Spezifikation von Informationssystemen als Objektsysteme: Das TROLL-Projekt[*,†]

T. Hartmann
R. Jungclaus
G. Saake

Informatik, Abteilung Datenbanken
Technische Universität Braunschweig

W–3300 Braunschweig

E-mail: {hartmann|jungclau|saake}@idb.cs.tu-bs.de

Zusammenfassung

Der Einsatzbereich von Software-Systemen ist gekennzeichnet durch immer komplexer werdende Anforderungen und eine Orientierung von funktions-orientierten Systemen zu daten-orientierten Systemen. Klassische Entwurfsmethoden für datenintensive Anwendungen beschreiben strukturelle resp. dynamische Eigenschaften des modellierten Weltausschnittes mit unterschiedlichen, wenig intergrierten Formalismen. Objektorientierte Ansätze integrieren dagegen strukturelle und dynamische Aspekte in Objekten, sind aber oft nicht formal beschrieben. In diesem Artikel stellen wir die Sprache TROLL vor, die auf einem formalen Objektkonzept aufbaut. TROLL dient der Spezifikation und Modellierung von Software- und Informationssytemen. Ein TROLL Modell stellt dabei deklarativ die Eigenschaften dar, die das System erfüllen soll. Eine Übersicht zu verwandten Ansätzen wird gegeben und auf gegenwärtige und zukünftige Arbeiten des Projektes verwiesen.

1 Einleitung

Der Einsatzbereich von Software-Systemen ist gekennzeichnet durch immer komplexer werdende Anforderungen und eine Orientierung von funktions-orientierten Sy-

[*]Die Arbeit wurde teilweise gefördert von der EG in der ESPRIT WG IS-CORE II (Information Systems – COrrectness and REusability). Die Arbeit von Thorsten Hartmann und Ralf Jungclaus wird von der Deutschen Forschungsgemeinschaft unter Sa 465/1-2 gefördert. Projektleiter sind G. Saake und H.-D. Ehrich.

[†]In: EMISA Forum, Mitteilungen der GI Fachgruppe 2.5.2, Nr. 1, 1993, S. 2–18

stemen zu daten-orientierten Systemen [CHJ86, CY89]. Die Probleme im Software-Lebenszyklus treten so immer deutlicher zutage. In der Wartungsphase werden Fehler aufgedeckt und behoben, die sowohl bei der Konzeption als auch bei der Implementierung des Systems entstanden sind. Konzeptionelle Fehler sind dabei meist sehr schwer zu entdecken und nur unter hohen Kosten zu beheben [Par90]. In der Wartungsphase werden auch Erweiterungs- und Änderungswünsche des Anwenders umgesetzt und das System im günstigen Fall geändert, im schlechtesten Fall jedoch neu entwickelt.

Software-Entwicklung ist in vielen Fällen heute noch mehr Handwerkskunst als ingenieurmäßiger Entwicklungsprozeß, wobei diese Phänomen besonders in den frühen Phasen der Systementwicklung auftritt. Als Antwort auf diese Probleme wurden in den letzten Jahren *formale Methoden* [CHJ86, Win90] der Systementwicklung vorgeschlagen.

Formale Methoden sind Techniken zur *Konstruktion formaler Modelle*, die analysiert werden können. Sie ermöglichen die Verifikation bzw. Validierung der Implementierung gegen die Spezifikation und tragen somit entscheidend zur Verminderung von Fehlern bei der Implementierung bei. Weiterhin ermöglichen sie die Aufdeckung von unpräzisen Anforderungen, Unvollständigkeiten und Widersprüchen in der Systemspezifikation und helfen somit bei der Vermeidung vieler *konzeptioneller* Fehler.

In der Vergangenheit wurden formale Methoden zur Systembeschreibung mit unterschiedlichem Anwendungsbereichen vorgeschlagen:

- Bei der *algebraischen Spezifikation abstrakter Datentypen* werden Klassen von Datenstrukturen durch die Angabe von Operationen auf diesen Daten beschrieben [EM85, VHL89, EGL89]. Die Bedeutung der Operationen wird durch Gleichungen spezifiziert.

- In *modellbasierten Spezifikationsansätzen* wie VDM [BJ82, Jon86] und Z [Spi89] wird aus mathematischen Primitiven ein mathematisches Modell des Systems konstruiert.

- Mehrere Ansätze befassen sich mit der *Spezifikation* des Verhaltens *nebenläufiger Systeme*, unter ihnen CSP [Hoa85], CCS [Mil89] und ACP [BK86]. Ein System wird dargestellt durch eine Menge kommunizierender nebenläufiger oder paralleler Prozesse über einer Menge von abstrakten Ereignissen. Das Verhalten des Systems kann aus der Spezifikation der einzelnen Prozesse abgeleitet werden.

Auch bei der Anwendung etablierter formaler Methoden des Systementwurfs gibt es jedoch Probleme. Zum einen müssen auf einer Abstraktionsebene zur Darstellung von Struktur und Verhalten verschiedene Formalismen benutzt werden (z.B. abstrakte Datentypen für Daten, Prozeßmodelle für Funktionalität). Zum anderen werden oft auf verschiedenen Abstraktionsebenen verschiedene Beschreibungsobjekte betrachtet (z.B. Funktionen und Abläufe in der Anforderungsanalyse, Datenstrukturen im Systementwurf).

Probleme bei der Anwendung formaler Methoden wirft weiterhin die Anforderung auf, daß Systemspezifikationen sowohl von Anwendern als auch von Entwicklern verstanden werden müssen, da die mathematischen Grundlagen nicht vorausgesetzt werden können. Viele konzeptionelle Fehler würden nicht entstehen, wenn Benutzer bereits die Spezifikation eines Systems beurteilten und feststellten, ob sie die Anforderungen erfüllt.

Mit den genannten Problemen befaßt sich der *objektorientierte Systementwurf* [MK90]. Objekte sind eine Einheit von lokaler Struktur und lokalem Verhalten und heben daher die Grenze zwischen Daten und Prozessen auf. Die Einkapselung unterstützt Daten- und Prozeßabstraktion und unterstützt damit die frühen Phasen des Systementwurfs.

Objekte, die in der Anforderungsanalyse identifiziert wurden, werden im Systementwurf und in der Implementierung verfeinert und direkt implementiert. Damit wird der Übergang zwischen den einzelnen Phasen des Systementwurfs geglättet. Objektorientierte Systementwicklung unterstützt die Wiederverwendung von Software, die Erweiterbarkeit von Software-Bausteinen und damit die *Konstruktion* von Systemen aus Systembausteinen. Objekte sind weiterhin direkte sprachliche Darstellungen von relevanten Objekten des darzustellenden Weltausschnitts.

Wir wollen im folgenden einen Ansatz vorstellen, der die Vorteile formaler Methoden der Software-Entwicklung mit denen objektorientierter Systementwicklung vereint. Damit wird unserer Meinung nach sowohl die Qualität der zu entwickelnden Systeme als auch die Einbeziehung der Benutzer in den Prozeß der Systementwicklung gefördert. Systemspezifikationen sollen einerseits formal, andererseits aber intuitiv verständlich sein und daher in ihrer Qualität als *Vertrag* zwischen Benutzer und Entwickler verbessert werden.

Wir stellen kurz einen Ansatz zur objektorientierten formalen Spezifikation von Softwaresystemen vor. Grundlegende Konzepte des Objektmodells sind in [SSE87, ES91, ESS92, SJE92, SE91, EGS90, FSMS91] entwickelt worden. Die Sprache TROLL wird in [JSHS91] vorgestellt.

2 Grundkonzepte des objektorientierten Ansatzes

Die Grundkonzepte der objektorientierten Sichtweise halten gegenwärtig Einzug in den Bereich des Entwurfs von Informations- und Softwaresystemen [BB92, dF92]. Die grundlegenden Ideen stammen jedoch ursprünglich aus dem Bereich der Programmiersprachen und Datenbanken [Weg90, ABD$^+$89].

Objekte werden als eingekapselte Einheiten im Sinne einer Menge von nach außen sichtbaren *Operationen* – der Objektschnittstelle – gesehen. Die Operationen arbeiten auf einem nicht sichtbaren, eingekapselten *Zustand*. Man unterscheidet oft noch ändernde von nur lesenden Operationen. Operationen werden durch Versenden von *Nachrichten* zwischen Objekten aufgerufen.

Als grundlegend wird bei objektorientierten Ansätzen die *Namensgebung* angesehen. Objekte besitzen einen eindeutigen *Identifikator*, der zur Unterscheidung von Objekten mit gleichem Zustand herangezogen werden kann. Der Identifikator wird dabei oft als Referenz realisiert.

Als vielen objektorientierten Ansätzen gemeinsames Abstraktionsmittel werden *Klassen* angesehen. Eine Klasse definiert hierbei eine Menge von gleichartig strukturierten und sich gleichartig verhaltenden Objekten. Während Programmiersprachen in der Regel *Klassen* mit *Typen* identifizieren, wird eine Klasse in der Datenbankwelt oft als *Behälter* für Instanzen eines Typs (*Schema*) angesehen.

Zur Strukturierung, Modularisierung und Wiederverwendung von Softwaredokumenten hat das Konzept der *Vererbung* Bedeutung erlangt. Man geht hierbei von der Beobachtung aus, daß Softwarekomponenten aus einfacheren, d.h. generalisierten Strukturen durch *Spezialisierung* hervorgehen können. Die einzelnen Komponenten des zu erstellenden Systems werden somit *hierarchisch* organisiert. Ein allgemeines Verhalten wird dann duch Vererbung und Anpassung weiter verfeinert. Vererbung ist oft auf Klassenebene angesiedelt.

Im Gegensatz zu objektorientierten Programmiersprachen, die alle in einem Softwaresystem behandelte Einheiten als Objekte ansehen (als prominenten Vertreter siehe z.B. Smalltalk [GR83]), wird in Datenbankansätzen oft zwischen *Objekten* als zeitlich veränderlichen Einheiten und *Daten* als statischen Einheiten unterschieden. Daher wird in diesem Bereich oft eine Erweiterbarkeit um *benutzerdefinierte Datentypen* gefordert [ABD+89].

Für formale, objektorientierte Ansätze zur Modellierung und Spezifikation von Systemen müssen diese Grundkonzepte nun angepaßt und erweitert werden. Neben der implementierungsnahen, operationalen Sichtweise müssen abstrakte, deklarative Konzepte in ein Objektmodell einfließen.

Das der Sprache TROLL unterliegende Objektmodell setzt einen Schwerpunkt auf die Modellierung des *Verhaltens* und der *Entwicklung* von Objekten über der Zeit. Objekte werden als *Prozesse* betrachtet, deren Zustand beobachtet werden kann. Der beobachtbare Zustand eines Objekts enthält die relevanten Eigenschaften des Objekts und kann beliebig strukturiert sein.

Spezifikationen von Objekten enthalten also Spezifikationen der beobachtbaren Eigenschaften, die Attribute genannt werden und den Attributen in semantischen Datenmodellen entsprechen. Änderungen des Zustands eines Objekts erfolgen ausschließlich durch *objektspezifische Ereignisse*, die Abstraktionen von Operationen auf Objektzuständen sind. Das mögliche Verhalten eines Objekts wird durch eine Menge von erlaubten Folgen von Ereignissen beschrieben und entspricht daher einer Spezifikation eines Prozesses in einer Spezifikationssprache für nebenläufige Prozesse.

Objekte können zusammengesetzt werden zu komplexen Objekten, sie können durch Aufruf von Ereignissen interagieren, sie können Spezialisierungen allgemeinerer Objekte sein u.v.m.

Ein System wird wird durch eine Menge von miteinander kommunizierenden Objekten beschrieben. Dies entspricht weitgehend der Denkweise von BenutzerInnen bei der Beschreibung von Systemen und macht deren Mitarbeit beim Systementwurf ohne Kenntnisse der mathematischen Grundlagen von formalen Spezifikationssprachen möglich. Spezifikationen in Troll sind jedoch formale Spezifikationen und machen den Einsatz formaler Verifikations-, Validierungs- und Implementierungsmethoden möglich.

3 Semantische Modellbildung für Objekte

Wie im vorigen Abschnitt angedeutet, sind Objekte im Troll unterliegenden Objektmodell als (beobachtbare) *Prozesse* anzusehen. Ein Objekt wird daher beschrieben als mögliche Abfolgen von *Ereignissen* (Prozesse) und der aus Anfangsstücken dieser Sequenzen abgeleiteten Beobachtungen (Attribut-Wert-Paare). Um Objektkommunikation bzw. in diesem Fall Prozeßsynchronisation beschreiben zu können, gehen wir nicht nur von Ereignissequenzen sondern von Sequenzen *synchron eintretender Ereignisse* (s.g. Schnappschüsse) aus.

Formal ist ein Objektmodell definiert als ein Paar bestehend aus einem Prozeß und einer Beobachtungsstruktur $ob = (P, V)$. Ein Prozeß ist dabei spezifiziert durch ein Alphabet von Ereignissen X und einer Menge Λ von *möglichen* Sequenzen über Teilmengen von Ereignissen $P = (X, \Lambda)$, die leere Sequenz eingeschlossen.

Die Menge der Ereignisse wird weiter unterteilt in *Geburts-*, *Todes-* und *Update*ereignisse. Für jedes $\lambda \in \Lambda$ muß dabei mindestens gelten, daß die Sequenz mit einem Schnappschuß beginnt, der ein Geburtsereignis enhält, daß endliche Sequenzen mit einem Schnappschuß enden, der ein Todesereignis enthält und das unendliche Sequenzen keine Schnappschüsse mit Todesereignissen enthalten. Der *Zustand* eines Objektes kann als die Folge der Schnappschüsse seit der Geburt angesehen werden.

Die zu einem Objektmodell gehörende Beobachtungsstruktur ist definiert als eine Menge von *Attributsymbolen* A und einer Abbildung α, die endlichen Sequenzen von Schnappschüssen (Zuständen) eine Menge von Attribut-Wert-Paaren zuordnet. Jedem Attribut ist dabei eine Datensorte zugeordnet, d.h., Attribute sind *typisiert*. Eine ausführliche Darstellung dieses Modelles ist in [JSS91, HJS92] zu finden.

Mit der hier kurz angerissenen Modellbildung werden zunächst nur einzelne Objekte beschrieben. Um von einzelnen Objekten zu *Klassen* von Objekten mit gleichartiger Struktur und gleichartigem Verhalten zu kommen, ist die Einführung eines *Klassentyps* notwendig. Semantisch wird ein Klassentyp durch eine Identifikationssorte und ein Objektmodell beschrieben $ct = (ID, ob)$. ID ist definiert als Trägermenge einer Sorte eines abstrakten Datentyps dessen Instanzen Objektmodellen zugeordnet werden. Die Zuordnung ist dabei statisch.

Eine Klasse ist nunmehr definert als eine Variable des Klassentyps. Mit anderen Worten, eine Klasse hat selbst einen Zustand, der durch die *Aggregation* der zugehörigen Objekte definiert ist. Zu diesem Zustand gehören auch Objekte mit (bisher noch) leerer Schnapschußsequenz („nicht existierende Objekte“). Eine genauere Darstellung von Klassentypen, Klassen und Objektinstanzen findet sich in

[HJSE92].

Als weitere grundlegende Konzepte wurden im letzten Abschnitt Objektzusammensetzungen und Objektkommunikation genannt. In diesem Zusammenhang ist also hier eine Beschreibung von *Prozeßsynchronisation* gefordert. Diese Forderung wird durch die Beschreibung von *sicheren Objekteinbettungen* genüge getan. Eine Objektinstanz b ist dabei in eine Objektinstanz a sicher eingebettet, wenn

1. die Signatur von b in der Signatur von a enthalten ist (bis auf Umbenennungen),

2. Lebensläufe von b *korrekte Teillebensläufe* in a sind,

3. Attribute von b nur durch lokale Ereignisse von b verändert werden.

Punkt 2 bedeutet dabei, daß das mögliche Verhalten der eingebetteten Instanz allenfalls weiter eingeschränkt werden darf. Punkt 3 bedeutet, daß Beobachtungen der eingebetteten Instanz nur von der lokalen Schnapschußsequenz abhängen. Beide Bedingungen zusammen rechtfertigen die Bezeichnung *sichere Einbettung* im Sinne einer Einkapselung lokal definierter Eigenschaften.

Kommunikation zwischen Objekten basiert auf den Konzepten *Ereignisschnappschuß* (synchron eintretende Menge von Ereignissen) und *Einbettung*. Das Verhalten der durch Objekteinbettung miteinander in Beziehung gesetzten Instanzen wird dazu geeignet *synchronisiert*. Dazu führen wir den *Ereignisaufruf* als Kommunikationsprimitiv ein. Ruft ein Ereignis x_1 ein Ereignis x_2 auf (notiert als $x_1 \gg x_2$), 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. Details zur Einbettung und Kommunikation sind u.a. in [HJ92] zu finden.

Wir werden nun im nächsten Abschnitt auf die Sprache TROLL eingehen, die auf dem skizzierten Objektmodell basiert. TROLL integriert dabei verschiedene Sprachen, die zur Beschreibung der angesprochenen Modellstrukturen geeignet sind. Das sind im einzelnen eine mehrsortige Logik erster Ordnung, eine Lineare Temporale Logik erster Ordnung (vergangenheits- wie zukunftsgerichtet) [Saa88, SL89] sowie eine auf CSP [Hoa85] basierende Sprache zur Spezifikation von Prozessen.

4 TROLL — Eine Sprache zur Objektspezifikation

Die kleinste Einheit des Entwurfes in TROLL ist die *Template*-Beschreibung. Templates werden im Zusammenhang mit einem geeigneten Namensmechanismus zur Beschreibung von Klassentypen genutzt. Aus dem Klassentyp leiten sich dann Objektinstanzen sowie die Klassen als *Behälter* für diese Objektinstanzen ab. Ein Template kann damit als ein *Prototyp* für Objekte gesehen werden.

In der Templatebeschreibung wird zunächst die Objekt-Signatur (**attributes**, **events**) sowie die Einbettung von Unterobjekten (**including**, hier nicht gezeigt) festgelegt. TROLL basiert auf einer Reihe von Logikdialekten. Attribut- und Ereignissymbole des Templates (eingeschlossen die Signatur von Teilobjekten) können daher in den einzelnen Abschnitten der Templatespezifikation zur Konstruktion von Formeln genutzt werden, die die Eigenschaften der spezifizierten Objekte festlegen. Mit anderen Worten, TROLL-Templatebeschreibungen legen die im letzten Abschnitt angerissenen Objektmodelle fest.

Als Beispiel werden wir ein Template für ein Kontenobjekt darstellen, an dem wir dann die einzelnen Spezifikationsteile näher erklären können.

```
template account
  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|;
    { Balance > 0 } ==> [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 template account;
```

In der **data types** Sektion werden die für dieses Template benötigten Datentypen bekannt gemacht. Wir gehen davon aus, daß Datentypen in einem geeigneten

Formalismus außerhalb TROLLs definiert wurden und hier nur ihre Signatur importiert wird. Eine spezieller Datentyp ist hier der Typ `|BankCustomer|`. Er wird automatisch bei der Spezifikation einer Klasse generiert (s.u.) und zur Identifikation von Objektinstanzen dieser Klasse genutzt.

In der **attributes** resp. **events** Sektion wird die Signatur eingeführt. Attribute beschreiben sichtbare Objekteigenschaften, Ereignisse beschreiben zustandsmanipulierende Objekteigenschaften. Ereignisparameter, wie `Amount:money` für das Ereignis `withdraw`, stellen eine Möglichkeit dar, Mengen von Ereignisse zu spezifizieren. Aktuelle Parameter dienen später dazu Zustandsübergänge näher zu beschreiben oder Daten bei Objektinteraktionen auszutauschen. Für letzteres kann dazu der formale Parameter als Eingabe- (**in**) bzw. Ausgabeparameter (**out**) klassifiziert werden. Aufbauend auf dieser Signatur kann nun die eigentliche Spezifikation erfolgen.

Im **constraints** Abschnitt werden *zulässige Attributentwicklungen über die Zeit* spezifiziert. Hier angegebene Formeln sind Formeln einer temporalen, in die Zukunft gerichteten Logik [Saa91]. Im Beispiel wird festgelegt, daß im ersten Objektzustand (**initially**) die Attribute `CreditLimit` und `Balance` den Wert 0 haben müssen. Die zweite Regel besagt, daß das Konto nach einer Überziehung irgendwann wieder ausgeglichen sein muß. Zu bemerken ist, daß die Regeln deklarativ eine Eigenschaft beschreiben, die eine spätere Realisierung des Objektes erfüllen muß. Man kann aus diesen Eigenschaften z.B. *ableiten*, daß ein Konto nach einer Überziehung und vor dem nächsten Ausgleich nicht aufgelöst werden darf.

Im **valuation** Abschnitt werden die Auswirkungen von Ereignissen auf die zu beobachtenden Attribute dargestellt. Im Beispiel wird hier beschrieben, wie z.B. das Ereignis `deposit`, instantiiert mit einem Parameter `p` den Kontostand erhöht. Auswertungsregeln dieser Form können *zustandsabhängig* spezifiziert werden, d.h., sie haben nur eine Auswirkung auf die Beobachtung wenn eine bestimmte Bedingung erfüllt ist. In diesem Falle wird ein neuer Überziehungskredit nur eingetragen, wenn das Konto nicht überzogen ist. Das bedeutet *nicht*, daß das Ereignis `new_credit_limit` nicht stattfinden darf. Die Spezifikation solcher Eigenschaften folgt im nächsten Abschnitt einer Templatespezifikation.

Das *Verhalten* von Objekten bezüglich der erlaubten Ereignissequenzen (Prozesse, s.o.) wird im **behavior** Abschnitt spezifiziert. Wir unterscheiden hier zwischen *Erlaubnis* für Ereignisse (**permissions**), und der Forderung des *Enthaltenseins* in einer Sequenz (**obligations**). Im Beispiel wurde nur von der ersten Möglichkeit Gebrauch gemacht:

```
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);
```

Die erste Regel besagt, daß das Konto nur geschlossen werden darf, wenn es einen Kontostand von 0 aufweist; eine genauere Bedingung als im **constraints** Abschnitt (dort implizit!) dargestellt wurde. Die zweite Bedingung ist in einer temporalen, in die Vergangenheit gerichteten Logik formuliert [MP92, SS92, Sch92]. Das dort verwendete **after**-Prädikat ist wahr im Zustand nach Eintreten seines Parameterereignisses. Die Regel besagt, daß keine zwei Ereignisse `update_request` aufeinander folgen dürfen, sondern eines der Ereignisse `update_failed`, `withdraw` oder `deposit` dazwischen liegen muß. Eine **obligation** kann hier formuliert werden, die eines dieser drei Ereignisse *erzwingt*. Wir haben der Kürze wegen hier darauf verzichtet (siehe dazu auch [JSHS91]).

Der volle Sprachumfang gibt neben dieser deklarativen Prozeßbeschreibung noch die Möglichkeit der *expliziten* Angabe von Prozessen in ener *CSP* [Hoa85] ähnlichen Notation, die sich für deterministisches Objektverhalten eignet. Außer der Möglichkeit mit **obligations** Ereignisse zu erzwingen, die sich eher auf längerfristige Anforderungen beziehen, bietet TROLL noch die Option *Aktivität* im Sinne von *kurzfristig zu verfolgenden Zielen* (**commitments**) an [JSHS91]).

Um nun nicht nur einzelne Objekte zu beschreiben, muß ein Identifikationsmechanismus eingeführt werden. Für *externe* Objektnamen wird ein Datentyp angegeben deren Werte als Objektnamen benutzt werden. Damit kommen wir zu folgender Einordnung:

$$Klassentyp = TemplateSpezifikation + IdSpezifikation$$

$$ObjektInstanz = TemplateInstanz + IdDatenwert$$

Ein Klassentyp als Beschreibung der Klasse sowie deren Instanzen wird nun wie folgt dargestellt:

```
object class Account
  identification No:nat;
  template account;
end object class Account
```

Mit der Klassentypbeschreibung wird ein *Namensraum* eingeführt, d.h., ein Datentyp `|Account|` als Identifier für Kontenobjekte. Dieser Datentyp wird isomorph zum Datentyp `nat` der natürlichen Zahlen konstruiert. Das Attribut `No:nat` dient somit als *externer* Name für Kontenobjekte und kann innerhalb des Templates wie ein Attribut verwendet werden. Details können wiederum in [JSHS91] gefunden werden.

Mit der Klasse als Abstraktionsmechanismus ist zwar ein erstes Mittel zur Strukturierung von Objektspezifikationen gegeben, es reicht aber im allgemeinen nicht aus, um komplexe Systeme zu beschreiben. Da TROLL Struktur sowie Verhaltensaspekte beschreibt, werden Spezifikationen schnell lang und unübersichtlich. Einen Ausweg bietet hier die Einführung einer *Spezialisierungshierarchie*. Wir werden hier nur die *dynamische* Spezialisierung mit Hilfe von *Rollen* darstellen, da sie das allgemeinere Konzept repräsentiert.

```
object class OverdrawnAccount
  role of Account;
  template
    events
      birth overdraw;
      death normal; ...
    ...
    behavior
      variables n:nat;
      { not always(after(overdraw)) } withdraw(n); ...
      interaction
        variables n:nat;
        { (Balance - n) < 0 } Account.withdraw(n) >> overdraw;
        { (Balance + n) >= 0 } Account.deposit(n) >> normal;
end object class OverdrawnAccount
```

Dieses Beispiel beschreibt ein temporäres Verhalten von `Account`-Objekten. Die Klasse `OverdrawnAccount` *erbt* die Eigenschaften der Klasse `Account`, d.h., Signatur und Verhalten sind in der Spezialisierung sichtbar (*is-a-Beziehung*). Es wird allerdings nicht nur eine syntaktische Verebung durchgeführt sondern auch die konkrete Ausprägung einer Konteninstanz kann benutzt werden. Das bedeutet auch eine Zugriffsmöglichkeit auf die Attributwerte. Das Rollenkonzept läßt sich mit Hilfe der sicheren Einbettung erklären. In diesem Falle wird eine `Account`-Objekt als Teilobjekt einer `OverdrawnAccount`-Instanz angenommen. Daher erklärt sich auch die Spezifikation von *Interaktionen* im letzten Teil des Beispiels.

Die so spezifizierte Klasse schränkt damit das Verhalten von Kontenobjekten weiter ein. In dieser etwas künstlichen Modellierung werden folgende Eigenschaften spezifiziert. Die Rolle wird begonnen, wenn eine Abhebung in den Soll-Bereich führt (erste Kommunikationsregel), und beendet wenn eine Einzahlung wieder in den Haben-Bereich führt. Als einzige weitere Regel ist im **behavior** Abschnitt eine *zusätzliche* Einschränkung des Lebenslaufes beschrieben worden. Man bemerke, daß dadurch die Spezifikation im Sinne der sicheren Objektinklusion nicht verletzt wird.

Troll bietet neben der dynamischen Spezialisierung (Rollen) noch die *statische* Spezialisierung an, bei der die Zugehörigkeit von Instanzen zu spezialisierten Klassen aufgrund *konstanter Objekteigenschaften* zur Geburtszeit festgelegt wird. Die Beschränkung auf statische Eigenschaften ergibt sich aus der Forderung, daß ein Klassenwechsel von Instanzen bei Zustandsübergängen explizit gemacht werden sollte. Generell kann die statische Spezialisierung über ähnliche Aufrufregeln wie im Beispiel mit Hilfe der dynamischen Spezialisierung modelliert werden. Außer der einfachen Verebung ist auch eine Mehrfachvererbung in Troll möglich.

Wie in den letzten Paragraphen schon angedeutet, kann der im Abschnitt 3 angedeutete Einbettungsmechanismus vielseitig verwendet werden. Grundsätzlich dient der Mechanismus jedoch der Darstellung von komplexen Objekten. Da Troll-Spezifikationen Ausschnitte der realen Welt modellieren, und nahezu alle dort vorkommenden Objekte *zusammengesetzt* sind, stellt TROLL Sprachkonstrukte zur Verfügung um zusammengesetzte Objekte (*part-of-Beziehung*) zu modellieren.

Neben einer statischen Variante, die direkt auf dem semantischen Konzept der Einbettung aufbaut, kennt TROLL *dynamische* komplexe Objekte, deren Struktur durch das Eintreten von Ereignissen verändert werden kann. Ferner werden solche Teilobjekte unterschieden, die nur *lokal* verwendbar sind und solche die Teilobjekte in *verschiedenen Objekten* sein können, d.h., nicht nur lokal verwendet werden können [BB84]. Erstere sind in ihrem Lebenslauf von den Elternobjekten abhängig, letztere sind unabhängig. Wir werden hier die *nicht (nur) lokalen* komplexen Objekte vorstellen (*shared subobjects* [HJS92]), da sie das allgemeinere Konzept darstellen.

Als Beispiel wählen wir eine Bank, die einen Manager, Kontenobjekte, und eine Schalterwarteschlange enthält:

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

Komponentenobjekte werden in TROLL im Abschnitt **components** eingeführt. Basierend auf Datenmodellierungskonzepten wie sie z.B. in erweiterten ER-Modellen zu finden sind [EGH$^+$92], können *einzelne* Komponentenobjekte, *Mengen* von Komponentenobjekten und *Listen* von Komponentenobjekten spezifiziert werden. In diesem Beispiel wurde eine Instanz der Klasse `Person`, die Menge aller `Accounts` und die Liste der gegenwärtig wartenden `BankCustomer` (eine Rolle von Person, hier nicht weiter dargestellt) spezifiziert.

Zur Manipulation und Beobachtung dieser Objektstruktur werden mit der Spezifikation von Komponentobjekten implizite Ereignisse und Attribute der Signatur hinzugefügt. Das sind z.B. Ereignisse wie `Accts.INSERT(|Account|)` und Attribute wie `ServiceQueue.LENGTH:nat`. Die so definierten Namen können wie andere

Signaturbestandteile in Formeln verwendet werden. Ein Beispiel dafür ist der Ereignisaufruf im Kommunikations-Abschnitt.

Die Komponentennamen dienen innerhalb des komplexen Objektes als *lokale Klassennamen*. Besonders Komponenten, die mit dem *Set-Konstruktor* eingeführt werden, können orthogonal zu (globalen) Klassennamen benutzt werden. Ihr Gültigkeitsbereich ist allerdings auf das komplexe Objekt beschränkt. Eine Auswahl einer sinnvollen Menge von implizit generierten Zugriffsmöglichkeiten findet sich wiederum in [JSHS91].

Nachdem die für den Weltauschnitt relevanten Objekte bzw. Klassen definiert wurden, müssen diese *Bausteine* geeignet in Verbindung gebracht werden. Bisher wurde ein Kommunikationsmechanismus nur innerhalb von (komplexen) Objekten und für Spezialisierungen beschrieben. Viele Objekte stehen aber nicht in einer engen *part-of* oder *is-a* Beziehung. In der Bankenwelt kann z.B. ein Geldautomat spezifiziert werden, der in Interaktion zu den Kunden steht. Da man hier weder Kunden als Teilobjekte der Geldautomaten noch Geldautomaten als Teilobjekte der Kunden sehen möchte, noch eine Spezialisierung in Frage kommt, werden als weiteres Sprachmittel **relationships** eingeführt. Mit **relationships** können sowohl Integritätsbedingungen als auch Interaktionen *zwischen* relativ autonomen Objekten spezifiziert werden [JHS92].

Eine weitere Notwendigkeit bei der Spezifikation eines ganzen Systems miteinander kommunizierender Objekte ist die Beschreibung von *Schnittstellen*, d.h., im wesentlichen eingeschränkte Sichtbarkeiten für Objektbestandteile [SJ92b, SJ92a]. Aus Platzgründen können wir hier nicht weiter auf diese Sprachmittel eingehen und beschließen damit den kurzen Überblick über die Sprache TROLL.

5 Verwandte Ansätze

Beim Entwurf von Informationssystemen kamen bisher meist Formalismen zum Einsatz, die entweder nur die Beschreibung der strukturellen Aspekte oder nur die Beschreibung der dynamischen Aspekte unterstützten. Diese diametralen Ansätze sind nicht gut verträglich, da sie unterschiedliche semantische Konzepte benutzen.

In letzter Zeit sind verschiedene Ansätze zur objektorientierten Modellierung bekannt geworden. Die Grundidee der Objektorientierung ist schon 25 Jahre alt und wurde im SIMULA-Ansatz [DN66] entwickelt. Simula unterstützte die Beschreibung von Simulationen und betrachtete dazu den zu simulierenden Weltausschnitt als *System von interagierenden Objekten.* Mit der Sprache Smalltalk [GR83] wurde diese Idee auf die Spitze getrieben, indem *alle* Elemente des Systems (auch Datenwerte) als Objekte betrachtet werden.

Objektorientierte Programmierung (OOP) unterstützt die Implementierung von Objekten und bietet daher operationale Sprachmittel an, die zur konzeptionellen Modellierung weniger gut geeignet sind. Das Konzept persistenter Objekte wird in OOP vernachlässigt, jedoch wiederum von objektorientierten Datenmodellen (OODM) aufgegriffen, die im Gegensatz zu OOP die strukturelle Modellierung in den Vordergrund stellen.

Ansätze zur objektorientierten Analyse (OOA) (z.B. [RBP+91, Boo90]) sollen die Entwicklung von Softwaresystemen unterstützen. Die meisten dieser Ansätze sind jedoch informell. Sie benutzen Konzepte, die unpräzise definiert wurden. Dies führt dazu, daß der primäre Zweck (nämlich die *präzise* Darstellung von Anforderungen) nur schlecht erfüllt wird. Die Ansätze fördern jedoch die Verständlichkeit von Entwürfen und sind daher geeignet zur Anforderungsanalyse. Die Anforderungsspezifikation oder das konzeptionelle Modell sollten jedoch präzise sein.

Der in diesem Papier vorgestellte Ansatz integriert semantische Datenmodellierung, algebraische Spezifikationstechniken und Spezifikationstechniken für Prozesse und reaktive Systeme. Sehr verwandte Ansätze sind

- OBLOG [SSE87, SFSE89] und Graphical OBLOG [SRGS91],
- TLOOM [Ara91, Ara92],
- CMSL [Wie90, Wie91],
- Glider [DDRW91, DDR92] und
- Object/Behavior Diagrams [KS91].

Diese Ansätze zeichnen sich durch eine präzise definierte Semantik aus. Sie werden im folgenden kurz charakterisiert.

Die Sprache OBLOG liegt der hier vorgestellten Sprache Troll zugrunde. Die Sprachmittel für Templates ähneln sich weitgehend, jedoch wird in OBLOG der Namensraum explizit in der Objektbeschreibung definiert. Es gibt jedoch noch keine Abstraktionsmechanismen wie Spezialisierung und Rollen – Beziehungen zwischen Spezifikationen (und Instanzen) werden explizit durch Abbildungen definiert. Graphical OBLOG repräsentiert i.W. die Sprachmittel in OBLOG durch graphische Symbole. In verschiedenen Diagrammen werden Namensräume und Beziehungen zwischen Namensräumen, Template-Signaturen, Lebensläufe und Interaktionen spezifiziert.

TLOOM benutzt eine temporale Logik zur Spezifikation des Verhaltens von Objekten. Die Schnittstelle eines Objekts besteht ausschließlich aus Nachrichten, die versandt oder empfangen werden können. Die wesentliche Aufgabe bei der Spezifikation von Systemen in TLOOM besteht in der Definition von erlaubten Sequenzen von Nachrichten durch temporale Logik. Als einzigen Abstraktionsmechanismus bietet TLOOM das Rollenkonzept an: Konzeptionelle Objekte werden durch die Rollen beschrieben, die das Objekt spielen kann.

CMSL ist ein algebraischer Formalismus, der auf den gleichen Konzepten wie Troll basiert. Objekte werden durch Attribute, Ereignisse und zulässige Lebensläufe spezifiziert und als Datentypen aufgefaßt. Die Lebensläufe werden im Gegensatz zu Troll jedoch in einer Prozeßalgebra beschrieben.

Glider ist ebenso wie CMSL ein algebraischer Formalismus, der jedoch eine deklarativere Spezifikation als CMSL durch temporale Logik ermöglicht. Glider wurde speziell als formale Sprache zur Anforderungsspezifikation entworfen.

Object/Behavior Diagrams ist wie Graphical OBLOG eine graphische Sprache. Es werden Diagramme benutzt, die auf einem erweiterten ER-Modell (für die Strukturbeschreibung) und auf Petri-Netzen (für die Dynamikbeschreibung) basieren. Im *Object Diagram* wird die Objektstruktur definiert, im *Life Cycle Diagram* werden die Lebensläufe von einzelnen Objekten durch Zustände und Übergänge definiert, in *Activity Diagrams* werden Ereignisse, ihre Auswirkungen und Interaktionen zwischen Ereignissen verschiedener Objekte definiert.

6 Stand der Arbeit und Ausblick

Informationssysteme heutiger Prägung zeichnen sich u.a. durch eine hohe Komplexität und besonders durch erhöhte *Sicherheitsanforderungen* bezüglich der verwendeten Software aus. Während objektorientierte Verfahren zur Modellierung und Programmierung ihren Beitrag zur Beherrschung der Komplexität leisten, sind formale Ansätze geeignet um zu verläßlichen Systemen zu gelangen. Eine Integration dieser Eigenschaften ist unserer Meinung daher besonders für Modellierungsansätze geeignet.

Eine Spezifikation in Troll ist durch die Modellierung der strukturellen sowie der dynamischen Aspekte eines Weltausschnittes ein komplexes Dokument. Da eine Modellierungssprache aber auch als Kommunikationsmittel zwischen EntwicklerInnen und AnwenderInnen dienen soll, arbeiten wir gegenwärtig auf dem Gebiet der *Animation* von Troll-Modellen. Dazu wird eine Troll-Kernsprache bereitgestellt, die nur noch operationale Anteile enthält, auf die die deklarativen Anteile abgebildet werden können. Als ein Problem ist hier die Umsetzung der deklarativen Beschreibung von erlaubten Objektlebensläufen anzusehen, für die ein Ansatz in [SS92] erarbeitet wurde.

Um die Abbildung auf die Kernkonstrukte formal beschreiben zu können, wurden Graphersetzungssysteme [Fal92] und die Beschreibung der Spezifikationsdokumente als Graphen in Troll selbst untersucht. Parallel dazu wird zur Zeit an einem Parser und der Analyse der statischen Semantik der Sprache gearbeitet [Ste92, Stö93]. Beide Phasen legen einen abstrakten Syntaxgraphen an, der durch entsprechende Transformationen in einen Troll-Kern-Graphen umgewandelt werden soll. Von diesem Kern-Graphen wird dann die Codeerzeugung stattfinden.

Zur Erprobung der Konzepte für die Codeerzeugung wurde ein Prototyp eines Animationssystems erstellt. Die Übersetzung wird auf die *Eiffel* [Mey88] ähnliche Programmiersprache *Sather* vorgenommen. Für die Sather Programme wurde eine Laufzeitumgebung in *C* implementiert, die u.a. die Animation von Objektgesellschaften auf verschiedenen Knoten eines Netzwerkes ermöglicht. Eine einfache graphische Oberfläche für das Animationssystem ist mit Hilfe von *TK/TCL* implementiert worden [Kus93].

Zukünftig wird die bisher geleistete Arbeit am Animationssystem bewertet. Daraufhin soll die Übersetzung nach Sather festgelegt und in den Parser und die Analyse integriert werden. Parallel zu den Implementierungsarbeiten werden Ideen zur graphischen Repräsentierung der TROLL-Sprachkonstrukte weiter entwickelt. Graphische Repräsentierung ist neben der Animation notwendig um die Übersicht über komplexe Spezifikationsdokumente zu wahren. Ein weiterer Schritt in diese Richtung werden Modulkonzepte und Bibliotheksfunktionen sein, die in die Sprache integriert werden müssen.

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.

[Ara91] Arapis, C.: Temporal Specifications of Object Behavior. In: Thalheim, B.; Demetrovics, J.; Gerhardt, H.-D. (Hrsg.): *Proceedings 3rd. Symp. on Mathematical Fundamentals of Database and Knowledge Base Systems MFDBS–91*, Rostock, 1991. LNCS 495, Springer-Verlag, Berlin, 1991.

[Ara92] Arapis, C.: *Dynamic Evolution of Object Behavior and Object Cooperation.* Dissertation, Université de Genève, Genève, 1992.

[BB84] Batory, B.; Buchmann, A.: Molecular Objects, Abstract Data Types and Data Models: A Framework. In: *Proc. VLDB'84*, Singapore, 1984. S. 172–184.

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

[BJ82] Bjørner, D.; Jones, C. (Hrsg.): *Formal Specification & Software Development.* Prentice-Hall, Englewood Cliffs, NJ, 1982.

[BK86] Bergstra, J. A.; Klop, J. W.: Algebra of Communicating Processes. In: Bakker, J. W. de; Hazewinkel, M.; Lenstra, J. K. (Hrsg.): *Mathematics and Computer Science*, CWI Monographs 1. North-Holland, Amsterdam, 1986, S. 89–138.

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

[CHJ86] Cohen, B.; Harwood, W.T.; Jackson, M.I.: *The Specification of Complex Systems.* Addison-Wesley, Wokingham, England, 1986.

[CY89] Coad, P.; Yourdon, E.: *Object-Oriented Analysis.* Yourdon Press/Prentice Hall, Englewood Cliffs, NJ, 1989.

[DDR92] Dubois, E.; Du Bois, P.; Rifaut, A.: Elaborating, Structuring and Expressing Formal Requirements of Composite Systems. In: *Proc. 4th Conf. on Advanced Information Systems Engineering CAISE'92*, Manchester (UK), 1992. Springer-Verlag, Berlin, 1992. *To appear.*

[DDRW91] Dubois, E.; Du Bois, P.; Rifaut, A.; Wodan, P.: GLIDER Manual. Icarus deliverable, Facultés Universitaires de Namur, Namur (B), 1991.

[dF92] de Champeaux, D.; Faure, P.: A Comparative Study of Object-Oriented Analysis Methods. *Journal of Object-Oriented Programming*, Band 3, Nr. 2, 1992, S. 21–33.

[DN66] Dahl, O.-J.; Nygaard, K.: SIMULA—an Algol Based Simulation Language. *Communications of the ACM*, Band 9, Nr. 9, 1966, S. 671–678.

[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, North-Holland*, Band 9, Nr. 2, 1992, S. 157–204.

[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.

[EM85] Ehrig, H.; Mahr, B.: *Fundamentals of Algebraic Specification 1. Equations and Initial Semantics.* Springer-Verlag, Berlin, 1985.

[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. of the 2nd Workshop of "Informationssysteme und Künstliche Intelligenz: Modellierung", Ulm (Germany).* Springer IFB 303, 1992, S. 1–19.

[Fal92] Faller, O.: Ansätze zur Beschreibung und Manipulation komplexer Strukturen mit Hilfe von Graphersetzungssystemen. Diplomarbeit, TU Braunschweig, 1992.

[FSMS91] Fiadeiro, J.; Sernadas, C.; Maibaum, T.; Saake, G.: Proof-Theoretic Semantics of Object-Oriented Specification Constructs. 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. 243–284.

[GR83] Goldberg, A.; Robson, D.: *Smalltalk 80: The Language and its Implementation.* Addison-Wesley, New York, 1983.

[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.

[HJSE92] Hartmann, T.; Jungclaus, R.; Saake, G.; Ehrich, H.-D.: Spezifikation von Objektsystemen. In: Bayer, R.; Härder, T.; Lockemann, P.C. (Hrsg.): *Objektbanken für Experten.* Springer, Berlin, Reihe Informatik aktuell, 1992, S. 220–242.

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

[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.*

[Jon86] Jones, C. B.: *Systematic Software Development Using VDM.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[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.

[KS91] Kappel, G.; Schrefl, M.: Object/Behavior Diagrams. In: *Proc. Int. Conf. on Data Engineering,* Kobe, 1991. IEEE Computer Society Press, 1991, S. 530–539.

[Kus93] Kusch, J.: Implementierung eines Animationssystems für eine objektorientierte Modellierungssprache. Diplomarbeit, TU Braunschweig, Erscheint 1993.

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

[Mil89] Milner, R.: *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, 1989.

[MK90] McGregor, J. D.; Korson, T. (Guest editors): Special Issue on Object-Oriented Design. *Communications of the ACM*, Band 33, Nr. 9, 1990.

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

[Par90] Partsch, H. A.: *Specification and Transformation of Programs: A Formal Approach to Software Development.* Springer-Verlag, Berlin, 1990.

[RBP+91] Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W.: *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, NJ, 1991.

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

[Saa91] Saake, G.: Descriptive Specification of Database Object Behaviour. *Data & Knowledge Engineering*, Band 6, Nr. 1, 1991, S. 47–74. North-Holland.

[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.

[SJ92a] 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.

[SJ92b] Saake, G.; Jungclaus, R.: Views and Formal Implementation in a Three-Level Schema Architecture for Dynamic Objects. In: Gray, P.M.D.; Lucas, R.J. (Hrsg.): *Advanced Database Systems : Proc. 10th British National Conference on Databases (BNCOD 10), July 6-8, 1992, Aberdeen (Scotland).* Springer, LNCS 618, Berlin, 1992, S. 78–95.

[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.

[Spi89] Spivey, J. M.: *The Z Notation: A Reference Manual.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[SRGS91] Sernadas, C.; Resende, P.; Gouveia, P.; Sernadas, A.: In-the-large Object-Oriented Design of Information Systems. In: Van Assche, F.; Moulin, B.; Rolland, C. (Hrsg.): *Proc. Object-Oriented Approach in Information Systems.* North Holland, 1991, S. 209–232.

[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.* Informatik-Bericht 92/5, TU Clausthal, 1992, S. 207–228.

[SSE87] Sernadas, A.; Sernadas, C.; Ehrich, H.-D.: Object-Oriented Specification of Databases: An Algebraic Approach. In: Stoecker, P.M.; Kent, W. (Hrsg.): *Proc. 13th Int. Conf. on Very Large Databases VLDB'87.* VLDB Endowment Press, Saratoga (CA), 1987, S. 107–116.

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

[Stö93] Stöcker, R.: Kontextsensitive Analyse von Troll Spezifikationen. Diplomarbeit, TU Braunschweig, Erscheint 1993.

[VHL89] Van Horebeek, I.; Lewi, J.: *Algebraic Specifications in Software Engineering.* Springer-Verlag, Berlin, 1989.

[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.: Equational Specification of Dynamic Objects. 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), 1990. North-Holland, Amsterdam, 1991, S. 415–438.

[Wie91] Wieringa, R. J.: A Conceptual Model Specification Language (CMSL Version 2). Technischer Bericht IR–248, Vrije Universiteit, Amsterdam, 1991.

[Win90] Wing, J. M.: A Specifier's Introduction to Formal Methods. *IEEE Computer*, Band 23, September 1990, S. 8–24.