Objektspezifikation von Benutzerschnittstellen in TROLL[*][†]

Gunter Saake
Thorsten Hartmann
Ralf Jungclaus

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

Zusammenfassung

Die konzeptionelle Modellierung des Weltausschnittes, der durch ein Informationssystem dargestellt werden soll, ist die erste Phase des Systementwurfes. Das konzeptionelle Modell ist damit die Grundlage der Implementierung. In diesem Modell sind das später zu implementierende System *und* seine Umgebung integriert. Zwischen diesen beiden Komponenten ist die Benutzerschnittstelle einzuordnen. Die Definition dieser Schnittstelle ist damit der konzeptionellen Modellierung zuzuordnen, während ihre konkrete Darstellung im Verlaufe des Designs gestaltet wird.

In unserem Ansatz erfolgt die Modellierung der Benutzerschnittstelle in dem Formalismus, der auch zur Modellierung des Gesamtsystems verwendet wird. Das Systemdesign trennt dabei die Teile des im Modell dargestellten Weltauschnittes, die in eine interne Repräsentierung des Systems abgebildet werden, von den Teilen, die eine Abstraktion des Benutzers bzw. der Außenwelt repräsentieren. Der beschriebene Ansatz *integriert* die Beschreibung persistenter Datenbestände, deren Manipulation sowie die Beschreibung (re)aktiver Komponenten wie der Benutzer bzw. der Systemumgebung.

1 Einleitung

Die Phase des *konzeptionellen Systementwurfs* 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 von Implementierungsentscheidungen abstrahieren. Es muß lediglich festlegen, *was* realisiert werden soll, nicht jedoch *wie* dieses geschehen soll.

[*]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.

[†]Erscheint in *GI Fachtagung: Objektorientierte Methoden für Informationssysteme*, Klagenfurt, Juni 93, Springer, Informatik aktuell.

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 Objekte eingekapselt sind, somit einen *lokalen Zustand* und *lokale Aktionen* beinhalten. Das Systemverhalten ergibt sich dann aus dem lokalen Verhalten der Komponentenobjekte und der Kommunikation zwischen ihnen.

Der TROLL-Ansatz zur Spezifikation von Informationssystemen versucht die Konzepte semantischer Datenmodelle mit objektorientierten Konzepten sowie Konzepten formaler Ansätze zur Softwarespezifikation zu integrieren. Wesentliche Einflüsse kommen aus den Gebieten algebraische Spezifikation abstrakter Datentypen [EGL89, Wir90], der algebraischen Spezifikation von Prozessen [Hoa85, Mil90], der temporallogik-basierten Spezifikation reaktiver Systeme [MP92], der objektorientierten Programmierung [Weg90] und den objektorientierten Datenbanken [ABD+89, Heu92]. TROLL basiert auf dem OBLOG-Modell, das zuerst in [SSE87] vorgestellt wurde. Weitere Darstellungen und Erweiterungen sind u.a. in [SFSE89, ES91, ESS92] zu finden.

Da Informationssysteme relevante Aspekte eines Weltausschnittes implementieren, sind sie immer in eine ihre Dienste nutzende Umgebung eingebettet. In den ersten Phasen des Entwurfs liegt es daher nahe, den betrachteten Weltausschnitt als *geschlossenes System* mit seiner Umgebung zu spezifizieren.

In den späteren Phasen des Systementwurfes, die im allgemeinen als *Design* bezeichnet werden, muß – im Gegensatz zu den frühen Phasen – die Einbettung der Systemumgebung in das Modell aufgehoben und die *Schnittstelle* zwischen System und Umgebung formalisiert werden. Während ein Teil der Modellierung mit geeigneten Mitteln in eine interne Repräsentation transformiert wird, bleibt die modellierte Welt der Benutzer von dieser Transformation unberührt. Die Schnittstelle ist damit als die Grenze zwischen *Weltrepräsentation innerhalb eines Systems* und der *realen Welt anzusehen.*

Die Bausteine von TROLL-Modellierungen sind Objekte, deren Struktur in Form von Attributen sichtbar ist und deren mögliches Verhalten in Form von möglichen *Aktionensequenzen* (Prozessen) modelliert wird. Schnittstellen zwischen Benutzern und Computersystemen zeichnen sich durch nach „außen“ vermittelte Information und an das System übermittelte Befehle aus. Ebenso ist aber auch die umgekehrte Sichtweise denkbar, bei der Informationen als Eingaben dienen und Befehle vom System an die Umgebung gegeben werden (letzteres bevorzugt im Falle von Schnittstellen mit anderen Systemen). Bei beiden Sichtweisen treten bestimmte *Aktionensequenzen* auf, die manchmal auch als *Protokolle* bezeichnet werden. Mit Protokollen wird das wechselseitige *Verhalten* von Kommunikationspartnern beschrieben und es liegt somit nahe, Benutzerschnittstellen als *Objekte* zu spezifizieren. Wenn wir im folgenden von Benutzern und Benutzerschnittstellen sprechen, sind natürlich ebenfalls Benutzerinnen und Benutzerinnenschnittstellen gemeint.

Der Vorteil des in dieser Arbeit beschriebenen Ansatzes liegt in der *einheitlichen Repräsentation* der informationsspeichernden Komponente (i.a. Datenbanken), der Benutzerschnittstelle und sogar der Systemumgebung, d.h. der Benutzer selber. Damit werden Aspekte der Beschreibung persistenter Datenbestände, Operationen auf diesen Daten und die Schnittstelle zum Benutzer integriert dargestellt. Wir werden hier besonders auf die Schnittstelle und deren Beschreibung im Verlaufe des *Designs* eingehen. Dazu müssen Aspekte wie aktives Verhalten, Bedienungssequenzen und Ein- und Ausgabe von Daten deklarativ modelliert werden.

Die Sprache TROLL stellt dazu, neben den deklarativen Sprachmitteln zur konzeptionellen Modellierung, auch eine operationale Kernsprache bereit [HJ92], die als Zielsprache für den Designprozeß und auch zur Animation von Spezifikationen dienen kann [HJSE92].

Da Spezifikationen *deklarativ* formuliert sind, d.h., nicht ausführbare Konstrukte enthalten [HJ89], muß eine TROLL Spezifikation analysiert werden um direkt ausführbare und nicht direkt ausführbare Teile zu identifizieren und den Transformationsprozeß zu steuern. Wir werden diese Aufgabe hier nicht weiter betrachten. Wir bemerken nur, daß der hier beschriebene Ansatz damit auch Möglichkeiten der *Schnittstellen-Validierung* durch Prototyping bereitstellt [Rot91].

Die Arbeit ist wie folgt gegliedert. Im Abschnitt 2 werden wir die zum weiteren Verständnis notwendigen TROLL Sprachmittel einführen. In Abschnitt 3 wird dann ein Beispiel für die Modellierung einer einfachen Benutzerschnittstelle in TROLL angeben, die auf einem Beispiel in einem etablierten Formalismus basiert. Die Vorteile der TROLL-Modellierung werden motiviert. In Abschnitt 4 folgt dann eine kurze Zusammenfassung und die Identifizierung der notwendigen weiterführenden Arbeiten.

2 TROLL

Um die im nächsten Abschnitt dargestellte Modellierung von Benutzerschnittstellen im TROLL-Modell verständlich zu machen, wollen wir hier zunächst einen Überblick über die Sprache geben. Dabei werden wir uns auf die wesentlichen Konstruktionen beschränken, die für die Beispiele wichtig sind und verweisen den interessierten Leser auf [HJS93, JSHS91, JHS93, HJS92, HJSE92].

TROLL-Objekte sind *eingekapselte* Einheiten im Sinne einer Menge von Ereignissen, die auf einem nicht sichtbaren lokalen Zustand arbeiten. Die Ereignisse bilden die *Operationenschnittstelle* eines Objekts. Sichtbare Eigenschaften von Objekten werden als datenwertige Attribute modelliert, deren Wert vom internen Objektzustand abhängig ist. Einer Spezifikation unterliegend wird ein Datentyp-Universum angenommen, welches in einem Formalismus zur Spezifikation von ADT's dargestellt werden kann (z.B. [EM85, VHL89]). In einer Objektbeschreibung muß in TROLL nur die Signatur der benutzten Datentypen bekannt gemacht werden.

Objekte sind, intuitiv gesprochen, *beobachtbare Prozesse*, d.h., wir betrachten Sequenzen von Ereignissen und Beobachtungen über Anfangsstücken solcher Sequenzen. Ein wesentlicher Teil einer TROLL-Spezifikation besteht damit aus einer Prozeßbeschreibung, die die möglichen Sequenzen von Ereignissen festlegt. Ein weiterer Teil besteht aus der Beschreibung der aus den Prozessen abgeleiteten Attributbeobachtungen. Die Kommunikation von Objekten wird als *Synchronisation* von Prozessen modelliert.

Im folgenden soll nun eine Beispielspezifikation von Objekten kurz dargestellt werden. In TROLL besteht eine Objektspezifikation aus der Angabe eines *Objektnamens* und einer *Objektbeschreibung*, die nach dem Schlüsselwort **template** gegeben wird. Die Objektbeschreibung beinhaltet die Deklaration von Attributen (**attributes**) und Ereignissen (**events**) sowie verschiedene Abschnitte zur Spezifikation der zulässigen Beobachtungen (**valuation**) und des zulässigen Verhaltens (**behavior**).

In der Signaturbeschreibung werden die importierten Datentypen, Attributsymbole und Ereignissymbole festgelegt. Attribute sind typisiert, Ereignissymbole können optional mit Parametern versehen sein, die später zur Beschreibung der Auswirkungen von Ereignissen auf Attributbeobachtungen bzw. den Datenaustausch bei Objektinteraktionen verwendet werden.

Beispiel 2.1 Wir geben nun als Beispiel eine einfache Beschreibung eines Kontos an, mit dem die wesentlichen Komponenten einer TROLL-Spezifikation verdeutlicht werden.

object `Konto`

```
template
   data types money;
   attributes
      KontoStand:money;
      DispoKredit:money;
   events
      birth ErzeugeKonto(InitialerStand:money, Kredit:money);
      death LoeseKontoAuf;
      Abheben(Betrag:money);
      Einzahlen(Betrag:money);
   valuation
      variables I,D:money;
      [ErzeugeKonto(I,D)] KontoStand = I, DispoKredit = D;
      [Abheben(I)] KontoStand = KontoStand - I;
      [Einzahlen(I)] KontoStand = KontoStand + I;
   behavior
      permissions
         variables I,K:money;
         { I >= 0 } ErzeugeKonto(I,K);
         { (KontoStand + DispoKredit) > I } Abheben(I);
         { KontoStand = 0 } LoeseKontoAuf;
      obligations
         LoeseKontoAuf;
end object Konto
```

Als zu beobachtende Eigenschaften stellt dieses so spezifizierte Konto die Attribute `KontoStand` und `DispoKredit` zur Verfügung, die Werte des Datentyps `money` annehmen können. `Money` muß dabei extern spezifiziert sein und muß im **data types** Abschnitt importiert werden. Um Kontenobjekte zu erzeugen wird das mit dem Schlüsselwort **birth** qualifizierte Ereignis `ErzeugeKonto` modelliert. Dieses Ereignis besitzt zusätzlich zwei Parameter, die den initialen Kontostand sowie den Dispositionskredit modellieren. Analog zu Geburtsereignissen können mit dem Schlüsselwort **death** Ereignisse deklariert werden die ein Objekt vernichten, in diesem Falle `LoeseKontoAuf`.

In den folgenden Abschnitten einer Objektbeschreibung werden nun die Auswirkungen von Ereignissen auf Attribute und das erlaubte Verhalten eines Objektes beschrieben. Auswirkungen von Ereignissen werden mit *Auswertungsregeln* dargestellt. Die Regel

```
variables I:money;
[ Einzahlen(I) ] KontoStand = KontoStand + I
```

beschreibt dabei den Sachverhalt, daß die Einzahlung eines Geldbetrages auf ein Konto mit einer Erhöhung des Kontostandes einhergeht. Die Regel ist als Formel einer (eingeschränkten) positionalen Logik [FSMS91, FS90] anzusehen und muß gelesen werden als

> *„Nach dem Eintreten des Ereignisses* `Einzahlen`*, instantiiert mit dem Parameter* `I`*, hat der Kontostand einen Wert, der durch den Datenterm* `KontoStand+I`*, ausgewertet vor Eintreten des Ereignisses, beschrieben ist“.*

Wie im Falle des Ereignisses `ErzeugeKonto` können als syntaktische Erleichterung Auswirkungen auf verschiedene Attribute in einer Regel beschrieben werden.

Im **behavior** Abschnitt einer Objektbeschreibung wird nun das erlaubte Verhalten eines Objektes in Form von *Vorbedingungen* (**permissions**) und *Anforderungen* (**obligations**) an das Eintreten von Ereignissen beschrieben. Eine Vorbedingung der Form

```
variables I:money;
{ (KontoStand + DispoKredit) >= I } Abheben(I)
```

beschreibt, daß ein Ereignis `Abheben`, instantiiert mit dem Parameter `I` im konkreten Lebenslauf des Objektes nur dann stattfinden kann, wenn die Formel

```
( KontoStand + DispoKredit ) >= I
```

im Zustand direkt vor Eintreten des Ereignisses zu *wahr* ausgewertet wird. Die Auswertung erfolgt mit der konkreten Belegung der Variablen `I`. Als allgemeinere Formeln für Vorbedingungen sind auch *temporale* Formeln in einer vergangenheitsgerichteten temporalen Logik zugelassen, die auf den bisherigen Lebenslauf des Objektes zurückgreifen. Ein Beispiel dafür ist im nächsten Abschnitt zu finden.

In einfachen Kontenobjekten werden weiter keine Vorbedingungen an Ereignisse gestellt. Möglich ist in TROLL auch die Bedingung, daß ein bestimmtes Ereignis, eine Kombination von Ereignissen oder eine Sequenz aus Ereignissen in einem Lebenslauf enthalten sein *müssen*. Diese Anforderungen werden als **obligations** notiert. Im Beispiel wurde die in diesem Falle etwas künstliche Modellierung des Todesereignisses als Anforderung gewählt. Die Bedeutung ist in diesem Falle also, daß ein Lebenslauf des Kontenobjektes erst dann vollständig ist, wenn dieses Ereignis in einem Lebenslauf auch eingetreten ist. □

Die Darstellung von Objektlebensläufen geschieht in der oben beschriebenen Form *deklarativ*. Es werden *Anforderungen* gestellt, die für einen konkreten Lebenslauf erfüllt sein müssen. Für manche Modellierungsbeispiele ist es jedoch einfacher auf einen expliziten Formalismus zur Beschreibung von Lebensläufen zurückzugreifen, die Spezifikation von *Mustern* für Ereignissequenzen, sogenannte *patterns*. Dieser Formalismus eignet sich besonders für deterministische *Abläufe* wie sie unter anderem auch in Benutzerschnittstellen auftreten.

Beispiel 2.2 Als einfaches Beispiel sei eine Spezifikation einer Uhr angegeben, deren Lebenslauf durch einen einfachen Prozeß beschrieben ist:

```
object UHR
template
  attributes
    Stunden:nat; Minuten:nat;
  events
    birth New(Stunden:nat, Minuten:nat);
    active Tick;
    active Tack;
    StundenTack;
  valuation
    variables S,M:nat;
    [New(S,M)] Stunden = S, Minuten = M;
    { Minuten<59 } ==> [Tack] Minuten = Minuten + 1;
    { Minuten=59 } ==> [Tack] Minuten = 0;
    { Stunden<23 } ==> [StundenTack] Stunden = Stunden + 1;
    { Stunden=23 } ==> [StundenTack] Stunden = 0;
  behavior
    patterns
      variables H,M:nat;
```

```
        process START = case New(H,M) esac -> RUN
            with
                process
                    RUN = Tick -> Tack -> RUN
                end process; /*RUN*/
        end process; /*START*/
        START;
    interaction
        { Minuten = 59 } ==> Tack >> StundenTack;
end object UHR
```

In diesem Beispiel wurden *bedingte Auswertungsregeln* benutzt, um die Auswirkungen der Ereignisse auf Attribute vom Zustand abhängig zu machen. Eine bedingte Auswertungsregel kommt nur zur Anwendung, wenn die vorangestellte Bedingung mit *wahr* ausgewertet wird. Dies gilt ebenso für andere bedingte Spezifikationsformeln (Ereignisaufruf, s.u.).

Der wichtigste Bestandteil dieser Spezifikation ist die Darstellung eines *Verhaltensmusters* in Form einer expliziten Prozeßbeschreibung. Nach der Erzeugung eines Uhrobjektes verhält sich die Uhr wie im Prozeß `RUN` beschrieben: jeweils auf ein `Tick`-Ereignis folgt ein `Tack`-Ereignis und der Teilprozeß startet von vorne (Rekursion).

Die beiden Ereignisse `Tick` und `Tack` sind als **active** gekennzeichnet, da sie aus der Initiative der Uhr stattfinden. Aktive Ereignisse sind ein Modelierungsmittel um darzustellen, daß keine äußere Bedingung für das Stattfinden des Ereignisses modelliert wurde. Dieses Mittel wird wichtig bei der abstrahierenden Darstellung von Benutzern bei denen nicht alle möglichen Aktionen kausal beschrieben werden können und sollen.

Der `START` Prozeß selbst beginnt mit einem Auswahloperator (**case**). Die Parameter von `New` bestimmen dabei die Menge der möglichen Ereignisse, d.h., es wird *ein* `New` Ereignis von der Umgebung des Uhr-Objektes über seine Parameter ausgewählt. Eine Auswahl wird in diesem Falle *extern* erfolgen, z.B. durch einen Ereignisaufruf der konkrete Parameterwerte festlegt. Die Sprache zur Prozeßbeschreibung enthält neben den hier aufgeführten Operatoren *Sequenz*, *Rekursion* und *Auswahl* außerdem noch einen *Mengen-Iterator* (**foreach**).

Am Ende der Uhr-Spezifikation wird dann ein Ereignisaufruf beschrieben. Mit jedem `Tack` Ereignis wird, wenn die entsprechende Anwendungsbedingung erfüllt ist, *synchron* ein `StundenTack`-Ereignis aufgerufen. Im Fall der Uhr findet genaugenommen keine Kommunikation statt wie das Schlüsselwort **interaction** vermuten läßt. Bei Objektbeziehungen (**relationships**) wird jedoch der gleiche Aufrufmechanismus auf die Kommunikation von Objekten untereinander verallgemeinert. Da wir in der kurzen Sprachbeschreibung nur auf die wichtigsten Konstrukte eingehen können, sei auf den nächsten Abschnitt verwiesen, in dem wir ein Beispiel für Objektbeziehungen geben werden. □

Die in diesem Abschnitt gegebene Darstellung der Sprache TROLL berücksichtigt nur die für diese Arbeit notwendigen Sprachkonstrukte. In [JSHS91] und [HJS93] werden darüberhinaus Sprachmittel zur Spezifikation von Klassen, Spezialisierungen, Generalisierungen und zusammengesetzten Objekten angegeben. Wir werden im nächsten Abschnitt nun genauer auf die Benutzung von Prozeßbeschreibungen in expliziter und deklarativer Form eingehen und ein Beispiel zur Spezifikation einer Benutzerschnittstelle angeben.

3 Benutzerschnittstellen als Objekte

Benutzerschnittstellen sind zwischen implementiertem System und der Außenwelt einzuordnen. Eine Formalisierung dieser Schnittstelle wird durch die Spezifikation der Benutzerschnittstelle erreicht. Die objektorientierte Spezifikation hat den Anspruch, die folgenden beiden Aspekte dieser Schnittstelle formal zu definieren:

- die *Schnittstelle* selber als Systemkomponente
- und den Benutzer aus der Außenwelt, d.h. natürlich nur eine *Abstraktion eines echten Benutzers*. Dieser „Benutzer“ kann – als Sonderfall – natürlich auch ein anderes System sein.

Sowohl die Eingabeschnittstellen als auch der abstrakte Benutzer können sowohl *spontane* als auch *reaktive Aktivität* zeigen — in den meisten Anwendungen wird die spontane Aktivität jedoch wohl beim Benutzer und die reaktive Aktivität bei der Systemschnittstelle zu finden sein. Unter spontaner Aktivität verstehen wir dabei das ohne eine (modellierte) Kausalbeziehung zu beobachtende Eintreten von Ereignissen. Reaktive Aktivität tritt dagegen auf, wenn ein Objekt von außen (z.B. durch Ereignisaufruf) in einen Zustand gelangt, in dem nachfolgende Ereignisse zwingend eintreten müssen.

Beginnen wir mit der Modellierung von Schnittstellen als Systemkomponenten. Übliche Modellierungen von derartigen Schnittstellen setzen etwa Zustandsgraphen als Modellierungprinzip ein [Den91]. Ein Beispiel von Benutzerschnittstellen als Zustandsgraph (dort Interaktionsdiagramm genannt) kann in Abbildung 1 gefunden werden (angelehnt an das Beispiel in [Den91, Seite 130]).

Der Beispielgraph modelliert Teile des Eingabeverhaltens eines Geldautomaten. Das Dreieck bezeichnet dabei den initialen Zustand. Kreise entsprechen Zuständen des Zustandsgraphen (hier mit **Z1** bis **Z3** bezeichnet). Pfeile entsprechen Zustandsübergängen eventuell gesteuert durch Eingaben (genannt virtuelle Tasten, hier etwa der Eingabe einer ‘Geheimzahl’), während Rechtecke (aufgerufenen) Aktionen entsprechen. Die Aktivitäten nach der Prüfung der Geheimzahl (Geldauszahlung, Drucken des Kontostandes, etc.) wurden der Einfachheit halber weggelassen.

Zustandsgraphen entsprechen deterministischen Prozessen. Somit liegt es nahe, derartige Zustandsgraphen direkt als Objekte zu kodieren [Saa92]. Im folgenden Beispiel wird erst eine Benutzerschnittstelle eines Geldautomaten mit einer aktiven Phase, nämlich einer Automatenbedienung entsprechend einer Bildschirmsitzung modelliert und anschließend in den folgenden Beispielspezifikationen ein abstrakter Benutzer und seine Kommunikation mit dieser Schnittstelle beschrieben.

Beispiel 3.1 *(Schnittstellen-Signatur)* Wir beginnen mit der Spezifikation der Benutzerschnittstelle angelehnt an das in Abbildung 1 dargestellte Beispiel. Zuerst geben wir nur einen Teil der Signatur an.

```
object GASchnittstelle
template
  data types Zustände, KartenInfo, ZiffernFolge, bool;
  attributes
    Zustand:Zustände;
  events
    birth ErzeugeSchnittstelle;
    AktiviereSchnittstelle;
    active KartePrüfen(Karte: KartenInfo, ok?:bool);
```

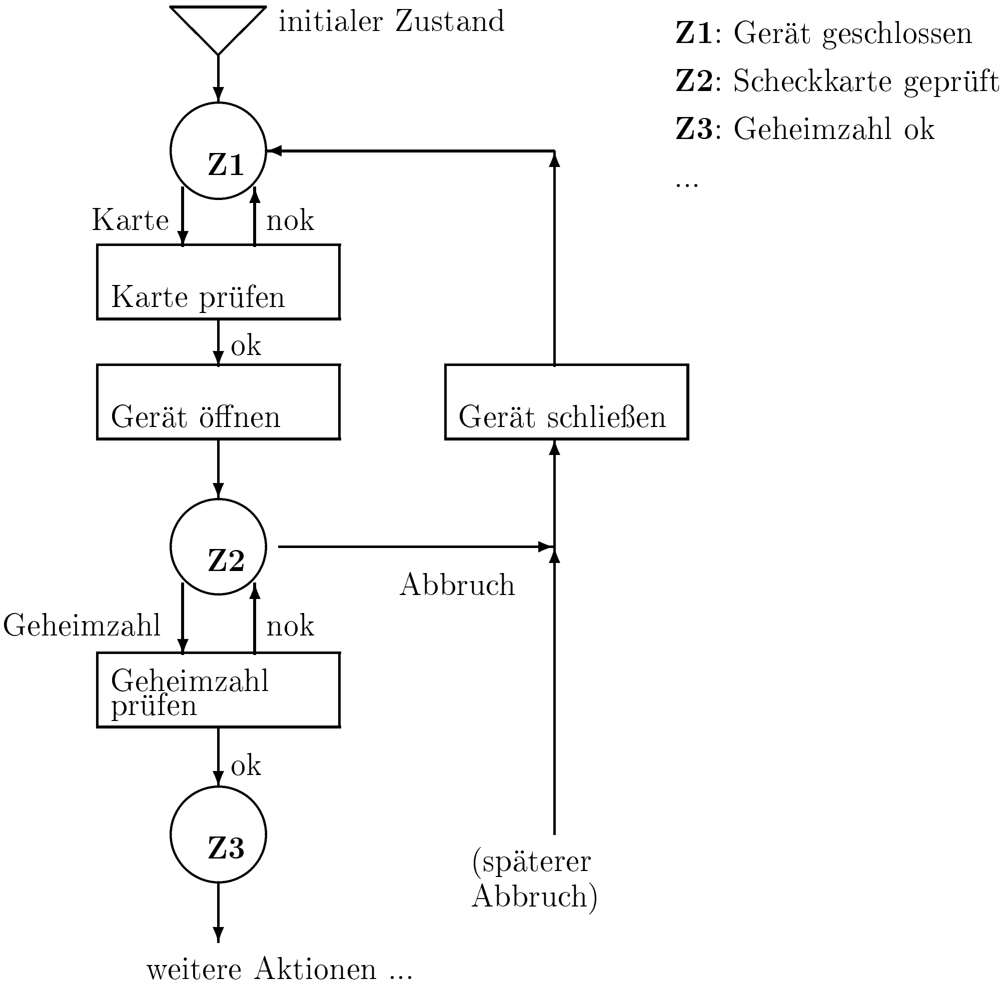

Abbildung 1: Zustandsgraph einer Benutzerschnittstelle (nach [Den91]).

```
active GerätÖffnen;
active KarteAuswerfen;
active GerätSchließen;
active GeheimzahlPrüfen(Zahl:ZiffernFolge, ok?:bool);
...;
KarteAnnehmen(Karte:KartenInfo);
GeheimzahlEingegeben(Zahl:ZiffernFolge);
AbbruchTasteGedrückt;
active AbbruchAkzeptiert;
...
```

Wir unterbrechen hier die Spezifikation des Objekts, um auf einen Unterschied zum Übergangsautomaten in Abbildung 1 hinzuweisen: In der Objektspezifikation werden Übergange durch Eingaben von außen, also vom Benutzer, als passive Ereignisse modelliert und damit von den Übergängen durch interne Entscheidungen (in Abbildung 1 ok versus nok) unterschieden, die hier Ergebnisparametern von Ereignissen oder Attributwerten entsprechen. Wir gehen im Beispiel davon aus, daß die Werte des `ok?` Parameters durch eine Anfrage (per Ereignisaufruf) bei einem anderen Objekt (z.B. einer Bank) gesetzt werden, die hier nicht weiter ausspezifiziert wird. □

Beispiel 3.2 *(Verwaltung der Zustandsinformation)* (Forts. von 3.1) Auch wenn in der TROLL Modellierung die Zustände nicht unbedingt benötigt werden, so soll hier diese Information analog zu der Beschreibung von Denert mit modelliert werden.

```
valuation
   variables z: ZiffernFolge;
   [AktiviereSchnittstelle] Zustand = Z1;
   [GerätÖffnen] Zustand = Z2;
   [GerätSchließen] Zustand = Z1;
   [GeheimzahlPrüfen(z,true)] Zustand = Z3;
   ...
```

Für die TROLL Spezifikation ist dieser Teil entbehrlich, da die Zustandsinformation implizit in einem Prozeß verwaltet wird (s.u.). Hier soll sie nur der einfachen Zuordnung zum Zustandsgraphen aus Abb. 1 dienen. □

Bisher haben wir die Objektsignatur und die Verwaltung der Zustandsinformation spezifiziert. Die eigentliche, fest vorgegebene Abarbeitung wird durch das Prozeßmuster in der `GASchnittstelle` gesteuert. Zusätzlich sollte auch eine aktive Phase eingeleitet werden um ein garantiertes Ende der Sitzung (etwa nach einer bestimmten Zeitspanne ohne Benutzerinteraktion) zu erzwingen. Hier wäre es z.B. möglich, daß Uhr-Objekt des vorigen Abschnittes zu verwenden um eine Art *TimeOut* zu spezifizieren. Darauf wurde hier verzichtet, da dieses in dem Originalbeispiel in Abbildung 1 nicht modelliert war und auch im dort verwendeten Formalismus nur auf einer Metaebene formulierbar wäre.

Beispiel 3.3 *(Verhalten der Schnittstelle)* (Forts. von 3.2) Die reaktive Aktivität wird nun durch die Gestalt des Prozesses und aktive Ereignisse spezifiziert.

```
patterns
   variables z:ZiffernFolge, k:KartenInfo;
   process GAS =
         case KarteAnnehmen(k) esac ->
         case
            KartePrüfen(k,false) -> KarteAuswerfen;
            KartePrüfen(k,true) -> GerätÖffnen -> KARTEOK?
         esac
      with
         process KARTEOK? =
               case GeheimzahlEingeben(z) esac ->
               case
                  GeheimzahlPrüfen(z,true) -> FORTSETZEN;
                  GeheimzahlPrüfen(z,false) -> KARTEOK?;
                  AbbruchTasteGedrückt -> ABBRUCH;
               esac
            with
               process ABBRUCH =
                  KarteAuswerfen -> GerätSchließen -> AbbruchAkzeptiert;
               end process;
               process FORTSETZEN = ... end process;
         end process;
   end process;
   GAS;
end object GASchnittstelle
```

Natürlich hätten wir in einer realistischen Anwendung eine Klasse von Geldautomatenschnittstellen spezifiziert anstatt wie hier eines einzelnen Objekts. Wir verzichten hier auf die Klassenspezifikation, da wir im Abschnitt 2 den Begriff der Klasse nicht eingeführt haben. □

Dieses Beispiel zeigt, daß Objektspezifikationen von Schnittstellen sich (halb-) automatisch aus Zustandsgraphen generieren lassen. Die erhaltenen Spezifikationen sehen allerdings nicht besonders gut lesbar aus; man sollte aber bedenken, daß auch Objektspezifikationen graphisch (und damit besser übersehbar) dargestellt werden können, und daß die Stärke der Objektspezifikation bei deskriptiven Beschreibungen und nicht bei der Umsetzung von operationalen Vorgaben liegt.

Beispiel 3.4 *(Benutzermodellierung)* (Fortsetzung von Beispiel 3.1–3.3) Nach der Spezifikation der Schnittstelle des Geldautomatens deklarieren wir jetzt eine spezielle Rolle von Personen, nämlich Personen in der Rolle eines `GeldAutomatBenutzer`s. Rollen wurden bisher nicht eingeführt. In TROLL erbt ein Rollenobjekt die Struktur, das Verhalten und die konkrete Zustandsinformation des Elternobjektes [JSHS91].

```
object GeldAutomatBenutzer
   role of Person;
template
   data types KartenInfo, ZiffernFolge, TastenSymbole;
   attributes
      Karte: KartenInfo;
      GeheimZahl: ZiffernFolge;
   events
      birth active SchiebeKarteInSchlitz(Karte:KartenInfo);
      active GebeZiffernfolgeEin(ZiffernFolge);
      active DrückeTaste(TastenSymbole);
      ...
      EmpfangeAbbruchBestätigung;
      EmpfangeGeld;
      death active BeendeSitzung;
   valuation
      variables K: KartenInfo;
      [ SchiebeKarteInSchlitz(K) ] Karte = K;
      ...
   permissions
      { sometime(after(EmpfangeAbbruchBestätigung)
                  or after(EmpfangeGeld) ) } BeendeSitzung;
   patterns
      process GAB = SchiebeKarteInSchlitz(Karte) ->
                        case GebeZiffernfolgeEin(GeheimZahl);
                             DrückeTaste(Abbruch);
                        esac -> ...-> BeendeSitzung
      end process;
      GAB;
end object GeldAutomatBenutzer
```

Das Symbol `Abbruch` bezeichnet hier eine Konstante des Datentyps `TastenSymbole`, der die Tastenbeschriftungen der einzelnen Tasten des Geldautomaten als Werte enthält. Die vom Benutzer erwarteten Tätigkeiten werden als Prozeß mit aktiven Ereignissen modelliert, der vom Benutzer selbst initiiert wird.

Ein Benutzer hat die Verpflichtung (sich selbst gegenüber) die Bedienung des Geldautomaten zu beenden — er muß dazu aber eine Reaktion des Automaten abwarten, entweder die Herausgabe des Geldes oder die Meldung, daß der Automat einen Abbruch akzeptiert (wer würde auch einen Geldautomaten verlassen, solange noch Geld zu erwarten ist...).

Spezifikationsteile, die zum Verständnis der Modellierung des Benutzerverhaltens nicht unbedingt notwendig sind, wurden der Übersichtlichkeit halber hier weggelassen. □

Bereits bei diesem einfachen Beispiel zeigt sich ein Vorteil der Benutzung von Objektspezifikationen gegenüber einfachen Zustandsdiagrammen. Objektspezifikationen können auch auf die Sprachmittel allgemeiner Beschreibungsformalismen zurückgreifen; so könnte etwa mittels eines Attributes die Anzahl der Fehlversuche bei der Eingabe der Geheimzahl bestimmt und gegebenenfalls eine Reaktion (etwa Abbruch) bei zu hoher Anzahl (etwa > 3) ausgelöst werden. Bei Zustandsdiagrammen müßte der entsprechende Teilgraph mit dem Zustand **Z2** hierbei dreimal kopiert werden, um das Hochzählen zu realisieren [Den91].

Die Interaktion zwischen dem Geldautomaten und der Benutzerrolle wird nun in Form einer Beziehung etabliert, um den Kommunikationsvorgang konzeptionell von den betroffenen Objekten zu trennen [JHS93]. Eine derartige Trennung erhöht unter anderem den Grad der Wiederverwendbarkeit der beteiligten Objekte und die Implementierungsunabhängigkeit der Modellierung.

Beispiel 3.5 *(Interaktionsbeschreibung)* (Fortsetzung von Beispiel 3.1–3.4) Als letzter (unvollständiger) Teil der Spezifikation der Interaktion zwischen Geldautomaten und Kunden wird eine Beziehung definiert, die die Kommunikation zwischen beiden steuert.

```
relationship GeldAutomatBenutzung
   between GeldAutomatBenutzer B, GASchnittstelle GAS;
   variables gz:ZiffernFolge, ki:KartenInfo;
   interaction
      B.SchiebeKarteInSchlitz(ki) >> GAS.KarteAnnehmen(ki);
      B.GebeZiffernfolgeEin(gz) >> GAS.GeheimzahlEingegeben(gz);
      B.DrückeTaste(Abbruch) >> GAS.AbbruchTasteGedrückt;
      GAS.AbbruchAkzeptiert >> B.EmpfangeAbbruchBestätigung;
      ...
end relationship GeldAutomatBenutzung
```

In diesem Beispiel besteht die Interaktion aus unbedingten Ereignisaufrufen. Allgemein können an dieser Stelle auch komplexere Interaktionsmuster und Integritätsbedingungen auftreten. Hier findet nun Ereignisaufruf nicht mehr innerhalb eines Objektes sondern zwischen verschiedenen Objekten statt. Wie schon oben erwähnt, wird damit die Interaktion explizit beschrieben und nicht innerhalb der Objekte versteckt. □

Analog zu Beispiel 3.5 kann nun auch die Kommunikation der Schnittstelle mit dem unterliegenden System spezifiziert werden. Hier würden zum Beispiel Ereignisaufrufe wie `GeheimzahlPrüfen` an die entsprechenden Stellen weitergeleitet. Da wir diese Systemteile in dieser Arbeit nicht spezifiziert haben, verzichten wir an dieser Stelle darauf. Zu bemerken ist, daß auf die Art der hier vorgestellten Modellierung die Schnittstelle klar zu beiden Seiten – den Benutzern und dem unterliegenden zu implementierenden System – abgegrenzt ist.

Ziel der Arbeit war es, zu zeigen, daß auch interaktive Benutzerschnittstellen eines Informationssystems mittels der ursprünglich für den Entwurf persistenter dynamischer

Objekte und deren zeitlichen Verhaltens entworfenen Spezifikationssprache Troll modelliert werden können. Dazu wurde als Beispiel eine deterministische Schnittstelle eines Geldautomaten mit Hilfe der in Troll integrierten expliziten Prozeßsprache skizziert.

4 Zusammenfassung und Ausblick

Bei der Modellierung von Informationssytemen, die ja im besonderen Maße Teile der realen Welt abbilden und somit mit der realen Welt in engem Kontakt stehen, muß besonderes Augenmerk auf die Schnittstelle zwischen dieser Abbildung und den Benutzern des Systems gelegt werden. In der hier kurz vorgestellte Sprache Troll ist die Modellierung der Benutzerschnittstelle mit Hilfe von *Objekten* möglich, die ebenso zur Modellierung des verwalteten Datenbestandes wie der (Abstraktion von) Benutzer(n) selber geeignet sind.

Ein integriertes Vorgehen – wie hier demonstriert – ist wünschenswert um eine einheitliche Beschreibung des gesamten Systems zu ermöglichen. Die Adaption anderer Formalismen ist nur dann sinnvoll, wenn die semantischen Grundlagen der benutzten Sprachen kompatibel sind. Es war daher unser Ziel, die prinzipielle Vorgehensweise einer Benutzerschnittstellen-Modellierung in Troll darzustellen, die durch die Mächtigkeit der Sprache selbst außerdem noch Vorteile gegenüber anderen Formalismen hat.

Die Vorteile eines integrierten Vorgehens sind im wesentlichen die einheitliche Sichtweise auf alle Modellierungskomponenten und die Mächtigkeit des Objektkonzeptes gegenüber Formalismen wie z.B. Zustandsdiagrammen, die im allgemeinen nur einfache Automaten beschreiben. Zusätzlich zur Objektabstraktion wurde auch die Benutzung von Relationships als über Objekte hinausgehende Sprachmittel kurz präsentiert und damit die Eigenständigkeit der Schnittstelle im Gegensatz zu Benutzer wie System hervorgehoben.

In Gebiet Schnittstellenmodellierung in Troll fehlen allerdings noch eine Entwurfsmethodik für den Entwurf und deren Validierung anhand realistischer Anwendungen. Auch der Bezug zu anderen Verfahren der Schnittstellenmodellierung, die nicht auf Übergangsautomaten basieren, muß sicher noch eingehender untersucht werden. In diesem Falle wurde eine Schnittstelle untersucht, die sich auch einfach mit operationalen Beschreibungsmitteln wie Zustandsdiagrammen darstellen läßt. Andere Arten von Schnittstellen verlangen nach deskriptiven Techniken, die noch mit den deklarativen Sprachmitteln von Troll verglichen werden müssen.

Weitergehende Arbeit hier wird unter anderem die Modellierung von Teilen eines Entwurfssystems zur Unterstützung der Modellierung in Troll selbst sein. Damit steht eine (aus der Sicht von Troll) Nicht-Standard Anwendung zur Verfügung, deren Komplexität zur Untersuchung der zur Verfügung stehenden Sprachmittel ausreicht.

Danksagung

Wir danken in erster Linie den Kolleginnen und Kollegen, die im IS-CORE Projekt mit uns zusammengearbeitet haben. Hier sind insbesondere Hans-Dieter Ehrich und Amílcar Sernadas zu nennen, deren Arbeiten zu formalen Objektmodellen die Grundlagen der Sprache Troll gelegt haben, sowie Cristina Sernadas, die an der Sprachdefinition mitgearbeitet hat.

Ferner danken wir allen Studentinnen und Studenten, die im Verlaufe von Diplom- und Studienarbeiten wertvolle Anregungen zum Sprachdesign lieferten, allen Kolleginnen und Kollegen, die in Diskussionen kritische Anmerkungen lieferten und nicht zuletzt den

anonymen Gutachterinnen und Gutachtern für ihre Kommentare zur Vorversion dieser Arbeit.

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.

[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*, Band 3, Nr. 2, 1992, S. 21–33.

[Den91] Denert, E.: *Software-Engineering. Methodische Projektabwicklung*. Springer-Verlag, Berlin, 1991.

[EGL89] Ehrich, H.-D.; Gogolla, M.; Lipeck, U.W.: *Algebraische Spezifikation abstrakter Datentypen*. Teubner, Stuttgart, 1989.

[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 (D)*. Springer IFB 303, 1992, S. 1–19.

[FS90] Fiadeiro, J.; Sernadas, A.: Logics of Modal Terms for System Specifications. *Journal of Logic and Computation*, Band 1, Nr. 2, 1990, S. 187–227.

[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.

[Heu92] Heuer, A.: *Objektorientierte Datenbanken*. Addison-Wesley, Bonn (D), 1992.

[HJ89] Hayes, I.J.; Jones, C.B.: Specifications are not (necessarily) executable. Technischer Bericht UMCS-89-12-1, Univ. of Manchester, 1989.

[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.

[HJS93] Hartmann, T.; Jungclaus, R.; Saake, G.: Spezifikation von Informationssystemen als Objektsysteme. *EMISA Forum, Mitteilungen der GI-Fachgruppe 2.5.2*, Band 1, 1993, S. 2–18.

[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.

[JHS93] 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, 1992, Hotel Ellivuori (SF).* IOS Press, Amsterdam. *Erscheint 1993.*

[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.

[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.

[RBP+90] Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W.: *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, NJ, 1990.

[Rot91] Rothenberg, J.: Prototyping as Modeling: What is being Modeled? In: Sol, H.G.; van Hee, K.M. (Hrsg.): *Dynamic Modeling of Information Systems.* Elsevier Science Publishers, 1991, S. 335–359.

[Saa92] Saake, G.: Objektorientierte Spezifikation von Informationssystemen: Konzepte und Sprachvorschläge. Habilitationsschrift, TU Braunschweig, 1992.

[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.

[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.

[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, OOPS 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.