus* als ausschließliches Konzept, Objekte miteinander in Beziehung zu setzen. Objektmorphismen für verschiedene Objektmodelle werden in [SSE87, ESS90, EGS90, SEC90, ES91] diskutiert. Die Darstellung in dieser Arbeit ist an [EGS90] angelehnt. Der Bezug verschiedener Abstraktionsmechanismus zu Objektmorphismen ist in [ES91, SJE92, ESS92] ausführlich dargestellt.

Spezialisierungshierarchieen so wie sie hier vorgestellt wurden haben ihren Ursprung in verschiedenen Wissensrepräsentationsformalismen und semantischen Datenmodellen [MBW80, HK87, UD86, PM88]. Die besondere Rolle temporärer Objektrollen bzw. -phasen wird von R. Wieringa in [Wie90] diskutiert.

Der Begriff der komplexen Objekte entstammt dem Bereich der Datenbanksysteme für Nicht-Standard-Anwendungen [LP83, LNE89, Mit88, Heu92]. Komplexe Objekte in TROLL werden in [HJS92] diskutiert.

Heterogene Objektklassen haben ihre Wurzeln in der Modellierungen von Generalisierungsklassen, bei der neue Klassen Objekte unterschiedlicher Struktur zusammenfassen [EGH+92]. Verschiedene Vorschläge für die Deklaration heterogener Objektklassen in Objektspezifikationen können in [CSS89, JSS91, JSHS91] gefunden werden.

Ungerichtete Beziehungen in der konzeptionellen Modellierung haben ihre Bedeutung auch durch den Erfolg des ER-Modells zur implementierungsunabhängigen Beschreibung von Datenbankstrukturen [Che76] gewonnen. [RBP+90] schlägt die Integration von ungerichteten Beziehungen in den objektorientierten Entwurf aus methodischen Gründen vor. Globale Beziehungen in Troll und ihre Rolle bei der objektorientierten Spezifikation werden in [JHS92] diskutiert.

Literatur

[ABD$^+$89] Atkinson, M.; Bancilhon, F.; DeWitt, D.; Dittrich, K. R.; Maier, D.; Zdonik, S. B.: The Object-Oriented Database System Manifesto. In: Kim, W.; Nicolas, J.-M.; Nishio, S. (Hrsg.): *Proc. Int. Conf. on Deductive and Object-Oriented Database Systems*, Kyoto, Japan, Dezember 1989. S. 40–57.

[AH87] Abiteboul, S.; Hull, R.: IFO — A Formal Semantic Database Model. *ACM Transactions on Database Systems*, Band 12, Nr. 4, 1987, S. 525–565.

[Ara91a] Arapis, C.: Temporal Specifications of Object Behavior. In: Thalheim, B.; Demetrovics, J.; Gerhardt, H.-D. (Hrsg.): *Proc. 3rd Symp. Mathematical Fundamentals of Database and Knowledge Base Systems MFDBD '91*, Rostock, 1991. LNCS 495, Springer-Verlag, S. 308–324.

[Ara91b] Arapis, C.: Temporal Specifications of Object Interactions. In: Göers, J.; Heuer, A.; Saake, G. (Hrsg.): *Proc. 3rd Workshop on Foundations of Models and Languages for Data and Objects*, Aigen, Österreich, 1991. Informatik-Bericht 91/3, TU Clausthal, S. 15–36.

[BB92] Breutmann, B.; Burkhardt, R.: *Objektorientierte Systeme. Grundlagen – Werkzeuge – Einsatz.* Carl Hanser Verlag, München, 1992.

[Bee90] Beeri, C.: A Formal Approach to Object Oriented Databases. *Data & Knowledge Engineering*, Band 5, Nr. 4, 1990, S. 353–382.

[BI86] Bibliographischen Instituts, Fachredaktionen des (Hrsg.): *Schülerduden. Die Informatik.* Bibliographisches Institut, Mannheim, 1986. Bearbeitet von Volker Claus und Andreas Schwill.

[BL91] Brass, S.; Lipeck, U.: Semantics of Inheritance in Logical Object Specifications. In: Delobel, C.; Kifer, M.; Masunaga, Y. (Hrsg.): *Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases DOOD'91.* Springer-Verlag, 1991, S. 411–430.

[BM91] Beeri, C.; Milo, T.: A Model for Active Object Oriented Database. In: Lohman, G. M.; Sernadas, A.; Camps, R. (Hrsg.): *VLDB'91.* Morgan Kaufman, 1991, S. 337–349.

[Che76] Chen, P. P.: The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, Band 1, Nr. 1, 1976, S. 9–36.

[CHJ86] Cohen, B.; Harwood, W.T.; Jackson, M.I.: *The Specification of Complex Systems.* Addison-Wesley, Wokingham, England, 1986.

[Cho92a] Chomicki, J.: History-less Checking of Dynamic Integrity Constraints. In: *Proc. 8th IEEE Conf. on Data Engineering, Phoenix, Arizona*, 1992.

[Cho92b] Chomicki, J.: Real-Time Integrity Constraints. In: *Proc. 11th ACM SIGACT-SIGMOD-SIGART Symp. on Database Systems, San Diego, California*, 1992.

[CS87] Carmo, J.; Sernadas, A.: A Temporal Logic Framework for a Layered Approach to Systems Specification and Verification. In: *Proc. IFIP WG 8.1 Conf. on Temporal Aspects in Information Systems*, 1987.

[CSS89] Costa, J. F.; Sernadas, A.; Sernadas, C.: OBL-89. User's Manual. Version 2.3. Technical Report, Instituto Superior Técnico, Instituto de Engenharia de Sistemas e Computadores, Lisbon, 1989.

[Day88] Dayal, U.: Active Database Management Systems. In: *Proc. Conf. on Data and Knowledge Bases*, Jerusalem, 1988. S. 150–169.

[Den91] Denert, E.: *Software-Engineering. Methodische Projektabwicklung.* Springer-Verlag, Berlin, 1991.

[EDG88] Ehrich, H.-D.; Drosten, K.; Gogolla, M.: Towards an Algebraic Semantics for Database Specification. In: Meersmann, R.A.; Sernadas, A. (Hrsg.): *Proc. 2nd IFIP WG 2.6 Working Conf. on Database Semantics "Data and Knowledge" (DS-2)*, Albufeira (Portugal), 1988. North-Holland, Amsterdam, S. 119–135.

[EGH+90] Engels, G.; Gogolla, M.; Hohenstein, U.; Hülsmann, K.; Löhr-Richter, P.; Saake, G.; Ehrich, H.-D.: Conceptual Modelling of Database Applications Using an Extended ER Model. Informatik-Bericht 90-05, Technische Universität Braunschweig, 1990.

[EGH+92] Engels, G.; Gogolla, M.; Hohenstein, U.; Hülsmann, K.; Löhr-Richter, P.; Saake, G.; Ehrich, H.-D.: Conceptual Modelling of Database Applications Using an Extended ER Model. *Data & Knowledge Engineering*, 1992. North-Holland, *To appear*.

[EGL89] Ehrich, H.-D.; Gogolla, M.; Lipeck, U.W.: *Algebraische Spezifikation abstrakter Datentypen.* Teubner, Stuttgart, 1989.

[EGS90] Ehrich, H.-D.; Goguen, J. A.; Sernadas, A.: A Categorial Theory of Objects as Observed Processes. In: deBakker, J.W.; deRoever, W.P.; Rozenberg, G. (Hrsg.): *Proc. REX/FOOL Workshop*, Noordwijkerhood (NL), 1990. LNCS 489, Springer, Berlin, S. 203–228.

[EKTW86] Eder, J.; Kappel, G.; Tjoa, A.M.; Wagner, R.R.: BIER : The Behaviour Integrated Entity Relationship Approach. In: Spaccapietra, S. (Hrsg.): *Proc. 5th Int. Conf. on Entity-Relationship Approach*, Dijon, 1986. S. 147–166.

[ELG84] Ehrich, H.-D.; Lipeck, U. W.; Gogolla, M.: Specification, Semantics, and Enforcement of Dynamic Database Constraints. In: *Proc. Int. Conf. on Very Large Databases VLDB '84*, Singapore, 1984. S. 301–308.

[EM85] Ehrig, H.; Mahr, B.: *Fundamentals of Algebraic Specification 1. Equations and Initial Semantics.* Springer-Verlag, Berlin, 1985.

[EM90] Ehrig, H.; Mahr, B.: *Fundamentals of Algebraic Specification 2. Module Specification and Constraints.* Springer-Verlag, Berlin, 1990.

[Eme90] Emerson, E. A.: Temporal and Modal Logic. In: Leeuwen, Jan van (Hrsg.): *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics.* Elsevier, Amsterdam, and The MIT Press, Cambridge, 1990, S. 995–1072.

[EN89] Elmasri, R.; Navathe, S.B.: *Fundamentals of Database Systems.* Benjamin / Cummings Publ., Redwood City, CA, 1989.

[ES90] Ehrich, H.-D.; Sernadas, A.: Algebraic Implementation of Objects over Objects. In: deBakker, J. W.; deRoever, W.-P.; Rozenberg, G. (Hrsg.): *Proc. REX Workshop "Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness".* LNCS 430, Springer, Berlin, 1990, S. 239–266.

[ES91] Ehrich, H.-D.; Sernadas, A.: Fundamental Object Concepts and Constructions. In: Saake, G.; Sernadas, A. (Hrsg.): *Information Systems –*

Correctness and Reusability. TU Braunschweig, Informatik Bericht 91-03, 1991, S. 1–24.

[ESS88] Ehrich, H.-D.; Sernadas, A.; Sernadas, C.: Abstract Object Types for Databases. In: Dittrich, K. R. (Hrsg.): *Advances in Object-Oriented Database Systems*, Bad Münster am Stein, 1988. LNCS 334, Springer, Berlin, 1988, S. 144–149.

[ESS89] Ehrich, H.-D.; Sernadas, A.; Sernadas, C.: Objects, Object Types, and Object Identification. In: Ehrig, H.; Herrlich, H.; Kreowski, H.-J.; Preuß, G. (Hrsg.): *Categorical Methods in Computer Science.* LNCS 393, Springer, Berlin, 1989, S. 142–156.

[ESS90] Ehrich, H.-D.; Sernadas, A.; Sernadas, C.: From Data Types to Object Types. *Journal on Information Processing and Cybernetics EIK*, Band 26, Nr. 1-2, 1990, S. 33–48.

[ESS92] Ehrich, H.-D.; Saake, G.; Sernadas, A.: Concepts of Object-Orientation. In: *Proc. of the 2nd Workshop of "Informationssysteme und Künstliche Intelligenz: Modellierung", Ulm (Germany).* Springer IFB 303, 1992, S. 1–19.

[EWH85] Elmasri, R. A.; Weeldreyer, J.; Hevner, A.: The Category Concept: An Extension to the Entity-Relationship Model. *Data & Knowledge Engineering*, Band 1, 1985, S. 75–116.

[FS88] Fiadeiro, J.; Sernadas, A.: Specification and Verification of Database Dynamics. *Acta Informatica*, Band 25, Nr. 6, 1988, S. 625–661.

[GH91] Gogolla, M.; Hohenstein, U.: Towards a Semantic View of an Extended Entity-Relationship Model. *ACM Transactions on Database Systems*, Band 16, 1991, S. 369–416.

[GJM91] Ghezzi, C.; Jazayeri, M.; Mandrioli, D: *Fundamentals of Software Engineering.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Hen88] Hennessy, M.: *Algebraic Theory of Processes.* MIT Press, Cambridge, Massachusetts, 1988.

[Heu92] Heuer, A.: *Objektorientierte Datenbanken. Konzepte, Modelle, Systeme.* Addison-Wesley Verlag (Deutschland) GmbH, Bonn, 1992. *To appear.*

[HG88] Hohenstein, U.; Gogolla, M.: A Calculus for an Extended Entity-Relationship Model Incorporating Arbitrary Data Operations and Aggregate Functions. In: Batini, C. (Hrsg.): *Proc. 7th Int. Conf. on the*

Entity-Relationship Approach, Rome, 1988. North-Holland, Amsterdam, 1988, S. 129–148.

[HJ92] Hartmann, T.; Jungclaus, R.: Abstract Description of Distributed Object Systems. In: Tokoro, M.; Nierstrasz, O.; Wegner, P. (Hrsg.): *Proc. ECOOP'91 Workshop on Object-Based Concurrent Computing. Geneva (CH), 1991.* Springer, LNCS 612, Berlin, 1992, S. 227–244.

[HJS92] Hartmann, T.; Jungclaus, R.; Saake, G.: Aggregation in a Behavior Oriented Object Model. In: Lehrmann Madsen, O. (Hrsg.): *Proc. European Conference on Object-Oriented Programming (ECOOP'92).* Springer, LNCS 615, Berlin, 1992, S. 57–77.

[HK87] Hull, R.; King, R.: Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, Band 19, Nr. 3, 1987, S. 201–260.

[HNSE87] Hohenstein, U.; Neugebauer, L.; Saake, G.; Ehrich, H.-D.: Three-Level Specification of Databases Using an Extended Entity-Relationship Model. In: Wagner, R.; Traunmüller, R.; Mayr, H. C. (Hrsg.): *Proc. GI-Fachtagung "Informationsermittlung und -analyse für den Entwurf von Informationssystemen"*, Linz, 1987. Informatik-Fachbericht 143, Springer, Berlin, 1987, S. 58–88.

[Hoa85] Hoare, C.A.R.: *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, 1985.

[Hoh90] Hohenstein, U.: *Ein Kalkül für ein erweitertes Entity-Relationship-Modell und seine Übersetzung in einen relationalen Kalkül.* Dissertation, Technische Universität Braunschweig, 1990.

[HS90] Hülsmann, K.; Saake, G.: Representation of the Historical Information Necessary for Temporal Integrity Monitoring. In: Bancilhon, F.; Thanos, C.; Tsichritzis, D. (Hrsg.): *Proc. Int. Conf. on Extending Database Technology EDBT'90*, Venice (Italy), 1990. LNCS 416, Springer, Berlin, 1990, S. 378–392.

[HS91] Hülsmann, K.; Saake, G.: Theoretical Foundations of Handling Large Substitution Sets in Temporal Integrity Monitoring. *Acta Informatica*, Band 28, Nr. Fasc 4, 1991, S. 365–407.

[JHS92] Jungclaus, R.; Hartmann, T.; Saake, G.: Relationships between Dynamic Objects. In: Kangassalo, H. (Hrsg.): *Proc. 2nd Eurpean-Japanese Seminar on Information Modelling and Knowledge Bases, Hotel Ellivuori (SF).* IOS Press, Amsterdam, 1992. *To appear.*

[JHSS91] Jungclaus, R.; Hartmann, T.; Saake, G.; Sernadas, C.: Introduction to TROLL – A Language for Object-Oriented Specification of Information Systems. In: Saake, G.; Sernadas, A. (Hrsg.): *Information Systems – Correctness and Reusability*. TU Braunschweig, Informatik Bericht 91-03, 1991, S. 97–128.

[Jon86] Jones, C. B.: *Systematic Software Development Using VDM*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[JSH91] Jungclaus, R.; Saake, G.; Hartmann, T.: Language Features for Object-Oriented Conceptual Modeling. In: Teory, T.J. (Hrsg.): *Proc. 10th Int. Conf. on the ER-approach*, San Mateo, 1991. S. 309–324.

[JSHS91] Jungclaus, R.; Saake, G.; Hartmann, T.; Sernadas, C.: Object-Oriented Specification of Information Systems: The TROLL Language. Informatik-Bericht 91-04, TU Braunschweig, 1991.

[JSS91] Jungclaus, R.; Saake, G.; Sernadas, C.: Formal Specification of Object Systems. In: Abramsky, S.; Maibaum, T. (Hrsg.): *Proc. TAPSOFT'91, Brighton*. Springer, Berlin, LNCS 494, 1991, S. 60–82.

[KA90] Khoshafian, S.; Abnous, R.: *Object Orientation. Concepts, Languages, Databases, User Interfaces*. John Wiley & Sons, Inc., New York, 1990.

[KC86] Khoshafian, S.N.; Copeland, G.P.: Object identity. In: *Proc. OOPS-LA Conference*, Portland, OR, 1986. ACM, New York, 1986, S. 406–416. (Special Issue of SIGPLAN Notices, Vol. 21, No. 11, November 1986).

[KL88] Karl, S.; Lockemann, P.C.: Design of Engineering Databases : a Case for More Varied Semantic Modelling Concepts. *Information Systems*, Band 13, Nr. 4, 1988, S. 335–358.

[KLMP84] Kim, W.; Lorie, R.; McNabb, D.; Plouffe, W.: A Transaction Mechanism for Engineering Design Databases. In: *Proc. 10th Int. Conf. on Very Large Databases*, Singapore, 1984. S. 355–362.

[Kre91] Kreowski, H.-J.: *Logische Grundlagen der Informatik*. R. Oldenbourg Verlag, München, 1991.

[Krö87] Kröger, F.: *Temporal Logic of Programs*. Springer-Verlag, Berlin, 1987.

[LEG85] Lipeck, U. W.; Ehrich, H.-D.; Gogolla, M.: Specifying Admissability of Dynamic Database Behaviour Using Temporal Logic. In: Sernadas,

A. et al. (Hrsg.): *Proc. IFIP Working Conf. on Theoretical and Formal Aspects of Information Systems*. North-Holland, Amsterdam, 1985, S. 145–157.

[LF89] Lipeck, U. W.; Feng, D.S.: Construction of Deterministic Transition Graphs from Dynamic Integrity Constraints. In: *Proc. 14th Int. Workshop on Graph-Theoretic Concepts in Computer Science (WG'88)*. LNCS 344, Springer, Berlin, 1989, S. 166–179.

[Lip89] Lipeck, U. W.: *Zur dynamischen Integrität von Datenbanken: Grundlagen der Spezifikation und Überwachung*. Informatik-Fachbericht 209. Springer, Berlin, 1989.

[Lip90] Lipeck, U.: Transformation of Dynamic Integrity Constraints into Transaction Specifications. *Theoretical Computer Science*, Band 76, 1990, S. 115–142.

[LNE89] Lohmann, F.; Neumann, K.; Ehrich, H.-D.: Entwurf eines Datenbank-Prototyps für geowissenschaftliche Anwendungen. In: Härder, T. (Hrsg.): *Proc. GI/SI-Fachtagung "Datenbanksysteme in Büro, Technik und Wissenschaft"*, Zürich (CH), 1989. Springer, Berlin, 1989, S. 43–57.

[LP83] Lorie, R.; Plouffe, W.: Complex Objects and their Use in Design Transactions. In: *Proc. ACM SIGMOD*, Database Week, San Jose, 1983. S. 115–121.

[LS87] Lipeck, U. W.; Saake, G.: Monitoring Dynamic Integrity Constraints Based on Temporal Logic. *Information Systems*, Band 12, 1987, S. 255–269.

[LS88] Lipeck, U. W.; Saake, G.: Entwurf von Systemverhalten durch Spezifikation und Transformation temporaler Anforderungen. In: Valk, R. (Hrsg.): *Proc. GI Jahrestagung*, Band 2, Berlin, 1988. Informatik-Fachbericht 188, Springer, S. 449–463.

[MBW80] Mylopoulos, J.; Bernstein, P. A.; Wong, H. K. T.: A Language Facility for Designing Interactive Database-Intensive Applications. *ACM Transactions on Database Systems*, Band 5, Nr. 2, 1980, S. 185–207.

[Mil80] Milner, R.: *A Calculus of Communicating Systems*. Springer-Verlag, Berlin, 1980.

[Mil89] Milner, R.: *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, 1989.

[Mit88] Mitschang, B.: *Ein Molekül-Atom-Datenmodell für Non-Standard-Anwendungen.* Informatik-Fachberichte 185, Springer, Berlin, 1988.

[MP81] Manna, Z.; Pnueli, A.: Verification of Concurrent Programs : The Temporal Framework. In: Boyer, R.S.; Moore, J.S. (Hrsg.): *The Correctness Problem in Computer Science*, London, 1981. Academic Press, S. 215–273.

[MP92] Manna, Z.; Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems. Vol. 1: Specification.* Springer-Verlag, New York, 1992.

[MW84] Manna, Z.; Wolper, P.: Synthesis of Communicating Processes from Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, Band 6, 1984, S. 68–93.

[Obe90] Oberweis, A.: Zeitstrukturen für Informationssysteme. Dissertation, Universität Mannheim, 1990.

[PM88] Peckham, J.; Maryanski, F.: Semantic Data Models. *ACM Computing Surveys*, Band 20, Nr. 3, 1988, S. 153–189.

[RBP+90] Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W.: *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[Rei85] Reisig, W.: *Petri Nets. An Introduction.* Springer, Berlin, 1985.

[RW91] Reuter, A.; Wächter, H.: The ConTract Model. *Data Engineering Bulletin*, Band 3, 1991, S. 0–0.

[Saa88] Saake, G.: *Spezifikation, Semantik und Überwachung von Objektlebensläufen in Datenbanken.* Dissertation, Technische Universität Braunschweig, 1988.

[Saa89] Saake, G.: On First Order Temporal Logics with Changing Domains for Information System Specification. Informatik-Bericht 89–01, Technische Universität Braunschweig, 1989.

[Saa91a] Saake, G: Conceptual Modeling of Database Applications. In: Karagiannis, D. (Hrsg.): *Proc. 1st IS/KI Workshop, Ulm (Germany), 1990.* Springer, Berlin, LNCS 474, 1991, S. 213–232.

[Saa91b] Saake, G.: Descriptive Specification of Database Object Behaviour. *Data & Knowledge Engineering*, Band 6, Nr. 1, 1991, S. 47–74. North-Holland.

[SC92] Sol, H. G.; Crosslin, R. L. (Hrsg.): *Dynamic Modelling of Information Systems, II.* North-Holland, Amsterdam, 1992.

[Sch89] Schöning, U.: *Logik für Informatiker.* Bibliographisches Institut, Mannheim, 1989.

[SE91] Sernadas, A.; Ehrich, H.-D.: What Is an Object, After All? In: Meersman, R.; Kent, W.; Khosla, S. (Hrsg.): *Object-Oriented Databases: Analysis, Design and Construction (Proc. 4th IFIP WG 2.6 Working Conference DS-4, Windermere (UK))*, Amsterdam, 1991. North-Holland, S. 39–70.

[SEC90] Sernadas, A.; Ehrich, H.-D.; Costa, J.-F.: From Processes to Objects. *The INESC Journal of Research and Development 1:1*, 1990, S. 7–27.

[Ser80] Sernadas, A.: Temporal Aspects of Logical Procedure Definition. *Information Systems*, Band 5, 1980, S. 167–187.

[SFNC84] Schiel, U.; Furtado, A.L.; Neuhold, E.J.; Casanova, M.A.: Towards Multi-Level and Modular Conceptual Schema Specifications. *Information Systems*, Band 9, 1984, S. 43–57.

[SFSE89] Sernadas, A.; Fiadeiro, J.; Sernadas, C.; Ehrich, H.-D.: The Basic Building Blocks of Information Systems. In: Falkenberg, E.; Lindgreen, P. (Hrsg.): *Information System Concepts: An In-Depth Analysis*, Namur (B), 1989. North-Holland, Amsterdam, 1989, S. 225–246.

[Sie90] Siefkes, D.: *Formalisieren und Beweisen. Logik für Informatiker.* Vieweg, Braunschweig, 1990.

[SJ91] Saake, G.; Jungclaus, R.: Konzeptioneller Entwurf von Objektgesellschaften. In: Appelrath, H.-J. (Hrsg.): *Proc. Datenbanksysteme in Büro, Technik und Wissenschaft BTW'91.* Informatik-Fachberichte IFB 270, Springer, Berlin, 1991, S. 327–343.

[SJ92] Saake, G.; Jungclaus, R.: Specification of Database Applications in the TROLL-Language. In: Harper, D.; Norrie, M. (Hrsg.): *Proc. Int. Workshop Specification of Database Systems, Glasgow, July 1991.* Springer, London, 1992, S. 228–245.

[SJE92] Saake, G.; Jungclaus, R.; Ehrich, H.-D.: Object-Oriented Specification and Stepwise Refinement. In: de Meer, J.; Heymer, V.; Roth, R. (Hrsg.): *Proc. Open Distributed Processing, Berlin (D), 8.-11. Okt. 1991 (IFIP Transactions C: Communication Systems, Vol. 1).* North-Holland, 1992, S. 99–121.

[SL88] Saake, G.; Lipeck, U. W.: Foundations of Temporal Integrity Monitoring. In: Rolland, C. et al. (Hrsg.): *Proc. IFIP Working Conf. on Temporal Aspects in Information Systems*. North-Holland Publ. Comp., Amsterdam, 1988, S. 235–249.

[SL89] Saake, G.; Lipeck, U.W.: Using Finite-Linear Temporal Logic for Specifying Database Dynamics. In: Börger, E.; Kleine Büning, H.; Richter, M. M. (Hrsg.): *Proc. CSL'88 2nd Workshop Computer Science Logic*. Springer, Berlin, 1989, S. 288–300.

[Spi89] Spivey, J. M.: *The Z Notation: A Reference Manual.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[SSE87] Sernadas, A.; Sernadas, C.; Ehrich, H.-D.: Object-Oriented Specification of Databases: An Algebraic Approach. In: Hammerslay, P. (Hrsg.): *Proc. 13th Int. Conf. on Very Large Databases VLDB'87*, Brighton (GB), 1987. Morgan-Kaufmann, Palo Alto, 1987, S. 107–116.

[SvH91] Sol, H. G.; Hee, K. M. van (Hrsg.): *Dynamic Modelling of Information Systems*. North-Holland, Amsterdam, 1991.

[SW91] Schek, H.-J.; Weikum, G.: Erweiterbarkeit, Kooperation, Föderation von Datenbanksystemen. In: Appelrath, H.-J. (Hrsg.): *BTW'91*. Springer, Berlin, 1991, S. 38–71.

[TM87] Turski, W. M.; Maibaum, T. S. E.: *The Specification of Computer Programs*. Addison-Wesley, Reading, MA, 1987.

[Tsi85] Tsichritzis, D. (Hrsg.): *Office Automation : Concepts and Tools.* Springer-Verlag, Berlin, 1985.

[UD86] Urban, S. D.; Delcambre, L.: An Analysis of the Structural, Dynamic, and Temporal Aspects of Semantic Data Models. In: *Proc. Int. Conf. on Data Engineering*, Los Angeles, 1986. ACM, New York, 1986, S. 382–387.

[WdJ92] Wieringa, R. J.; Jonge, W. de: The Identification of Objects and Roles. Object Identifiers Revisited. Unpublished Draft, 1992.

[Weg90] Wegner, P.: Concepts and Paradigms of Object-Oriented Programming. *ACM SIGPLAN OOP Messenger*, Band 1, Nr. 1, 1990, S. 7–87.

[Wie90] Wieringa, R. J.: *Algebraic Foundations for Dynamic Conceptual Models.* Dissertation, Vrije Universiteit, Amsterdam, 1990.

[Win90] Wing, J. M.: A Specifier's Introduction to Formal Methods. *IEEE Computer*, Band 23, September 1990, S. 8–24.

[WL91] Wächter, H.; Lutz, R: Transaktionskonzept und Betriebsmodell des APRICOT-Systems. Vortragsfolien, 7. Workshop im DFG-Schwerpunkt “Objektbanken für Experten”, 1991.

[WR90] Wächter, H.; Reuter, A.: Grundkonzepte und Realisierungsstrategien des ConTract-Modells. *Informatik Forschung und Entwicklung*, Band 5, 1990, S. 202–212.

Index