Spezifikation von Objektsystemen *

Thorsten Hartmann
Ralf Jungclaus
Gunter Saake
Hans-Dieter Ehrich

Abt. Datenbanken, Techn. Universität Braunschweig
Postfach 3329, W-3300 Braunschweig
E-mail {hartmann|jungclau|saake|ehrich}@idb.cs.tu-bs.de

Zusammenfassung

Die konzeptionelle Modellierung des Weltausschnitts, der durch ein Informationssystem dargestellt werden soll, ist die entscheidende Phase beim Systementwurf, da das konzeptionelle Modell die Grundlage der Implementierung ist. Eine Sprache zur konzeptionellen Modellierung sollte daher auf einer soliden formalen Grundlage basieren, um einerseits die systematische Konstruktion einer Implementierung, andererseits eine frühe Animation zu ermöglichen.

In diesem Artikel wird die Sprache TROLL vorgestellt, die auf einem prozeßorientierten Objektmodell basiert. TROLL ermöglicht die deklarative Beschreibung der statischen sowie der dynamischen Aspekte eines zu beschreibenden Weltausschnittes mit Hilfe von Objekten.

Da deklarative Beschreibungsformalismen in der Regel nicht direkt ausführbar sind, werden Ideen zur Transformation von deklarativen in operationale Sprachkonstrukte eines Sprachkerns diskutiert. Für die Animation von Objektsystemen wird ein einfaches Ausführungsmodell zur Kommunikation von Objekten dargestellt und die Systemarchitektur eines verteilten Animationssystems erläutert.

1 Einleitung

Das Projekt „Formale Spezifikation und korrekte Implementierung von objektorientierten Informationssystemen“ ist ein Teilprojekt im Schwerpunktprogramm „Objektbanken für Experten“. Es befaßt sich mit dem auf formalen Grundlagen basierenden Entwurf von Informationssystemen.

Informationssysteme implementieren relevante Aspekte eines Weltausschnitts und sind immer in eine Umgebung eingebettet. In den ersten Phasen des Entwurfs ist das System also als geschlossenes System mit seiner Umgebung zu spezifizieren. Die relevanten Aspekte eines Weltausschnitts sind sowohl statischer als auch dynamischer Natur [Gri82]. Der Weltausschnitt kann als *System interagierender Komponenten* aufgefaßt werden.

Damit werden eine Reihe von Charakteristika impliziert:

- Die Komponenten eines Systems haben einen *lokalen Zustand*, d.h. Zustandsinformationen sind den Komponenten und nicht dem Gesamtsytem zugeordnet.

- Die Komponenten zeigen eine *zeitliche Entwicklung* ihres Zustands. Unabhängige Komponenten entwickeln sich auch unabhängig voneinander.

*In: Bayer, R.; Härder, T.; Lockemann, P.C.: *Objektbanken für Experten*, Reihe Informatik aktuell, Springer-Verlag, 1992

- Es gibt *agierende* und *reagierende* Komponenten. Agierende Komponenten lösen Zustandsübergänge aus, in reagierenden Komponenten werden Zustandsübergänge durch Interaktionen mit der Umgebung der Komponente ausgelöst.

- Die Komponenten eines Systems sind (wie das System selbst) u.U. strukturiert (z.B. aus Teilen zusammengesetzt).

Die Phase des *konzeptionellen Entwurfs* hat die Erstellung einer Spezifikation eines abstrakten Modells des Informationssystems zum Ziel. Diese Spezifikation wird oft auch *konzeptionelles Modell* genannt. Das konzeptionelle Modell sollte soweit als möglich *formal* sein, da es die Grundlage der Systementwicklung und somit einen Vertrag zwischen Anwendern und Systementwicklern darstellt. Um dieser Forderung nachzukommen, muß das konzeptionelle Modell festlegen, *was* realisiert werden soll, nicht jedoch *wie* dieses realisiert werden soll.

Die bisher zur konzeptionellen Modellierung eingesetzten Formalismen wie das ER-Modell und Erweiterungen [Che76, EN89, GH91], semantische Datenmodelle [HK87, PM88], aber auch informale Ansätze wie Datenflußdiagramme oder JSD [Jac83] sind nicht ausreichend, um alle Anforderungen an ein konzeptionelles Modell zu erfüllen. Semantischen Datenmodellen fehlen generell Konstrukte zur Beschreibung von Operationen und der zeitlichen Entwicklung von Objekten oder Entitäten. Datenflußdiagramme besitzen keine formale Semantik und passen in der Regel nicht zur einer Darstellung der Struktur eines Systems in einem anderen Formalismus. Auch JSD fehlt eine formale Semantik.

Die Betrachtungsweise eines Weltausschnitts als ein System interagierender Komponenten ist prinzipiell der *objektorientierten* Sichtweise sehr nahe. Beim objektorientierten Entwurf von Softwaresystemen [Boo90, RBP+90, CF92] werden die Systemkomponenten durch Objekte repräsentiert, die sowohl statische Struktur als auch dynamisches Verhalten aufweisen. Wesentlich ist dabei, daß die Objekte eingekapselt sind, also einen *lokalen Zustand* und *lokale Aktionen* beinhalten. Das Systemverhalten ergibt sich dann aus dem lokalen Verhalten der Komponentenobjekte und der Kommunikation zwischen ihnen.

Unser Ansatz eines Modellierungsformalismus für dynamische Informationssysteme versucht nun, die Konzepte semantischer Datenmodelle mit objektorientierten Konzepten sowie Konzepten formaler Ansätze zur Softwarespezifikation zu integrieren. Die wesentlichen Einflüsse aus der formalen Spezifikation kommen aus den Gebieten algebraische Spezifikation abstrakter Datentypen [EGL89, Wir90], der algebraischen Spezifikation von Prozessen [Hoa85, Mil90] und der logischen Spezifikation reaktiver Systeme [MP92]. Ziel ist es, einen integrierten Formalismus zur deskriptiven Spezifikation zu entwickeln, der die Strukturierungskonzepte semantischer Datenmodelle mit Ansätzen zur verhaltensorientierten Modellierung vereinigt. Der Ansatz basiert auf dem OBLOG-Modell, das zuerst in [SSE87] vorgestellt wurde. Weitere Darstellungen und Erweiterungen sind u.a. in [SFSE89, SE91, EGS90, ES91, ESS92] zu finden.

Die Definition der logikbasierten abstrakten Spezifikationssprache TROLL [JSHS91] ist ein Schwerpunkt in dem von uns bearbeiteten Teilprojekt. Es soll eine Spezifikationssprache basierend auf dem OBLOG-Modell entworfen werden, die für den Einsatz in der konzeptionellen Modellierung von Informationssystemen geeignet ist. Das bedeutet im wesentlichen, daß die Sprache dem Entwickler eine Reihe von deklarativen Beschreibungsmitteln zur Verfügung stellen muß, mit denen ein gegebener Weltausschnitt angemessen dargestellt werden kann. Das Ziel dieses Teilschrittes im Entwurf ist das *konzeptionelle Modell*.

Im weiteren Verlauf des Informationssystem-Entwurfsprozesses muß dann das konzeptionelle Modell in ein ausführbares System umgesetzt werden. Wir unterscheiden hier zwei (orthogonale) Zielrichtungen:

- Die *Animation* hat die automatische Erzeugung einer *ausführbaren* Spezifikation aus einer konzeptionellen Spezifikation auf gleicher Abstraktionsebene zum Ziel, um ein konzeptionelles Modell zu simulieren und die *Korrektkeit* der Spezifikation bezüglich der informellen Anforderungen soweit als möglich zu validieren.

- Die *Implementierung* bedeutet die Realisierung abstrakter Konzepte durch konkretere, insbesondere die Darstellung abstrakter Konzepte durch Dienste, die von der aktuellen Basissystemumgebung zur Verfügung gestellt werden.

Animation in frühen Entwurfsphasen ist notwendig, da nur auf diese Art und Weise die Übereinstimmung des formalen konzeptionellen Modells mit der informalen Anforderungsanalyse sichergestellt werden kann. Zur Animation werden Konstrukte eines abstrakten konzeptionellen Modells in äquivalente ausführbare Konstrukte umgesetzt. Bei der Animation spielen Kriterien wie Effizienz, ergonomische Benutzerschnittstellen, Sicherheit usw. eine untergeordnete Rolle. Wesentlich ist hier, daß eine solche Transformation automatisch durchgeführt werden kann.

Unser Ansatz basiert auf der Definition einer operationalisierbaren Kernsprache (TROLL-Kern), die eine echte Teilsprache von TROLL ist. Die weitergehenden Sprachkonstrukte von TROLL sollen dann mit Hilfe von Konstruktionen der Kernsprache dargestellt und so eine Transformation von TROLL-Spezifikationen in Spezifikationen der Kernsprache ermöglicht werden.

Zur Implementierung werden konzeptionelle Einheiten eines abstrakten Modells durch (in der Regel verschiedenartige) Einheiten eines weniger abstrakten Modells dargestellt [ES90]. Wesentliche Teilbereiche der Implementierung sind die Dekomposition komplexer Objekte und die Realisierung abstrakter Aktionen oder Operationen durch abstrakte *Transaktionen* auf einer tieferen Systemebene. Unter abstrakten Transaktionen werden in diesem Zusammenhang atomare Prozesse, also Folgen von Aktionen, verstanden. Bei der Implementierung muß der Übergang von einer mehr problemorientierten Sichtweise zu einer mehr systemorientierten Sichtweise vollzogen werden, d.h., die Gegebenheiten der zur Verfügung stehenden Systemumgebung müssen berücksichtigt werden. Ein wichtiger Punkt bei der Implementierung ist die Gewährleistung der Korrektheit eines Implementierungsschrittes bezüglich der abstrakten Spezifikation.

Ein zweiter Schwerpunkt des hier vorgestellten Projekts befaßt sich mit der Entwurfsunterstützung bei Animation und Implementierung von konzeptionellen Spezifikationen. Wesentliche Arbeitsfelder in der jetzigen Phase des Projekts sind die Definition einer möglichst einfach zu implementierenden Kernsprache der Sprache TROLL [HJ92] und die Entwicklung eines *Ausführungsmodells* für TROLL-Spezifikationen.

Das hier beschriebene Projekt startete im Frühjahr 1991. In der ersten Phase wurde aufbauend auf Vorarbeiten aus einem ESPRIT-Grundlagenprojekt[1] die Spezifikationssprache TROLL entwickelt [SJ91, JSS91, SJ92, JSH91, SJE92, HJS92, JHS92] und festgelegt [JSHS91]. Zur Zeit wird schwerpunktmäßig an einem Ausführungsmodell für TROLL-Spezifikationen und an der Transformation von abstrakten Sprachkonstrukten in Konstrukte der Kernsprache gearbeitet.

In diesem Papier wollen wir über die bisherigen Ergebnisse des Projekts berichten. Im nächsten Abschnitt werden die grundlegenden Konzepte unseres Ansatzes informell eingeführt. Abschnitt 3 befaßt sich mit der Modellierungssprache TROLL. Wir werden die wesentlichen Sprachkonzepte anhand von Beispielen vorstellen. Abschitt 4 beschäftigt sich mit Konzepten zur Ausführung und Animation von Spezifikationen. Wir werden auf die operationale Bedeutung von Kommunikationsstrukturen eingehen und kurz erste Ideen zur Operationalisierung abstrakter Spezifikationen diskutieren. In Abschnitt 5 werden wir zukünftige Arbeitsgebiete identifizieren und in Abschnitt 6 eine kurze Zusammenfassung präsentieren.

2 Konzepte

Das objektorientierte Paradigma hat in jüngster Zeit Einzug gehalten in den Bereich des Entwurfs von Softwaresystemen und Informationssystemen [KL89, RBP+90, BB92, CF92].

Die wesentlichen Einflüsse kommen aus der objektorientierten Programmierung [Nie89, Weg90] und den objektorientierten Datenbanken [ABD+89]. Die wichtigsten Eigenschaften lassen sich wie folgt zusammenfassen:

[1] ESPRIT Basic Research Action WG 3023 IS-CORE (Information Systems—COrrectness and REusability)

- Objekte sind *eingekapselte* Einheiten, meist im Sinne einer Menge von Operationen (*Methoden* genannt), die auf einem nicht sichtbaren lokalen Zustand arbeiten. Die Operationen bilden die *Schnittstelle* eines Objekts. Im Bereich der objektorientierten Datenbanken wird meist zwischen *Attributen* (als Leseoperationen auf Zustandsvariablen) und Methoden als Änderungsoperationen unterschieden.

- Objekte sind *eindeutig identifizierbar*. Damit können zum einen Objekte im gleichen Zustand unterschieden und zum anderen verschiedene Zustände desselben Objektes als solche identifiziert werden.

- Objekte *kommunizieren* durch das *Versenden von Nachrichten* (*message passing*). Dabei schickt ein Sender eine Nachricht an einen Empfänger, der daraufhin die passende Methode zur Reaktion auswählt.

- Objekte sind üblicherweise in *Klassen* zusammengefaßt. Klassen können dabei entweder als Muster für Instanzen dienen oder nur als "Sammelbehälter" für gemeinsame Information von Instanzen (z.B. Methodendefinitionen) dienen. Leider wird das Thema Klassen in der Literatur uneinheitlich behandelt und oft auch mit dem Begriff *Objekttyp* gleichgesetzt. In [Nie89] werden Objekttypen als Spezifikationen eingestuft, d.h., die Ausprägungen werden nicht betrachtet. In [ABD+89] werden Klassen als Zusammenfassung eines Musters für die Erzeugung von Instanzen mit der aktuellen Menge existierender Instanzen betrachtet.

- *Vererbung* wird meist auf der Klassenebene behandelt und bezeichnet die Wiederverwendung von Klassendefinitionen in anderen Klassendefinitionen. Vererbung ist in den meisten Ansätzen nur als *syntaktische* Vererbung zu sehen – der Mechanismus betrifft nur Programm*texte*, nicht jedoch Instanzen. In [ABD+89] werden jedoch auch andere Formen der Vererbung als möglich erachtet, die Instanzen in Vererbungsbeziehungen setzen, so z.B. die Untermengenbeziehung zwischen Ausprägungen von Objektklassen oder Spezialisierungen.

- Speziell für objektorientierte Datenbanken werden Erweiterbarkeit um benutzerdefinierte Datentypen, Persistenz von Objekten und Nebenläufigkeit als notwendig eingestuft [ABD+89].

Zum formalen Entwurf von Informationssystemen müssen die o.g. Charakteristika etwas modifiziert werden. Die Modifikationen werden durch den *deklarativen Charakter* von Spezifikationen und durch die *Formalisierung von Spezifikationen* notwendig. Es muß abstrahiert werden von der implementierungsnahen, operationalen Sichtweise objektorientierter Programmiersprachen und von der datenstruktur- und zugriffsorientierten Sichtweise objektorientierter Datenmodelle.

Das *Objekt als Basiseinheit des Entwurfs* kann wie folgt charakterisiert werden:

- Objekte sind die Basiseinheiten von *Struktur und Verhalten*, aus denen ein komplexes System zusammengesetzt ist.

- Objekte haben beobachtbare Eigenschaften, die durch *Datenwerte* dargestellt werden. Diese Eigenschaften werden *Attribute* genannt.

- Objekte haben einen *eingekapselten internen Zustand*, von dem einige Informationen über Attribute nach außen sichtbar sind.

- Der interne Zustand eines Objekts wird ausschließlich durch das Eintreten *lokaler Ereignisse* verändert. Ereignisse sind vordefinierte mögliche Zustandsübergänge.

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

Dieses Basiskonzept eines Objekts vereinigt wesentliche Konzepte objektorientierter Programmiersprachen mit wichtigen Konzepten aus dem Bereich Informationssysteme, welche sich bei der Modellierung von Datenbanken als sinnvoll erwiesen haben.

Um nun ganze Systeme von interagierenden Objekten darzustellen, müssen einzelne Objekte in Beziehung zueinander gesetzt werden. Auf der Modellebene sind unserer Ansicht nach zwei Konzepte ausreichend, um weitergehende Konzepte zu erklären:

- Objekte werden in *Objektklassen* zusammengefaßt. Innerhalb einer Klasse hat jede Instanz einen eindeutigen *Identifikator*.
- Objektinstanzen können eingekapselter Teil (*Unterobjekt*) einer anderen Instanz sein. Das Konzept der *Objektinklusion* stellt sicher, daß keine Eigenschaft des Unterobjekts durch die Einbettung in ein anderes Objekt verändert wird.

Wir wollen nun diese Konzepte der Modellebene genauer erläutern, bevor wir die Grundlagen der *Spezifikation* darlegen. Einzelne *Objekte* sind, intuitiv gesprochen, *beobachtbare Prozesse*, d.h., wir betrachten Sequenzen von Ereignissen und Beobachtungen über Anfangsstücken solcher Sequenzen. Im Gegensatz zu der Darstellung in [SJ91] betrachten wir hier nicht nur Sequenzen von atomaren Ereignissen, sondern Sequenzen von *Mengen gleichzeitig eintretender Ereignisse* (*Schnappschüsse*). Damit wird die Zusammensetzung von Objektmodellen zu Modellen von Systemen nebenläufiger Objekte vereinfacht.

Ein Objektmodell $ob = (P, V)$ besteht aus einem *Prozeß* P und einer *Beobachtungsstruktur* V. Ein Prozeß P ist definiert durch ein Paar (X, Λ) wobei X ein Alphabet von Ereignissen ist und Λ die Menge der zulässigen Folgen von Schnappschüssen ist. Ein Schnappschuß ist eine nichtleere Menge von Ereignissen, die konzeptionell gleichzeitig stattfinden. Die Menge Λ der zulässigen Lebensläufe muß mindestens folgende Bedingungen erfüllen:

1. Lebensläufe eines Objektes beginnen mit einem Schnappschuß, der ein lokal zu diesem Objekt spezifiziertes Geburtsereignis enthält.
2. Endliche Lebensläufe eines Objektes enden mit einem Schnappschuß, der ein lokal zu diesem Objekt spezifiziertes Todesereignis enthält. Vor dem letzten Schnappschuß treten keine Todesereignisse auf.
3. Unendliche Lebensläufe enthalten keine Schnappschüsse mit Todesereignissen.

Eine Beobachtungsstruktur V über einem Prozeß P ist definiert durch ein Paar (A, α). Dabei ist A eine Menge von (typisierten) Attributen und α eine Abbildung von endlichen Präfixen von Lebensläufen in Beobachtungen. Beobachtungen wiederum sind Mengen von *Attribut-Wert-Paaren.* Für eine genauere Darstellung dieses Objektbegriffes und seine Formalisierung sei auf [JSS91, HJS92] verwiesen.

Einzelne Objekte können zu *Objektklassen* gruppiert werden, in denen die Instanzen durch gleichartige Objektmodelle repräsentiert werden. Jeder Objektklasse ist ein *Klassentyp* zugeordnet, der einen Prototyp für Instanzen einer Klasse sowie einen *Identifikationsmechanismus* enthält. Der Identifikationsmechanismus legt einen *Namensraum* (d.h., die Menge der möglichen Namen von Instanzen) fest.

Das semantische Modell eines *Klassentyps* $ct = (ID, ob)$ besteht aus einem Wertebereich ID als Namensraum und einem prototypischen Objektmodell *ob*. Der Wertebereich ID ist definiert als die Trägermenge einer Sorte eines abstrakten Datentyps.

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

Ein *Zustand einer Objektklasse* oc wird beschrieben durch den zugeordneten Klassentyp ct und eine Zustandsfunktion $\zeta\colon ct.ID \to \mathbf{Prefix}(ct.ob.P.\Lambda)$ aus dem Namensraum ID des zugeordneten Klassentyps ct in die Menge der möglichen Anfangsstücke (*Präfixe*) von Lebensläufen im zugehörigen Objektmodell $ct.ob$.

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

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

Eine grundlegende Eigenschaft des hier dargestellten Objektmodells ist die Zusammensetzbarkeit von Objektmodellen, d.h., die Darstellung von *komplexen Objekten*. Wir wollen uns daher nun der Objektinklusion zuwenden. Seien zwei Instanzen a und b gegeben. Die Instanz b ist in die Instanz a *sicher eingebettet*, wenn gilt:

1. Die Signatur von b ist bis auf Umbenennungen in der Signatur von a enthalten.

2. Jeder Lebenslauf von b, der in a beobachtet werden kann wenn nur Ereignisse aus b betrachtet werden, ist ein korrekter Lebenslauf von b. Durch die Einbettung darf das *mögliche* Verhalten der eingebetteten Instanz also nicht erweitert werden. Hiermit ist *nicht* gesagt, daß alle möglichen Lebensläufe von b beobachtet werden können. (*„Einkapselungsprinzip für Lebensläufe“*)

3. Attribute von b werden nicht direkt durch Ereignisse von a, die nicht auch Ereignisse von b sind, verändert. Diese Eigenschaft sichert die Einkapselung von Objekteigenschaften des eingebetteten Objektmodells. (*„Lokalitätsprinzip für Attributveränderungen“*)

Mittels dieses Konzepts können Objektzusammensetzungen spezifiziert werden [JSHS91, HJS92, Saa92]. Dabei muß das Verhalten der durch Objektinklusion miteinander in Beziehung gesetzten Instanzen geeignet *synchronisiert* werden und *Kommunikation* durch Datenfluß möglich sein. Dazu führen wir den *Ereignisaufruf* als Kommunikationsprimitiv ein.

Ein Ereignisaufruf definiert eine *asymmetrische, synchrone* Kommunikation und ist definiert auf Prozessen: Ruft ein Ereignis x_1 ein Ereignis x_2 auf (notiert als x_1 >> x_2), so müssen alle Schnappschüsse, die x_1 enthalten, auch x_2 enthalten; d.h. also, wann immer x_1 stattfindet, muß auch x_2 stattfinden (nicht jedoch umgekehrt). Das Konzept des Ereignisaufrufs ermöglicht damit die konzeptionelle Modellierung eines Kontrollflusses.

Ein Ereignis kann auch mehrere andere Ereignisse aufrufen, die mehreren Instanzen zugeordnet sein können (*Multicasting*). Durch die Forderung nach Synchronität müssen auf der konzeptionellen Ebene die Adressaten von Aufrufen immer explizit festgelegt werden. Ein Ereignisaufruf kann nur dann stattfinden, wenn alle Adressaten zum Zeitpunkt des Aufrufs existieren (Ausnahme: aufgerufene Geburtsereignisse) und die aufgerufenen Ereignisse in den Lebensläufen der beteiligten Objekte stattfinden dürfen—andernfalls kann die gesamte Aufrufkette nicht stattfinden.

Während durch den Aufruf von Ereignissen der Kontrollfluß festgelegt ist, kann der *Datenfluß* prinzipiell auch entgegen dem Kontrollfluß erfolgen. Zur Beschreibung des Datenflusses werden formale Ereignisparameter verwendet.

Zur *Spezifikation* solcher Modellstrukturen werden verschiedene formale Sprachen verwendet, die in Systembeschreibungen in der Sprache TROLL kombiniert werden. In TROLL wird eine mehrsortige Logik erster Ordnung, eine Lineare Temporale Logik erster Ordnung [Saa88, SL89] sowie eine auf CSP [Hoa85] basierende Sprache zur Spezifikation von (Teil-)Prozessen benutzt.

Die Interpretationsstrukturen der benutzten temporalen Logik sind Folgen von einfachen Algebren, die jeweils einen Zustand repräsentieren. Die Beschreibung von Zustandsübergängen durch Ereignisse in temporaler Logik wird ermöglicht durch die Definition eines **after**-Prädikats, das genau in den Zuständen gilt, die durch das Stattfinden eines Ereignisses erreicht werden. Aus Platzgründen können wir hier nicht genauer auf die Syntax und Semantik der Teilsprachen zur logischen Spezifikation eingehen. Die Teilsprachen werden in [JSHS91] beschrieben.

3 Die Sprache TROLL

Die Basiseinheit der Spezifikation von Objektgesellschaften in der Sprache TROLL ist eine *Objektbeschreibung*, engl. *template*. Eine Objektgesellschaft ist eine Menge von *interagierenden* aber eigenständigen Objekten. Eine Objektbeschreibung ist eine generische Beschreibung einer Prototyp-Instanz und beschreibt die Struktur und das Verhalten von Instanzen. In der Datenbankterminologie entspricht eine Objektbeschreibung vereinfacht gesehen einem *Schema*. Eine Objektinstanz entspricht dann einem passenden Tupel zu diesem Schema.

Um eine Menge gleichartiger Objekte zu beschreiben, werden *Objektklassen* spezifiziert. Eine Spezifikation einer Objektklasse definiert implizit auch den *Klassentyp*. Der Klassentyp beschreibt die *potentiellen* Ausprägungen von zugehörigen Objektklassen. Die Objektklasse definiert dann die aktuellen Ausprägungen (also eine sich zeitlich ändernde Menge von Instanzen) einer Klassentypdefinition. In der Datenbankterminologie entspricht einem Klassentyp ein Relationenschema und einer Klasse eine Relationenausprägung.

3.1 Objekt- und Klassenbeschreibungen

Wir werden im folgenden, um die wesentlichen Eigenschaften der Sprache TROLL zu erläutern, Beispiele aus der Modellierung einer Bankanwendung darstellen. Für diese Bankanwendung werden sukzessive Konten (engl. accounts), Geldautomaten (engl. Automatic Teller Machines, ATM's), die Bank selbst als ein einzelnes Objekt sowie ein Kommunikationskanal zwischen der Bank und den Geldautomaten modelliert. Für dieses Beispiel werden nur die zum Verständnis wichtigen Teile angegeben. Das vollständige Beispiel kann in [JSHS91] gefunden werden.

In TROLL folgt eine Objektbeschreibung nach dem Schlüsselwort **template** und beinhaltet die Deklaration von Attributen (**attributes**) und Ereignissen (**events**) sowie verschiedene Abschnitte zur Spezifikation der zulässigen Beobachtungen und des zulässigen Verhaltens.

Die Deklaration von Attributen und Ereignissen legt die *Signatur* einer Objektbeschreibung fest. Attribute in TROLL sind *typisiert*, sie können nur Werte aus der Wertemenge des Attributdatentyps annehmen. Attribute können als **constant** markiert werden—der mit der Geburt der Instanz angenommene Wert darf sich dann über die Existenzdauer einer Instanz nicht ändern. Als Beispiel möchten wir die Attributdeklaration für eine Beschreibung von Konten-Objekten geben:

```
attributes
  constant Holder:|BankCustomer|;
  Balance:money;
  CreditLimit:money;
  Cards:set(|CashCard|);
```

Eine Besonderheit stellt der Datentyp `|BankCustomer|` dar, der als Wertebereich die *Menge der möglichen Identifikatoren* für Instanzen der Klasse `BankCustomer` (die hier nicht aufgeführt wird) hat. Ein solcher Datentyp wird implizit mit der Spezifikation eines Klassentyps bereitgestellt (s.u.).

Ereignisdeklarationen bestehen aus einen Ereignisnamen und eventuellen Parametern für Ereignisse. Die Parameter sind Werte eines Datentyps und können zusätzlich optional als Eingabeparameter (Schlüsselwort **in**) oder Ausgabeparameter (Schlüsselwort **out**) gekennzeichnet werden. Die

Geburtsereignisse werden durch das Schlüsselwort **birth** markiert, die Todesereignisse durch das Schlüsselwort **death**. Für Konten-Objekte könnte man z.B. folgende Ereignisse deklarieren:

```
events
  birth open(in Holder:|BankCustomer|);
  death close;
  new_credit_limit(in Amount:money);
  assign_card(in C:|CashCard|);
  cancel_card(in C:|CashCard|);
  update_request(in Type:UpdateType, in Amount:money);
  update_failed;
  withdraw(in Amount:money);
  deposit(in Amount:money);
```

Dabei wird mit dem Geburtsereignis `open` auch der Wert des konstanten Attributs `Holder` festgelegt. Das Ereignis `update_request` dient der Anforderung einer Kontenbewegung von außen und wird im Lebenslauf eines Konten-Objektes entweder von einem der Ereignisse `withdraw`, `deposit` oder `update_failed` gefolgt (s.u.). Die restliche Objektbeschreibung gliedert sich in folgende Abschnitte:

- *Integritätsbedingungen* unter dem Schlüsselwort **constraints** spezifizieren analog zu statischen und dynamischen Integritätsbedingungen im Datenbankbereich die möglichen Attributwerte bzw. deren zeitliche Entwicklungen. Die für Integritätsbedingungen benutzte Sprache ist eine *zukunftsgerichtete temporale Logik* erster Ordnung mit Operatoren wie **sometimef** („Irgendwann in der Zukunft wird gelten“) oder **alwaysf** („Immer in der Zukunft gilt“). Diese Logiksprache wird ausführlich in [JSHS91], Kapitel 2.3, beschrieben.

- Die *Effekte von Ereignissen auf Attributwerte* werden im Abschnitt unter **valuation** beschrieben. Die dort spezifizierten Regeln bieten einen eingeschränkten Mechanismus zur Spezifikation von Nachbedingungen für Ereignisse an.

- *Vorbedingungen* für das Eintreten von Ereignissen werden im Abschnitt **permissions** definiert. Ereignisse können im aktuellen Zustand nur dann auftreten, wenn ihre Vorbedingung erfüllt ist. Vorbedingungen können sich auch auf die Vergangenheit einer Objektinstanz beziehen. Dabei wird eine *vergangenheitsgerichtete temporale Logik* verwendet, die u.a. Operatoren wie **sometime** („Es galt irgendwann“) oder **always** („Es galt immer“) enthält. Diese Logiksprache wird ebenfalls in [JSHS91], Kapitel 2.3, beschrieben.

- *Lebendigkeits-* bzw. *Vollständigkeitsbedingungen* für Lebensläufe werden unter dem Schlüsselwort **obligations** formuliert. Hier werden Ziele definiert, die eine Instanz im Laufe ihrer Existenz erfüllen *muß*.

- *Aktivitäten* werden unter **commitments** definiert. Hier werden Reaktionen spezifiziert, die eine Instanz beim Eintreten von Bedingungen selbständig ausführen muß, falls dies nicht durch äußere Bedingungen verhindert wird.

- *Prozesse* werden unter dem Schlüsselwort **patterns** spezifiziert. Hier können Teilstücke möglicher Lebensläufe explizit mit Hilfe einer CSP-ähnlichen Untersprache [Hoa85] definiert werden.

Als Beispiel geben wir die Definition einer Objektklasse an. Die Klasse `Account` definiert die Klasse der Konten in der bereits skizzierten Bankanwendung. Mit der Spezifikation wird ein Datentyp `|Account|` implizit festgelegt. Der Namensraum für Instanzen dieser Klasse ist isomorph zu der Menge der natürlichen Zahlen. Die weiterere Objektbeschreibung enthält (temporale) Integritätsbedingungen, (bedingte) Auswertungsregeln und (temporale) Vorbedingungen für Ereignisse.

```
object class Account
  identification
    data types nat;
    No:nat;
  template
    data types nat, |BankCustomer|, money, UpdateType;
    attributes
      constant Holder:|BankCustomer|;
      Balance:money;
      CreditLimit:money;
    events
      birth open(in Holder:|BankCustomer|);
      death close;
      new_credit_limit(in Amount:money);
      update_request(in Type:UpdateType, in Amount:money);
      update_failed;
      withdraw(in Amount:money);
      deposit(in Amount:money);
    constraints
      initially CreditLimit = 0 and Balance = 0;
      (Balance < 0) ==> sometimef(Balance >= 0);
    valuation
      variables m:money; C:|CashCard|;
      { m > CreditLimit } ==> [new_credit_limit(m)]CreditLimit = m;
      [withdrawal(m)]Balance = Balance - m;
      [deposit(m)]Balance = Balance + m;
      ...
    behavior
      permissions
        variables t,t1:UpdateType; m,m1,m2:money;
        { Balance = 0 } close;
        { not sometime(after(update_request(t1,m1)))
          since last(after(update_failed) or
                     after(deposit(m2)) or
                     after(withdrawal(m2)))} update_request(t,m);
end object class Account;
```

Die für den Identifikationsmechanismus und die Templatebeschreibung notwendigen Datentypen müssen in einer Spezifikation aus dem unterliegenden Datentyp-Universum importiert werden. TROLL selbst sieht keine Spezifikation von Datentypen vor; wir nehmen einen Formalismus zur Spezifikation von Datentypen als gegeben an. Um die Beispiele kurz zu halten, verzichten wir im folgenden auf den Import von Datentypen.

Im Abschnitt **valuation** werden die Effekte von Ereignissen auf Attributwerte spezifiziert. Eine allgemeine Regel der Form:

```
{ m > CreditLimit } ==> [new_credit_limit(m)]CreditLimit = m;
```

muß gelesen werden als „Nach Eintreten des Ereignisses `new_credit_limit`, instantiiert mit einem aktuellen Parameter `m`, nimmt das Attribut `CreditLimit` den Wert des Terms auf der rechten Seite, ausgewertet im Zustand vor Eintreten des Ereignisses, an" (in diesem Fall ist der Term nicht zustandsabhängig). Diese Regel ist nur gültig, wenn die Bedingung `m > CreditLimit` im Zustand *vor* Eintreten des Ereignisses wahr ist (allgemeiner Fall einer Auswertungsregel). Ein solches Ereignis darf eintreten, hat aber möglicherweise keinen *über Attribute sichtbaren* Effekt.

Vorbedingungen für Ereignisse können jedoch das Eintreten eines Ereignisses in einem bestimmten Objektzustand verhindern. Diese Vorbedingungen können auch auf die Vergangenheit einer Instanz

Bezug nehmen. Dazu wird eine Untersprache einer Temporalen Logik eingesetzt. Im Beispiel wird folgende Vorbedingung für das Ereignis `update_request` formuliert:

```
{ not sometime(after(update_request(t1,m1)))
  since last(after(update_failed) or
             after(deposit(m2)) or
             after(withdraw(m2)))} update_request(t,m);
```

Diese Bedingung spezifiziert, daß nicht zwei Ereignisse vom Typ `update_request` aufeinander folgen dürfen—nach der „Beantwortung“ einer `update_request`-Anforderung durch eines der Ereignisse `update_failed`, `deposit` oder `withdraw` darf höchstens ein Ereignis vom Typ `update_request` folgen.

Das Ereignis `update_request` stellt in diesem Beispiel die Anforderung zur Änderung des Kontostandes von Konten-Objekten (`Balance`) dar. Die Ereignisse `withdraw` und `deposit` und sind intern und werden von den Konteninstanzen in eigener Initiative ausgelöst (hier nicht gezeigt). In einer vollständigen Spezifikation wären sie durch ein *Interface* [SJ92] verborgen. Wir haben in dieser Kurzdarstellung darauf sowie auf die Darstellung dieser Eigeninitiative bzw. Aktivität (**commitments**) aus Platzgründen verzichtet.

3.2 Strukturierung von Objektbeschreibungen

Objekte können in vielfältigen Beziehungen zueinander stehen. Neben expliziten Kommunikationsbeziehungen (die als *Relationships* weiter unten vorgestellt werden) sind im Bereich der semantischen Datenmodellierung im wesentlichen zwei andere Beziehungen bekannt:

- die *is_a* oder Spezialisierungsbeziehung und
- die *part_of* oder Aggregationsbeziehung.

Bei der is_a-Beziehung werden verschiedene *Aspekte* desselben konzeptionellen Objekts beschrieben, also z.B. Eigenschaften eines Kunden als ein Aspekt einer Person. In TROLL unterscheiden wir zwischen *Rollen* und *Spezialisierungen*. Rollen werden von Instanzen der Basisklassen *zeitweilig* während ihres Lebens gespielt. Dabei kann ein konzeptionelles Objekt gleichzeitig mehrere Rollen spielen. Als Beispiel sei eine `Person` genannt, die in einer Rolle `BankCustomer` und in einer Rolle als `Patient` bei einem Arzt beobachtet werden kann.

Bei der Spezialisierung wird dagegen ein *statischer* Aspekt eines konzeptionellen Objekts spezifiziert, also z.B. die Spezialisierung einer `Person` zu `Frau`. Spezialisierung kann daher als ein Spezialfall der Rollenbeziehung angesehen werden, einer Rolle, die mit der Geburt des Objektes festgelegt ist und nicht geändert werden kann. Wir wollen in diesem Beitrag aus Platzgründen nicht weiter auf Aspekte und die dadurch implizierte Mehrfachvererbungshierarchie eingehen und verweisen auf [JSHS91].

Die part_of-Beziehung beschreibt *komplexe Objekte*, die aus Komponenten zusammengesetzt sind. Die Komponenten sind dabei vollständige Objekte. Das Verhalten der Komponenten ist durch Interaktionen mit anderen Komponenten und dem umschließenden komplexen Objekt *synchronisiert*. Interaktionen sind die einzige Möglichkeit, den Zustand einer anderen Komponente zu beinflussen—eine direkte Manipulation ist durch das Lokalitätsprinzip für Attributveränderungen ausgeschlossen.

In diesem Beitrag wollen wir nur die wichtigste Variante komplexer Objekte in TROLL vorstellen, die *dynamischen komplexen Objekte*. Hier kann sich die Zusammensetzung eines komplexen Objekts im Laufe seiner Lebenszeit verändern. Das Einschließen und das Herauslösen von Komponenten geschieht durch spezielle Ereignisse, die mit der Benutzung von Komponentenkonstruktoren implizit definiert sind.

Als dynamisch veränderbare Komponenten sieht TROLL neben Einzelinstanzen, die durch die Angabe des Klassennamens bestimmt werden, auch die Definition von Mengen (**SET**-Konstruktor) und Listen (**LIST**-Konstruktor) von Komponenteninstanzen einer Klasse vor.

Da das zugrundeliegende Objektmodell nur die statische Zusammensetzung von Objekten erlaubt, muß die Semantik dynamischer komplexer Objekte zweistufig definiert werden: Alle möglichen Zusammensetzungen eines dynamischen komplexen Objekts werden durch statische komplexe Objekte dargestellt, deren Zusammensetzung konstant ist. Die Semantik eines dynamischen komplexen Objekts wird nicht direkt sondern über das den augenblicklichen Zustand repräsentierenden statische komplexe Objekt definiert. Dieses Thema wird ausführlich in [HJS92] diskutiert.

Als Beispiel wollen wir eine vereinfachte Bank modellieren. Die Bank ist hier nur ein *Einzelobjekt*; es besteht in TROLL nicht die Notwendigkeit, in jedem Fall eine Klasse zu definieren. Das Bank-Objekt enthält hier eine Menge von Konten-Objekten und ein Personen-Objekt (nicht weiter spezifiziert) als Komponenten. Die Synchronisation der Komponenten mit dem komplexen, umschließenden Objekt `Bank` wird im **interaction**-Abschnitt spezifiziert:

```
object Bank
  template
    components
      Manager:Person;
      Accounts:SET(Account);
    events
      birth establish;
      death close_down;
      open_account(in No:nat);
      close_account(in No:nat);
      ...
    behavior
      permissions
        variables n:nat;
        { not Accounts.IN(Account(n)) } open_account(n);
        { sometime after(open_account(n)) } close_account(n);
        ...
    interaction
      variables n:nat;
      open_account(n) >> Accounts.INSERT(Account(n));
      open_account(n) >> Accounts(Account(n)).open;
      close_account(n) >> Accounts.REMOVE(Account(n));
      close_account(n) >> Accounts(Account(n)).close;
      ...
end object Bank;
```

Die in der **interaction**-Sektion von den entsprechenden `open` und `close` Ereignissen aufgerufenen `INSERT` und `REMOVE` Ereignisse werden implizit mit dem **SET**-Konstruktor für die entsprechende Komponente (hier `Accounts`) definiert. Diese Sichtweise von komplexen Objekten ist operational und spiegelt die Dynamik einer sich verändernden Welt wider.

Zu bemerken ist hier, daß Konten-Objekte *nicht nur in einer Bank* Komponenten sein können. Gemeinsame Komponenten verschiedener Objekte stellen dann eine gemeinsam benutzte Ressource dar. Aus diesem Grunde werden hier auch die Geburts- bzw. Todesereignisse der Konten-Objekte von den die Struktur des komplexen Objektes verändernden Ereignissen wie `INSERT` und `REMOVE` getrennt betrachtet. Die Tatsache, daß die Konten-Objekte bei der Einbindung in das komplexe Objekt `Bank` erzeugt werden, ist somit ein Spezialfall. Für eine komplette Darstellung der Konstruktoren für dynamische komplexe Objekte sei auf [JSHS91] verwiesen.

3.3 Beziehungen zwischen Objekten

Als letztes Sprachkonstrukt von TROLL wollen wir in dieser kurzen Einführung das *Relationship*-Konstrukt vorstellen. Zur Motivation sei zuerst eine Klasse von Geldautomaten (ATMs) spezifiziert, deren Instanzen über ein Kommunikationsnetz mit der Bank in Verbindung stehen, ansonsten aber unabhängig arbeiten. Ein Geldautomat enthält eine änderbare Menge an Geld. Wir spezifizieren außerdem ein *abgeleitetes Attribut* `Empty`, das über den Werten anderer Attribute berechnet wird. Es zeigt an, ob der Geldautomat leer ist.

Die Ereignisse definieren die möglichen Aktionen und Zustandübergänge des Geldautomaten. Wichtig sind hier nur die Ereignisse `check_card_w_bank` (welches das Signal zur Überprüfung einer Scheckkarte an die Bank darstellt) sowie die Signale `card_accepted`, `bad_PIN_msg` (ungültige Geheimnummner) und `bad_account_msg` (ungültige Kontonummer), die von außen angestoßen werden. Wir haben aus Platzgründen die vollständige Spezifikation des Verhaltens der Geldautomaten wie auch der Bank (s.o.) nicht aufgeführt. Sie kann in [JSHS91], S. 39–40, gefunden werden.

```
object class ATM
  identification
    IDNumber:nat
  template
    attributes
      CashOnHand:money;
      derived Empty:bool;
    events
      birth set_up;
      death remove;
      refill(in Amount:money);
      read_card(in C:|CashCard|);
      check_card_w_bank(in Acct:nat:, in PIN:nat);
      bad_PIN_msg; bad_account_msg; card_accepted;
      dispense_cash(in Amount:money);
    constraints
      initially CashOnHand = 0;
    derivation
      Empty = (CashOnHand < 100);
    valuation
      variables m:money;
      [refill(m)]CashOnHand = CashOnHand + m;
      [dispense_cash(m)]CashOnHand = CashOnHand - m;
    behavior
      permissions
        variables n:nat; m:money; C:|CashCard|;
        { not Empty } read_card(C);
      ...
end object class ATM
```

Wir möchten nun die Kommunikationsbeziehungen zwischen Geldautomaten und der Bank spezifizieren, mit denen diese Komponenten eines (verteilten) Systems zusammengesetzt werden. Dazu wird das Konstrukt der *Relationships* benutzt.

Relationships ermöglichen das Einbinden von ansonsten unabhängigen Komponenten in eine Systemumgebung. Durch das explizite Modellierungskonstrukt **relationship** wird die Spezifizierung von kontextabhängigen Interaktionen innerhalb einer Objektbeschreibung vermieden. Dadurch wird die Modularisierung eines Entwurfs erhöht und die Wiederverwendbarkeit von Objektbeschreibungen (die ja nur lokale Informationen enthalten) unterstützt. Eine Diskussion des Relationship-Konzepts kann in [JHS92] gefunden werden.

Im folgenden Beispiel werden lokale Ereignisse in einem Geldautomaten und in der Bank synchronisiert. Dazu werden hier *Ereignisaufrufe* benutzt: Ruft ein Ereignis e_1 ein Ereignis e_2 auf (notiert als e_1 `>>` e_2), so impliziert jedes Auftreten von e_1 ein *synchrones* Auftreten von e_2.

```
relationship RemoteTransaction between Bank,ATM;
```

```
  interaction
    variables atm:|ATM|; n,p:nat;
    ATM(atm).check_card_w_bank(n,p) >> Bank.verify_card(n,p,atm);
    Bank.no_such_account(atm) >> ATM(atm).bad_account_msg;
    Bank.bad_PIN(atm) >> ATM(atm).bad_PIN_msg;
    Bank.card_OK(atm) >> ATM(atm).card_accepted;
end relationship RemoteTransaction;
```

Der Ereignisaufruf ist hier syntaktisch analog zum Ereignisaufruf innerhalb von komplexen Objekten zu sehen. Im Unterschied dazu wird die Kommunikation hier nicht *innerhalb* eines Objektes spezifiziert. Der Aufruf:

```
ATM(atm).check_card_w_bank(n,p) >> Bank.verify_card(n,p,atm);
```

bedeutet also, daß das Auftreten eines `check_card_w_bank`-Ereignisses in einer Instanz der Klasse `ATM` das Auftreten eines `verify_card`-Ereignisses in der `Bank`-Instanz impliziert. Hierbei serialisiert die `Bank`-Instanz konkurrierende Anforderungen verschiedener Geldautomaten, da in der `Bank`-Instanz jeweils nur ein Ereignis vom Typ `verify_card` zu einen Zeitpunkt auftreten kann.

Die Analogie des Ereignisaufrufes in komplexen Objekten und im Relationship-Konstrukt kann nunmehr auch zur Festlegung der Semantik dieser Art von Beziehungsspezifikation genutzt werden. Die Definition eines Relationships kann in ein *den beteiligten Objekten gemeinsames Teilobjekt* (einen Kanal) transformiert werden, welches die Kommunikation vermittelt [JHS92, HJ92].

Mit diesen Bemerkungen wollen wir die Vorstellung der wichtigsten Sprachelemente der Sprache TROLL beschließen. Der volle Sprachumfang der ersten Version von TROLL ist in [JSHS91] beschrieben. Zur Zeit wird ein Parser mit kontextsensitiver Analyse im Rahmen zweier Diplomarbeiten realisiert [Ste92, Stö92].

4 Ausführung von Spezifikationen

In diesem Abschnitt werden wir erste Ergebnisse bezüglich der Ausführung bzw. Animation von abstrakten TROLL-Spezifikationen vorstellen.

Das zentrale Problem liegt darin, daß die in TROLL enthaltenen deklarativen Sprachmittel eine direkte Ausführung von TROLL-Spezifikationen verhindern. Diese Sprachmittel sind jedoch für die konzeptionelle Modellierung unverzichtbar, da gerade hier eine abstrakte, deklarative Spezifikation gefordert ist [Saa92].

4.1 Deklarative und operationale Sprachanteile

Zu den deklarativen Anteilen in TROLL sind im wesentlichen die Techniken zur Beschreibung der erlaubten *Lebenszyklen*, der erlaubten *Attributentwicklungen* sowie den Sprachmitteln zur *Strukturierung von Spezifikationen* zu zählen. Zu der letztgenannten Gruppe gehören sowohl die Vererbungshierarchie als auch die Konstruktoren für komplexe Objekte.

Im folgenden werden kurz die für eine direkte Ausführung von Spezifikationen als problematisch erkannten Konzepte der Sprache aufgelistet und erste Lösungsvorschläge angegeben:

- Ereignisse können durch *Temporale Vorbedingungen* näher spezifiziert, d.h., ihr Eintreten in bestimmten Lebensläufen verboten werden. Da für eine Auswertung temporaler Formeln die Aufzeichnung der gesamten Historie eines Objektes nicht praktikabel ist, soll eine geeignete Repräsentation von *notwendiger* historischer Information erzeugt werden [Sch92, SS92].

- Temporale Integritätsbedingungen (*constraints*)—das Problem der Umsetzung/Implementierung deklarativer Integritätsbedingungen wurde intensiv erforscht. Ein Ansatz zur Implementierung sind z.B. integritätserhaltende Transaktionen wie sie in [Lip89] vorgestellt wurden.

- Die TROLL Spezialisierungshierarchie definiert für die beteiligten Klassentypen eine syntaktische sowie für die Objektinstanzen eine semantische Vererbungsrelation in Form von *is-a* Beziehungen. Hier ist die Verwendung der Objektinklusion, die eine syntaktische sowie eine semantische Einbettung von Teilobjekten formalisiert, als mögliche Darstellung der Vererbungshierarchie geplant. Vererbung wird damit in eine Form von *Delegation* [Ste87] überführt.

- Im Falle der Konstruktoren für komplexe Objekte ist—wie schon in Abschnitt 3 angedeutet—die Semantik von Spezifikationen komplexer Objekte nur zweistufig erklärbar [HJS92]. Da diese zweistufige Semantikfestlegung aus Effizienzüberlegungen (selbst für die Animation) ausscheidet, ist eine Darstellung von dynamischen komplexen Objekten durch geeignete Kommunikationsbeziehungen vorgesehen.

- Relationships können als Kommunikationsobjekte repräsentiert werden. Mit Hilfe der Objekteinbettung lassen sich diese Objekte dann als den in Beziehung stehenden Objekten bekannte und die Kommunikation vermittelnde Objekte [HJ92, JHS92] darstellen.

Ein wesentliches Problem bei deklarativen Spezifikationen besteht darin, daß unerwünschte Zustandsänderungen explizit ausgeschlossen werden müssen (*Frame-Problem*) [HR92, Lip89].

Die vorgestellte Sprache TROLL enthält jedoch auch einen starken Anteil operationaler Sprachanteile: Das Konzept der Ereignisse und deren Auswirkungen auf die sichtbaren Objekteigenschaften (Attribute) ist ein zentrales Konzept in unserem Ansatz. Wesentlich ist, daß Zustandsänderungen nur durch lokal definierte Ereignisse möglich sind. Die direkt operationalisierbaren Sprachanteile in TROLL sind die folgenden:

- Einfache, datenwertige Attribute als Abstraktion von *Instanzvariablen*.

- Ereignisse als Abstraktion von *Methoden*.

- Einfache, nicht-temporale Vorbedingungen für Ereignisse (*simple permissions*), formuliert in Formeln der Prädikatenlogik erster Stufe (eingeschränkt auf Quantoren über endlichen Bereichen).

- Auswertungsregeln als zuweisungsorientierte, einfache Nachbedingungen für Ereignisse.

- Ereignisaufrufe (*event calling*) als Kommunikationsprimitiv und Abstraktion von *Nachrichten*.

- Sichere Objektinklusion als Basiskonzept zur Zusammensetzung von Objekten.

- Einfache, nicht temporale Integritätsbedingungen.

Diese Sprachanteile haben einen direkten Bezug zu objektorientierten Programmiersprachen. Unser Ansatz zur Ausführung von Spezifikationen besteht nun darin, eine *Kernsprache* von TROLL zu definieren, die nur operationalisierbare Konstrukte enthält, und dann alle weitergehenden Sprachanteile von TROLL durch äquivalente Konstruktionen der Kernsprache darzustellen. Eine entsprechend formalisierte Umsetzung definiert dann eine Menge von *formalen Transformationen*, mit denen deklarative Konstrukte korrektheitserhaltend in operationalisierbare Konstrukte umgesetzt werden können.

Als nicht direkt, jedoch mit geringem Transformationsaufwand operationalisierbar können die folgenden Konzepte angesehen werden:

- Vollständigkeitsbedingungen für Lebensläufe (*obligations*) können umgesetzt werden in Vorbedingungen für Todesereignisse.

- Prozessbeschreibungen können umgesetzt werden in *Reihenfolgebedingungen* für das Eintreten von Ereignissen.

Zum gegenwärtigen Zeitpunkt sind unsere Arbeiten bezüglich der Formalisierung einer Umsetzung von TROLL-Spezifikationen in die Kernsprache erst am Anfang. Bisher wurde ein erster Vorschlag zur Definition einer Kernsprache gemacht und die Kernsprache auf die Spezifikation verteilter Systeme angewandt [HJ92]. Die Definition der Umsetzung deklarativer Sprachkonstrukte von TROLL wird ein zukünftiger Schwerpunkt unserer Arbeit sein.

4.2 Ausführung von TROLL-Kern Spezifikationen

Durch korrektheitserhaltende Transformationen wird das Problem der Ausführung von TROLL-Spezifikationen reduziert auf das Problem der Ausführung von Spezifikationen in der Kernsprache. Die wesentlichen Arbeitsschwerpunkte sind dabei:

- die Definition eines Ausführungsmodells für Ereignisse und Ereignisaufrufe und
- die Darstellung der sicheren Objektinklusion als Vorbedingung für die Kommunikation.

Ereignisaufrufe in TROLL bewirken, daß alle *transitiv* durch ein Ereignis aufgerufenen Ereignisse synchron stattfinden, falls sie im augenblicklichen Zustand erlaubt sind. Das bedeutet, daß die gesamte Aufrufkette nicht ausgeführt werden kann, falls auch nur ein Ereignis in der Kette aufgrund von unerfüllten Vorbedingungen verhindert wird.

Als Beispiel für eine relativ komplexe Aufrufkette wollen wir ein etwas vereinfachtes Beispiel aus der Bankwelt mit TROLL-Kern-Konstrukten formulieren. Wir spezifizieren wiederum Konten-Objekte, nun aber in der folgenden vereinfachten Form:

```
object class SimpleAccount
  identificaton
    No:nat;
  template
    attributes
      Balance:money;
      Count:nat;
    events
      birth open; death close;
      deposit(money); withdraw(money);
      countBooking;
    valuation
      variables m:money;
      [deposit(m)]Balance=Balance+m;
      [withdraw(m)]Balance=Balance-m;
      [countBooking]Count=Count+1;
    behavior
      permissions
        variables m:money;
        {Balance>=m}withdraw(m);
    interaction
      variables m:money;
      withdraw(m) >> countBooking;
      deposit(m) >> countBooking;
end object class SimpleAccount
```

Die so spezifizierten Konten-Objekte haben nur einfache Ereignisse zum Überweisen (`deposit`) und Abheben (`withdraw`) von Geldbeträgen, zusätzlich aber ein Ereignis zum Zählen von Buchungen (`countBooking`). Das Ereignis `countBooking` wird bei jedem `deposit`- bzw. `withdraw`-Ereignis synchron aufgerufen. Die so spezifizierten einfachen Konten-Objekte werden nun in einer folgendermaßen modellierten Bank benutzt (wiederum mit der TROLL-Kernsprache spezifiziert):

```
object SimpleBank
  template
    including A in SimpleAccount as Acct;
    attributes
      AccTransactionCount:nat;
    events
      ...
      transfer(in From:|SimpleAccount|,in To:|SimpleAccount|,in Amount:money);
      countBankTransaction;
      ...
    valuation
      [countBankTransaction]AccTransactionCount=AccTransactionCount+1
      ...
    interaction
      variables m:money;A,A1,A2:|SimpleAccount|
      transfer(A1,A2,m) >> Acct(A1).withdraw(m);
      transfer(A1,A2,m) >> Acct(A2).deposit(m);
      Acct(A).countBooking >> countBankTransaction;
      ...
end object SimpleBank
```

Das Objekt `SimpleBank` ist, mit Hilfe des **including**-Konstruktes, zusammengesetzt aus *allen möglichen* Objekten des Klassentyps `SimpleAccount`. Dieses Sprachkonstrukt beschreibt dabei den TROLL-Kern-Mechanismus zur Einbettung von Objektmodellen (Inklusion) wie er in Abschnitt 2 dargestellt wurde. Auf Teilobjekte der Klasse `SimpleAccount` kann innerhalb des Objektes `SimpleBank` mit dem lokalen Klassennamen `Acct` und einem Identifier aus `|SimpleAccount|` zugegriffen werden.

Mit dem Ereignis `transfer` kann eine Überweisung von einem Konto auf ein anderes Konto innerhalb derselben Bank ausgeführt werden. Dazu werden in den entsprechenden Konten-Teilobjekten die Ereignisse `withdraw` und `deposit` aufgerufen. Dieser Aufruf nun bewirkt wiederum einen Aufruf des Ereignisses `countBooking` in beiden Konten-Objekten, diese wiederum einen Aufruf des `countBankTransaction`-Ereignisses im Bank-Objekt selbst.

Hier stellt sich nun sofort die Frage, ob letzteres Ereignis *einmal* oder *zweimal* stattfinden soll. Die Intention des Entwerfers des Bank-Objektes wird das einmalige Eintreten gewesen sein, da er auch ein einzelnes Ereignis `transfer` innerhalb der Bank spezifiziert hat. Diese Interpretation stimmt in diesem Fall auch mit dem *Ausführungsmodell* überein—dort legen Aufrufketten Mengen gleichzeitig auftretender Ereignisse fest. Die gesamte Menge von Ereignissen, die durch das `transfer`-Ereignis angestoßen wird, kann nur stattfinden, wenn für *alle* betroffenen Ereignisse die Vorbedingungen erfüllt sind. In diesem Fall ist es möglich, daß auf dem Ausgangskonto nicht genug Geld vorhanden ist. Die Vorbedingung für das `withdraw`-Ereignis in den Konten-Objekten verhindert also unter Umständen die gesamte Aufrufkette.

Ereignisaufrufe sind zum einen ein operationales Sprachkonstrukt, bergen durch den Aufruf beliebig vieler weiterer Ereignisse zudem aber ein Potential zur Formulierung komplexer Zustandsübergänge. Die Möglichkeit, Aufrufketten beliebiger Länge zu spezifizieren, kann als deklarativ und für eine Spezifikation von komplexen Zustandsübergängen als unbedingt notwendig eingestuft werden.

Als weiteres Beispiel sei ein Objekt `DezimalZähler` angeführt. Dieses Objekt habe als Attribute eine potentiell unendliche Anzahl von *Stellen*, die ganzzahlige Werte von '0'–'9' annehmen können, ein Ereignis *Zähle* sowie ein Ereignis zum Verarbeiten eines Übertrags (`SetzeStelleWeiter`):

```
object DezimalZähler
  template
    attributes
      Stelle(nat):nat;
    events
      birth ErzeugeZähler;
```

```
      Zähle;
      SetzeStelleWeiter(nat);
    constraints
      variables n:nat;
      Stelle(n) <= 9;
    valuation
      variables n:nat;
      [ ErzeugeZähler ] Stelle(n) = 0;
      { Stelle(n) < 9 } ==>
            [ SetzeStelleWeiter(n) ] Stelle(n) = Stelle(n) + 1;
      { Stelle(n) = 9 } ==>
            [ SetzeStelleWeiter(n) ] Stelle(n) = 0;
    interaction
      variables n:nat;
      Zähle >> SetzeStelleWeiter(1);
      { Stelle(n) = 9 } ==>
            SetzeStelleWeiter(n) >> SetzeStelleWeiter(n+1);
end object DezimalZähler
```

Diese Beispiel enthält einige Besonderheiten, die in der Modellierung einer „Bankenwelt“ schwer zu demonstrieren sind, die aber für die Ausführung von Ereignissen bzw. Spezifikationen generell von Interesse sind. Zunächst wird eine unendliche Zahl von Attributen spezifiziert. Diese Attribute werden mit der Geburt des Objektes auf '0' gesetzt (erste Auswertungsregel). Eine Ausführung dieser Spezifikation ist nicht direkt möglich. Vielmehr muß die Auswertung verzögert werden, bis das Attribut das erste Mal gelesen wird. Dieses Problem soll hier aber nicht weiter diskutiert werden.

Das Ereignis `Zähle` ruft das Ereignis `SetzeStelleWeiter(1)` auf, welches die niedrigstwertige Stelle des Zählers hochzählt. Dieses Ereignis ruft nun seinerseits eine nicht vorher bekannte Zahl von `SetzeStelleWeiter`-Ereignissen für höherwertige Stellen auf, da möglicherweise Überträge auftreten. Diese Aufrufkette wird mit einem bedingten Ereignisaufruf spezifiziert.

Bei der Ausführung von Spezifikationen ist es aufgrund dessen nötig, Ereignisaufrufe zu analysieren, bevor sie tatsächlich auftreten können. Dabei ergibt sich das prinzipielle Problem, daß die Berechnung der Menge der durch ein Ereignis aufgerufenen Ereignisse sowie der Menge der veränderten Attribute wegen Vorbedingungen erster Ordnung nicht allgemein entscheidbar ist. Hier muß gegebenenfalls die Ausdrucksmächtigkeit der Sprache eingeschränkt werden oder eine Analyse zur Laufzeit erfolgen. Letzterer Weg scheint uns für TROLL angemessen, da die Mächtigkeit der Sprache nach Möglichkeit nicht eingeschränkt werden soll.

Wir wollen nun einen Ansatz eines Modells zur Ausführung von Ereignissen kurz erläutern. Das Ausgangsproblem besteht darin, ein gegebenes Ereignis e auszuführen. Dazu sind—zunächst konzeptionell betrachtet—folgende Schritte durchzuführen:

1. Konstruiere die *Menge* der von e aufgerufenen Ereignisse E;

2. Prüfe, ob sich beim Eintreten aller Ereignisse in E Konflikte bezüglich der Attributauswertung ergeben;

3. Prüfe, ob alle Vorbedingungen für Ereignisse in E im augenblicklichen Zustand erfüllt sind;

4. Falls dies der Fall ist, werte die Auswertungsregeln aus und bestimme die neuen Objektzustände.

5. Prüfe, ob alle Integritätsbedingungen von den neuen Objektzuständen eingehalten werden. Ist dies der Fall, ist der Zustandsübergang *gültig*, andernfalls wird er *abgewiesen.*

Schritt 1 terminiert möglicherweise nicht (wie schon oben angedeutet). Eine Verbesserung ist hier aus offensichtlichen Gründen nicht möglich ohne die erwünschten Eigenschaften des Ereignisaufrufs zu verletzen. Die Konstruktion einer *Menge* sichert die im (Bank-)Beispiel erwähnte Abwesenheit von Duplikaten (siehe dort `countBankTransaction`). Der Aufbau der Menge kann zur Laufzeit durchgeführt werden. Dazu werden sukzessive aufgerufene Ereignisse zur Menge hinzugefügt. Das Terminierungsproblem wird damit natürlich nicht gelöst.

Schritt 2 bedeutet, daß Attribute nicht durch mehr als ein Ereignis einer Aufrufkette verändert werden dürfen. Dazu wird zu jedem Ereignis x in E die Menge A_x der durch x veränderten Attribute bestimmt. Für alle Ereignisse $x_1, x_2 \in E$ müssen nun die Mengen A_{x_1} und A_{x_2} disjunkt sein. Bei Konflikten, die auch als Fehler in der Modellierung angesehen werden können, kann die Aufrufkette nicht stattfinden. Auch die Menge der veränderten Attribute kann zur Laufzeit bestimmt werden. Erkannte Konflikte führen dann zum Abbruch der Ereignisausführung.

Der TROLL-Ansatz fordert nun, daß alle durch ein Ereignis e aufgerufenen Ereignisse synchron stattfinden – die Ausführung eines Ereignisses ist also als eine Ausführungseinheit mit den Eigenschaften einer *ACID-Transaktion* zu verstehen.

Der Problembereich Objektinklusion ist bisher nur angedacht worden. Bei der Inklusion sind Instanzen als Ganzes konzeptionell in andere Instanzen (die dann zu komplexen Instanzen werden) eingebettet. Nur so kann von Instanzen auf die Signatur und den augenblicklichen Zustand anderer Instanzen zugegriffen werden. Sämtliche Interaktionen zwischen komplexen Instanzen und deren Teilinstanzen müssen duch Ereignisaufrufe spezifiziert werden.

Zur Implementierung muß hier neben dem Ereignisaufruf auch der Lesezugriff auf Attributwerte von Teilobjekten realisiert werden. Hier sieht ein erster Ansatz vor, daß die Beziehung selbst durch Referenzen realisiert werden kann. Zugriffe auf Attributwerte eingebetteter Instanzen erfordern dann die *Delegation* [Ste87] von Lesezugriffen an die eingebetteten Instanzen mit Hilfe der Inklusionspfade zu den entsprechenden Objektteilen. Das Konzept der Delegation erleichtert dabei die Verwaltung *gemeinsamer Teilobjekte*, wie sie in der TROLL-Kernsprache vorgesehen sind.

4.3 Animationsumgebung

Da die operationalen Eigenschaften der Kernprache TROLL einen direkten Zusammenhang mit objektorientierten Programmiersprachen erkennen lassen, arbeiten wir gegenwärtig an der Umsetzung von Konstrukten der Kernsprache in die objektorientierte Programmiersprache *Sather* [Omo91], die auf Eiffel [Mey88] basiert. Die so erhaltenen Sather-Programme sollen dann in einer *Laufzeitumgebung* zur *verteilten* Animation von Spezifikationen ablauffähig sein. Die Laufzeitumgebung unterstützt die Verteilung von Zustandsinformationen, die transparente Kommunikation und die Objektverwaltung.

Die Animation hat zur Aufgabe, ein erstelltes Modell zu *validieren.* Die Idee ist hier, daß der Entwerfer durch das Anstoßen von Ereignissen und die Beobachtung der dadurch induzierten Aktionen und betroffenen Objekte einen Anhaltspunkt für die Korrektheit bzw. Fehlerhaftigkeit seiner Modellierung erhält.

Hier sind zwei Bemerkungen angebracht. Zum einen ist für eine „realistische“ Animation eine Testdatenmenge notwendig. Die Konstrukton dieser Testdaten wird von uns zunächst noch zurückgestellt, da dieses Problem ein ganz eigenes Forschungsgebiet darstellt. Ansätze zur Konstruktion von Testdaten wurden, im Rahmen eines Projektes zur Modellierung mit Hilfe eines erweiterten ER-Modells, in einer Diplomarbeit untersucht [Zam92]. Zum anderen darf dieses Vorgehen nicht zu der Annahme verleiten, ein solchermaßen getesteter Entwurf sei fehlerfrei. Formale Methoden zur Verifikation von Spezifikationen stoßen jedoch schnell an ihre Grenzen, wenn Sprachen wie TROLL oder sogar nur der TROLL-Kern betrachtet werden. Zur „Verifikation“ gegen die *informellen* Anforderungen sind sie außerdem ungeeignet.

Der prinzipielle Aufbau des Animationssystems ist in Abbildung 1 skizziert. Die Objektverwaltung wird durch sog. *Objektmanager* realisiert. Jeder Knoten des verteilten Systems enthält einen Objektmanager. Diese führen die Kommunikation zwischen Objekten auch über Maschinengrenzen hinaus durch. Teile des oben skizzierten Ausführungsmodells, wie die Konstruktion der Aufrufmengen und die Konflikterkennung, sind in den Objekten selbst realisiert. Zur Zustandsspeicherung wird auf jedem Knoten eine Datenbank (ggf. auch eine Dateisimulation) eingesetzt. Die Datenbank

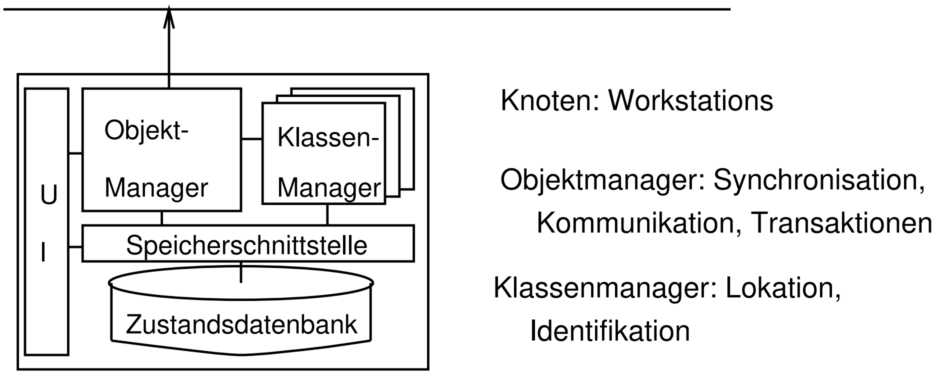

Abbildung 1: Knoten eines verteilten Animationssytems

ist durch eine einheitliche *Speicherschnittstelle* eingekapselt, um möglichst einfach verschiedene Datenhaltungskomponenten integrieren zu können. Die Klassenmanager verwalten Informationen über die gegenwärtigen Klassenpopulationen sowie in einer späteren Ausbaustufe des Animationssystems möglicherweise Informationen über die Lokation und Migration von Objekten. Eine Modellierung der Klassenmanager in TROLL selbst ist geplant.

Wir wollen nun noch kurz darstellen, wie die Ausführung von Ereignissen in der oben skizzierten Umgebung implementiert werden soll: Die Ereignisaufrufe werden zwischen den Objektmanagern derjenigen Knoten weitergereicht, auf denen die angesprochenen Instanzen lokalisiert sind. Damit ergibt sich ein abstrakter Aufrufbaum. In den Knoten dieses Aufrufbaumes—und damit in den Objekten selbst—wird die Konflikterkennung ausgeführt.

Dazu wird in den Knoten des Aufrufbaumes die bisher konstruierte Teilmenge der aufgerufenen Ereignisse und die Menge der veränderten Attribute benötigt. Durch die Verteilung und damit echte Parallelität der Ereignisausführung ist die Weiterreichung dieser Mengen in beide Richtungen des Baumes notwendig. Der Aufwand ist in diesem Fall der Preis für die Ausdrucksfähigkeit des Ereignisaufrufs.

Die Ausführung eines Ereignisses muß weiterhin durch eine (verteilte) ACID-Transaktion geklammert sein. Eine solche Transaktion muß nach einem *2-Phase-Commit-Protokoll* ablaufen und es müssen die angesprochenen Objektinformationen in den Zustandsspeichern durch Sperren gegen nebenläufige Veränderungen geschützt werden. Zur Vermeidung von Verklemmungen und zum Erreichen hoher Parallelität müssen die Objektmanager auf der Implementierungsebene asynchron miteinander kommunizieren.

5 Ausblick

Wir sind uns der Einschränkungen unseres ersten Ausführungsmodells bewußt. Deshalb werden wir in der Zukunft an der Definition eines verfeinerten Modells arbeiten. Schwerpunkt dabei wird die Integration von Subtransaktionen als Verfeinerung von atomaren Ereignissen sein.

Ein weiterer Arbeitsschwerpunkt in der Zukunft wird die Definition von deklarativen Sprachkonstrukten in TROLL durch Konstruktionen in der Kernsprache sein. Großer Wert wird dabei auf die Korrektheit solcher Darstellungen gelegt werden, d.h. mit Hilfe der zugrundeliegenden mathematischen Strukturen und der Logiken zur Spezifikation soll die Äquivalenz korrespondierender Konstruktionen gezeigt werden.

Zur *Verfeinerung* von Spezifikationen muß das Konzept einer abstrakten *Transaktion* als atomare Sequenz von Ereignissen in den TROLL-Ansatz integriert werden. Nur dann wird es möglich, Ereignisse einer höheren Abstraktionsebene durch Abläufe auf einer konkreteren Ebene zur realisieren. Die Schwierigkeit dabei ist die Atomizität von Ereignissen selbst: Es muß bei gleichzeitiger Existenz von atomaren Ereignissen und Transaktionen mit verschiedenen Granularitäten einer impliziten Zeit gearbeitet werden, d.h. ein Ereignis muß mit ganzen Folgen von Ereignissen synchronisiert werden.

Auf der Seite der verteilten Implementierung müssen noch Probleme, die durch die Verteilung auftreten, gelöst werden. So wird die verteilte Überprüfung von Aufrufketten auf Konflikte bezüglich der Auswertungsregeln als problematisch eingestuft. Ebenso ist das Problem der Erkennung und Beseitigung von Deadlocks, hervorgerufen durch Sperren auf Objekten, zu lösen. Auch die verteilte Ausführung von Abläufen ist ein eigenständiges Forschungsgebiet, das z.B. von der Gruppe um Prof. Reuter bearbeitet wird [WR90]. Wir planen die Analyse und den möglichen Einsatz des von dieser Gruppe realisierten APRICOT-Systems [RSW92], welches das Konzept der *ConTracts* realisiert.

6 Zusammenfassung

In diesem Papier haben wir die Ergebnisse der ersten, Anfang 1991 begonnenen Phase des Projekts „Formale Spezifikation und korrekte Implementierung von objektorientierten Informationssystemen" im Schwerpunktprogramm „Objektbanken für Experten" vorgestellt. Die Schwerpunkte der Projektarbeit lagen in der Definition der objektorientierten Sprache TROLL zur konzeptionellen Modellierung von dynamischen Informationssystemen und in der Erarbeitung von Grundlagen zur Ausführung von Spezifikationen.

Wir haben die Konzepte des unserem Ansatz zugrundeliegenden Objektmodells kurz dargelegt. Objekte sind danach beobachtbare Prozesse, deren Zustand durch Attribute beschrieben ist. Der Zustand ändert sich beim Eintreten von Ereignissen. Ein Objektmodell besteht aus einem Prozeß und einer Beobachtungsstruktur, die die Attributbelegung nach einer endlichen Folge von Ereignissen bestimmt. Der Ansatz erlaubt den gleichzeitigen Eintritt von Ereignissen.

Auf diesen Konzepten basiert die Spezifikationssprache TROLL zur konzeptionellen Modellierung von Informationssystemen. Wir haben die wichtigsten Sprachmittel von TROLL kurz vorgestellt: Objektbeschreibungen (*templates*) als Beschreibungen für prototypische Instanzen, Klassen, zusammengesetzte Objekte und Beziehungen (*relationships*).

Unser Ansatz zur Ausführung von Spezifikationen beinhaltet die Transformation von deklarativen Sprachkonstrukten in operationalisierbare und einen ersten Ansatz eines Ausführungsmodells. Wir haben die Problembereiche Objektinklusion und Ausführung von Ereignissen und Ereignisaufrufen diskutiert und erste Lösungsansätze vorgestellt, die in einer Umgebung zur verteilten Animation von Spezifikationen implementiert werden sollen.

Danksagung

Die Entwicklung der formalen Grundlagen von Objektmodellen und darauf aufbauender Sprachen hatte ihren Ursprung im IS-CORE-Projekt. Wesentlich beteiligt waren dabei u.a. Amílcar Sernadas sowie Cristina Sernadas und Felix Costa. Cristina Sernadas hat wesentlich zur Entwicklung der Sprache TROLL beigetragen.

Für die Arbeiten am TROLL-Parser und dem entstehenden Animationssystem gebührt unser Dank Jan Kusch, Urs Thuermann, Axel Stein, und Rüdiger Stöcker. Wertvolle Anregungen haben die Diplomarbeiten von Olaf Faller, Axel Nölke, Scarlet Schwiderski und Michael Thulke gebracht.

Zu früheren Versionen des vorliegenden Beitrags haben wir konstruktive Kommentare von Helmut Wächter und Friedemann Schwenkreis sowie den Herausgebern dieses Bandes bekommen.

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.

[BB92] Breutmann, B.; Burkhardt, R.: *Objektorientierte Systeme. Grundlagen – Werkzeuge – Einsatz.* Carl Hanser Verlag, München, 1992.

[Boo90] Booch, G.: *Object-Oriented Design.* Benjamin/Cummings, Menlo Park, CA, 1990.

[CF92] de Champeaux, D.; Faure, P.: A Comparative Study of Object-Oriented Analysis Methods. *Journal of Object-Oriented Programming*, März/April 1992, S. 21–33.

[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.

[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*, Noordwijkerhout (NL), 1990. LNCS 489, Springer, Berlin, S. 203–228.

[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.

[ESS92] Ehrich, H.-D.; Saake, G.; Sernadas, A.: Concepts of Object-Orientation. In: *Proc. 2. Workshop "Informationssysteme und Künstliche Intelligenz: Modellierung", Ulm.* Springer IFB 303, 1992, S. 1–19.

[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.

[Gri82] Griethuysen, J.J. van (Hrsg.): Concepts and Terminology for the Conceptual Schema and the Information Base. Report N695, ISO/TC97/SC5, 1982.

[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. Genf (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.

[Hoa85] Hoare, C.A.R.: *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, 1985.

[HR92] Hagelstein, J.; Roelants, D.: Reconciling Operational and Declarative Specifications. In: *Proc. 4th Conf. on Advanced Information Systems Engineering CAISE'92*, Manchester (UK), 1992. Springer-Verlag, Berlin, 1992.

[Jac83] Jackson, M. A.: *System Development*. Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[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, erscheint 1992.

[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.

[KL89] Kim, W.; Lochovsky, F. H. (Hrsg.): *Object-Oriented Concepts, Databases, and Applications*. ACM Press/Addison-Wesley, New York, NY/Reading, MA, 1989.

[Lip89] Lipeck, U. W.: *Zur dynamischen Integrität von Datenbanken: Grundlagen der Spezifikation und Überwachung*. Informatik-Fachbericht 209. Springer, Berlin, 1989.

[Mey88] Meyer, B.: *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Mil90] Milner, R.: Operational and Algebraic Semantics of Concurrent Processes. In: Leeuwen, J. van (Hrsg.): *Formal Models and Semantics*. Elsevier Science Publishers B.V., 1990, S. 1201–1242.

[MP92] Manna, Z.; Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems. Vol. 1: Specification*. Springer-Verlag, New York, 1992.

[Nie89] Nierstrasz, O.M.: A Survey of Object-Oriented Concepts. In: Kim, W.; Lochovsky, F. (Hrsg.): *Object-Oriented Concepts, Databases and Applications*. ACM Press and Addison-Wesley, 1989, S. 3–21.

[Omo91] Omohundro, S. M.: The Sather Language, Bericht des International Computer Science Institute (ICSI), Berkeley, Califonia, 1991.

[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.

[RSW92] Reuter, A.; Schwenkreis, F.; Wächter, H.: Zuverlässige Abwicklung großer verteilter Anwendungen mit ConTracts. In diesem Band, 1992.

[Saa88] Saake, G.: *Spezifikation, Semantik und Überwachung von Objektlebensläufen in Datenbanken*. Dissertation, Technische Universität Braunschweig, 1988.

[Saa92] Saake, G.: Objektorientierte Modellierung von Informationssystemen. Vorlesungsskript, TU Braunschweig, 1992.

[Sch92] Schwiderski, S.: Realisation von Objekten in einem Relationalen Datenbanksystem. Diplomarbeit, TU Braunschweig, 1992.

[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.

[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.

[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.

[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.

[SS92] Schwiderski, S.; Saake, G.: Monitoring Temporal Permissions using Partially Evaluated Transition Graphs. In: Lipeck, U.; Thalheim, B. (Hrsg.): *Proc. 4th International Workshop on Modelling Database Dynamics, Volkse, Oct. 1992*, erscheint 1992.

[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.

[Ste87] Stein, L.A.: Delegation is Inheritance. *OOPSLA'87 Proceedings, ACM SIGPLAN Notices (Special Issue)*, Band 22, Nr. 12, 1987, S. 138–146.

[Ste92] Stein, A.: Implementierung eines Parsers für eine objektorientierte Spezifikationssprache für Informationssysteme. Diplomarbeit, TU Braunschweig, erscheint 1992.

[Stö92] Stöcker, R.: Kontextsensitive Analyse von TROLL Spezifikationen. Diplomarbeit, TU Braunschweig, erscheint 1992.

[Weg90] Wegner, P.: Concepts and Paradigms of Object-Oriented Programming. *ACM SIGPLAN OOP Messenger*, Band 1, Nr. 1, 1990, S. 7–87.

[Wir90] Wirsing, M.: Algebraic Specification. In: Van Leeuwen, J. (Hrsg.): *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics.* Elsevier Sci. Publ. B.V., Amsterdam, 1990, S. 675–788.

[WR90] Wächter, H.; Reuter, A.: Grundkonzepte und Realisierungsstrategien des ConTract-Modells. *Informatik Forschung und Entwicklung*, Band 5, 1990, S. 202–212.

[Zam92] Zamperoni, A.: Ein konzeptioneller Rahmen zur pragmatischen Modellgenerierung für EER-Schemata. Diplomarbeit, TU Braunschweig, 1992.