In H. C. Mayr and B. E. Meyer, editors, Proc. Fachgespräch "Entwurf von Informationssystemen - Methoden und Modelle", pages 1-19, Tutzing, 1985. Evangelische Akademie.

Spezifikation konzeptioneller Schemata mit abstrakten Datentypen und Versionen

H.-D. Ehrich

Inst.f.Informatik, TU Braunschweig, Postfach 3329
D-3300 Braunschweig

Abstract

Illustrated by an example of a simple graphical database, an informal survey of some new concepts for conceptual modelling and specification of non-standard database applications is presented. Main features are the incorporation of user-defined abstract data types and user-defined transactions, especially for versioning. The approach emphasizes a clear conceptual separation of data, object, and transaction levels as well as a sound semantical basis.

1. Einleitung

Die Planung, Entwicklung und Konstruktion immer größerer Software-Systeme hat zu stark wachsenden Anforderungen an Methoden und Hilfsmittel geführt, die geeignet sind, diesen Entwicklungsprozess effizienter und zugleich exakter und zuverlässiger zu gestalten. In vielen Bereichen der Informatik werden insbesondere Ansätze diskutiert, die versuchen, durch Einführung einer höheren Abstraktionsebene der Systembeschreibung zur Bewältigung der Probleme beizutragen. Eine interessante Gegenüberstellung derartiger Ansätze in den Bereichen Künstliche Intelligenz, Programmiersprachen und Datenbanken wird in [BMS84] gegeben, wo für die Systembeschreibung auf höherer Abstraktionsstufe allgemein die Bezeichnung "konzeptionelle Modellierung" (conceptual modelling) eingeführt wird.

Nahezu alle Vorschläge zur Methodik des Entwurfs von Datenbanken und Informationssystemen gründen sich auf ein sogenanntes "semantisches Datenmodell", mit dem sich

möglichst "direkt" und "natürlich" der von der Anwendung gegebene Ausschnitt der realen Welt modellieren läßt, bevor man zu detaillierten "logischen" und "physischen" Entwürfen der Datenbank-Schemata voranschreitet.

Viele dieser "semantischen Datenmodelle" orientieren sich ziemlich eng an den Konzepten verfügbarer Datenbanksprachen und weisen entsprechende Unzulänglichkeiten auf, besonders bei der Spezifikation von Integritätsbedingungen, aber auch bei der Einbeziehung von abstrakten Datentypen und bei der Modellierung von Zeitaspekten, insbesondere Versionen. Weitere Schwächen bestehen bei der Darstellbarkeit partieller Information, oft auch bei den semantischen Grundlagen und daher bei der Präzision und der inneren Konsistenz des Modells, sowie bei der Verfügbarkeit von automatischen Hilfsmitteln zur Unterstützung des Entwurfsvorgangs.

In dieser Arbeit wird anhand eines konkreten und anschaulichen Beispiels einer einfachen graphischen Datenbank ein informeller Überblick über einige neuere Ideen zur konzeptionellen Modellierung gegeben, insbesondere zur Strukturierung und Spezifikation konzeptioneller Schemata für Nicht-Standard-Datenbank-Anwendungen. Wesentliche Teile unseres Ansatzes sind in [ELG84, LEG85] beschrieben, vor allem die Einbeziehung abstrakter Datentypen und die Spezifikation von statischen und dynamischen Integritätsbedingungen. Dem fügen wir hier einen ersten Vorschlag zur Spezifikation von Transaktionen hinzu, der auch die Spezifikation von Transaktionen zur Versionen-Verwaltung erlaubt.

Wesentliches Kernstück unseres Ansatzes ist die Unterscheidung von drei aufeinander aufbauenden, begrifflich jedoch streng zu trennenden Schichten eines konzeptionellen Schemas:

1. Datentypen
2. Objekttypen
3. Transaktionen

Die meisten "semantischen Datenmodelle" unterscheiden nicht konsequent zwischen Daten und Objekten, und benutzerdefinierte Transaktionen werden häufig nicht in die Be-

trachtung einbezogen.

Bei der Entwicklung unseres Ansatzes wurden wir angeregt und beeinflußt von einer Reihe von Arbeiten mit ähnlicher Ausrichtung, insbesondere von [GMS83, CCF82, CF82, Se80, Ku84]. Diesen Arbeiten verdanken wir vor allem Hinweise auf die Nützlichkeit modaler bzw. temporaler Logik für die Datenbank-Spezifikation. Wir gründen unseren Ansatz auf das funktionale Datenmodell [BF79, Sh81], da dies die Übertragung algebraischer Konzepte aus der Theorie der abstrakten Datentypen [Eh82, EM85, Kl83] ermöglicht (s. auch [GMS83]). Die wesentlichen Ideen lassen sich jedoch ebenfalls auf einer relationalen Grundlage aufbauen (wie z.B. in [CCF82, CF82]).

In [ELG84, LEG85] behandeln wir lediglich die ersten beiden Schichten, Daten und Objekte. Der hier beschriebene Ansatz zur Spezifikation benutzerdefinierter Transaktionen benutzt soweit möglich die konventionelle Methode der Angabe von Vor- und Nachbedingungen. Wo dies nicht ausreicht - und das ist z.B. bei der Verwaltung von Versionen der Fall - greifen wir auf die von Bartussek und Parnas vorgeschlagene Methode der Trace-Spezifikation zurück [BP77, McL84].

2. Das Beispiel

Es soll eine graphische Datenbank zur Verwaltung von Dreieck-Szenen spezifiziert werden. Eine Dreieck-Szene (Bild 1) besteht aus einer Menge sich nicht überlappender Dreiecke mit verschiedenen Farben (im Bild nicht dargestellt). Dreiecke sollen u.a. eingefügt und verschoben werden können. Diese Operationen sollen sich auf die jeweils in Arbeit befindliche Szene ("current") beziehen. In jeder Szene (nicht nur in der "current"-Szene) ist ein aktuell in Arbeit befindliches Dreieck ("actual", im Bild schraffiert) ausgezeichnet.

Von der "current"-Szene sollen Versionen angelegt werden können. Gearbeitet wird immer in der neuesten Version. Es soll auf die jeweils vorherige Version zurückgegangen werden können, wobei die neueste Version verlorengeht

(LIFO-Stack von Versionen). Ältere Versionen sind ansonsten nicht direkt zugänglich.

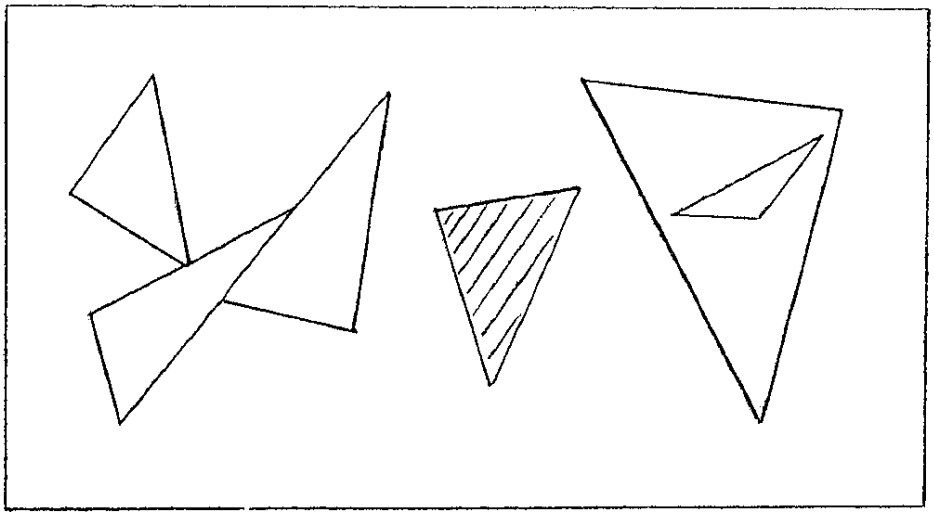

Bild 1 : Eine Dreieck-Szene

Bemerkung : Diese Art der Versionenverwaltung ist sicherlich zu simpel für die meisten realen Situationen. Es sollte jedoch hinreichend deutlich werden, daß - und wie - auch realistischere Arten der Versionenverwaltung spezifiziert werden können.

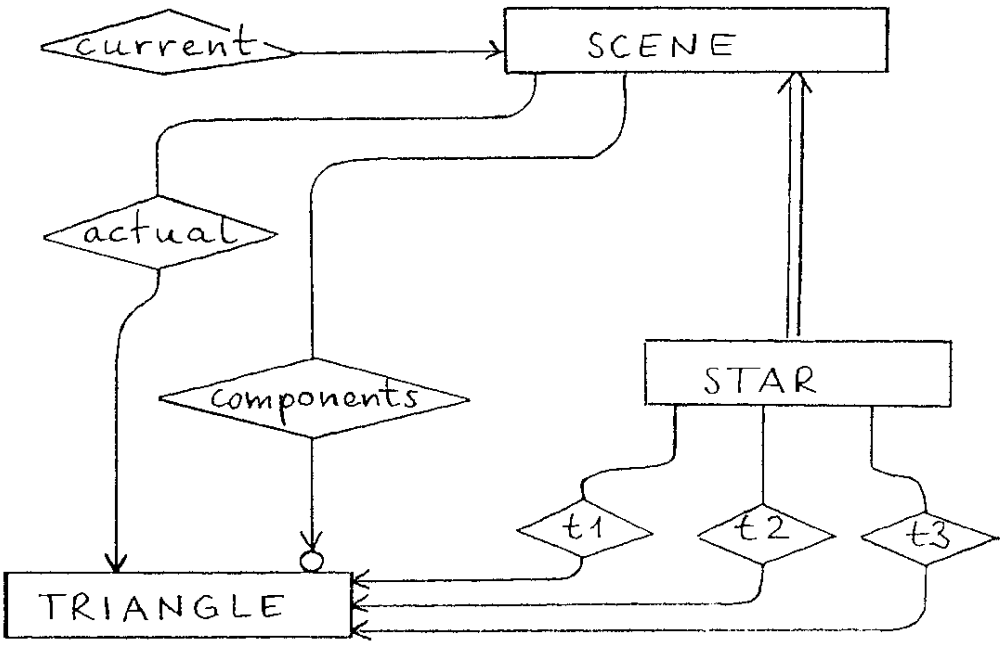

Bild 2 : ER-Diagramm

Bild 2 gibt einen ersten Überblick über die Struktur der Objekte und Beziehungen als (funktionales) ER-Diagramm (Attribute sind weggelassen). Es gibt drei Objekt-Typen : SCENE, TRIANGLE und STAR. current ist eine Variable vom Typ SCENE, und dies ist als nullstellige Objekt-funktion darstellbar. components ist eine mengenwertige Objektfunktion, dargestellt durch den Kreis an der Pfeilspitze; components ordnet jeder Szene die in ihr enthaltenen Dreiecke zu. Ein Star soll eine spezielle Szene sein, die aus drei Dreiecken t1, t2, t3 besteht, deren Grundseiten wiederum ein Dreieck bilden (Bild 3). Das so entstehende Zentraldreieck ist nicht Komponente der Szene ! Der Doppelpfeil im ER-Diagramm von STAR nach SCENE stellt eine Generalisierungs-Beziehung dar.

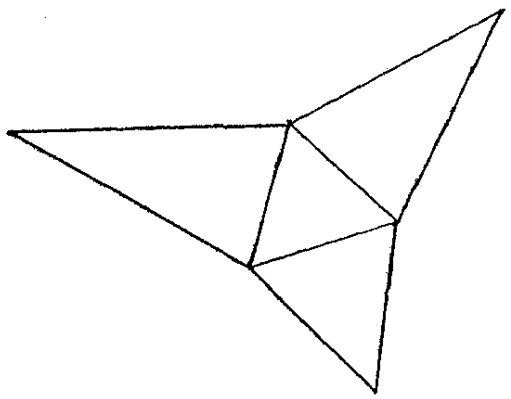

Bild 3: Ein STAR

Die Dreiecks-Szenen unseres Beispiels sollen einer Reihe von Integritätsbedingungen genügen. Diese lassen sich im ER-Diagramm nicht darstellen. Für SCENE allgemein fordern wir:

IB1 : Das aktuelle Dreieck muß in den Komponenten einer Szene enthalten sein.

IB2 : Dreiecke in einer Szene dürfen sich nicht überlappen. Ineinander dürfen sie liegen. (Dreieckseiten dürfen sich berühren, jedoch nicht schneiden.)

IB3 : Die Gesamtfläche aller Dreiecke in einer Szene darf nur steigen.

IB4 : Jedes Dreieck in einer Szene muß irgendwann einmal ein anderes berühren, d.h. es darf keine "dauernd isolierten" Dreiecke geben.

Die letzten beiden Bedingungen sind dynamisch. Sie geben keine Kriterien für die Korrektheit einzelner Szenen,

sondern für die Korrektheit von Szenenfolgen im Laufe der Zeit ("Filmen"). Hinzu kommen noch die folgenden Integritätsbedingungen speziell für STAR:

IB5 : Die Komponenten eines STAR s sind t1(s), t2(s) und t3(s).

IB6 : Die Grundseiten (A-B) der drei Dreiecke bilden wieder ein Dreieck, das "Zentrum".

IB7 : Das Zentrum soll gleichseitig sein.

IB8 : Die drei Dreiecke sollen gleichschenklig sein, wobei die gleichen Schenkel vom außenliegenden Punkt (C) ausgehen.

Zur Demonstration unseres Ansatzes zur Spezifikation von Transaktionen, insbesondere solchen zur Verwaltung von Versionen, werden folgende Transaktionen hier spezifiziert :

ADDTRI : Einfügen eines neuen Dreiecks sowohl in TRIANGLE als auch als aktuelles Dreieck in die aktuelle Szene (current).

SHIFTACT : Verschieben des aktuellen Dreiecks in der aktuellen Szene um einen angebenen Vektor.

SIZEOFMAXSTAR : Ermittlung der Größe (Fläche) der größten Szene vom speziellen Typ STAR.

NEWVERSION : Anlegen einer neuen Version.

OLDVERSION : Zurücksetzen auf die letzte Version.

Selbstverständlich sind für ein vollständiges CAD-System für Dreiecksszenen viele weitere Transaktionen erforderlich. Die grundlegenden Prinzipien lassen sich jedoch bereits an dieser kleinen Auswahl demonstrieren.

3. Datentypen

Jedes Informationssystem basiert auf Daten als grundlegenden Einheiten der Eingabe, Ausgabe, Speicherung und Verarbeitung. Es ist verbreitete Praxis, Daten gemäß den auf sie anwendbaren Operationen zu Datentypen zusammenzufassen. So finden sich z.B. Standardtypen wie bool, int, real, char, text, etc mit ihren charakteristischen Operationen in vielen Programmiersprachen.

Sei langem gibt es Bestrebungen, die verfügbaren Standard-Datentypen um anwendungsbezogene, benutzerdefinierte Datentypen zu erweitern. Im Rahmen der konzeptionellen Modellierung geht es darum, "abstrakte" Datentypen zu spezifizieren, um sie später mit den Mitteln einer verfügbaren Programmier- bzw. Datenbanksprache zu implementieren. Die Vorteile dieser Vorgehensweise sind in der Literatur so vielfach beschrieben worden, daß wir hierauf nicht näher eingehen.

Unter den Methoden zur Spezifikation abstrakter Datentypen hat die sogenannte "algebraische Spezifikation" in den letzten Jahren zunehmend starke Beachtung gefunden, Es gibt bereits zwei Lehrbücher [EM85, KI83], die diese Methode und ihre theoretischen Grundlagen darstellen. Wir orientieren uns hier an dieser Methode, ohne die Spezifikationen jedoch im Detail auszuführen. Wir gehen in unserem Beispiel lediglich die Signaturen der abstrakten Datentypen formal, d.h. die den jeweiligen Datentyp bestimmenden Operationen mit ihren Argument- und Wertsorten, in einer wohl für sich verständlichen Ad-hoc-Notation. Die beabsichtigte Wirkung der Operationen erläutern wir nur verbal. Dies kann im Prinzip auch formal, etwa durch die Angabe von (Bestimmungs-) Gleichungen, geschehen, würde jedoch den Rahmen dieser Arbeit sprengen.

Eine Signatur $\Sigma=(S,\Omega)$ besteht aus einer Menge von Sorten S und einer Menge von Operatoren Ω. Ein Operator besteht aus einem Operatorsymbol ω, einer Liste von Argumentsorten $s_1,\ldots,s_n$ und einer Wertsorte s_0. Wir notieren dies folgendermaßen:

$$\omega : s_1 \times s_2 \times \ldots \times s_n \rightarrow s_0$$

Es kann n=0 sein, ω : $\rightarrow s_0$. In diesem Fall bezeichnet ω eine Konstante der Sorte s_0 (Konstanten entsprechen nullstelligen Funktionen).

Es seien (Spezifikationen von) Standard-Datentypen bool, int, real, text mit Sorten bool, int, real, text (ohne Unterstreichung) und hier nicht näher bezeichneten Standard-Operatoren gegeben. Diese verwenden wir zur Spezifikation der für unser Dreiecks-Szenen-Beispiel relevanten abstrakten Datentypen point, line, triangle, color :

3.1 point

```
sorts point,real
ops   createpoint : real x real -> point
      x           : point -> real
      y           : point -> real
```

createpoint soll aus zwei real-Zahlen a,b den Flächen-punkt mit den Koordinaten (a,b) generieren, x und y ergeben die jeweilige Koordinate, also x(a,b)=a und y(a,b)=b.

3.2 line

```
sorts line, point, bool
ops   createl     : point x point -> line
      begin       : line -> point
      end         : line -> point
      intsct      : line x line -> bool
```

createl generiert eine Linie aus Anfangs- und Endpunkt, und begin und end ergeben den jeweiligen Anfangs- bzw. Endpunkt; intsct prüft, ob sich zwei Linien echt schneiden. Berühren sie sich nur, soll der Wert false sein.

3.3 triangle

```
sorts triangle, point, bool, real
ops   createt  : point x point x point -> triangle
      A        : triangle -> point
      B        : triangle -> point
      C        : triangle -> point
      shift    : triangle x point -> triangle
      area     : triangle -> real
      touch    : triangle x triangle -> bool
      equi     : triangle -> bool
      iso      : point x triangle -> bool
```

createt generiert ein Dreieck aus seinen drei Eckpunkten, die mit A, B und C wieder ermittelt werden können; shift verschiebt ein Dreieck, wobei der Punkt im zweiten Argument als Verschiebevektor dient; area gibt den Flächeninhalt des Dreiecks; touch prüft, ob sich zwei Dreiecke berühren oder schneiden (d.h. ob die Seiten mindestens einen Punkt gemeinsam haben); equi prüft, ob das Dreieck gleichseitig ist (equilateral); iso prüft, ob die von dem angegebenen Punkt ausgehenden Seiten gleich lang sind, d.h. ob das Dreieck gleichschenklig ist (isosceles).

3.4 color sorts color

ops red : -> color
blue : -> color
yellow : -> color
green : -> color
.....

Die Farbwerte sind Konstanten der Sorte color.

Eine formale Ausführung dieser Datenspezifikation würde der Signatur $\Sigma=(S,\Omega)$ mit den oben gegebenen Sorten und Operatoren - anstelle der verbalen Erläuterungen - Axiome hinzufügen, z.B. in der Form von Gleichungen, die die Wirkung der Operatoren festlegen. Man erhielte so eine (algebraische) Spezifikation

$$D = (\Sigma, E),$$

wobei E eine Menge von Gleichungen ist.

Als Semantik der Spezifikation nehmen wir eine fest gegebene Interpretation DATA

- der Sorten s durch Mengen $\hat{s}$
- der Operatoren $\omega : s_1 x \ldots x s_n \rightarrow s_0$ durch Operationen $\hat{\omega} : \hat{s}_1 x \ldots x \hat{s}_n \rightarrow \hat{s}_0$

als gegeben an. DATA sei ein durch D spezifizierter Datentyp (D spezifiziert einen abstrakten Datentyp, i.e. eine Klasse von Datentypen. Algebraisch gesehen ist DATA eine Algebra zur Signatur Σ, die die Axiome E erfüllt.).

4. Objekttypen

Die Daten-Spezifikation D = (Σ,E) mit ihrer festen Semantik DATA bildet die Grundlage, auf der die Schicht der Objekttypen aufbaut. Objekte sind in der Datenbank darzustellende Einheiten. Objekte können jedoch nicht direkt eingegeben, ausgegeben, gespeichert oder verarbeitet werden. Sie können nur durch Daten beschrieben werden, und nur diese Daten können eingegeben, ausgegeben, gespeichert und verarbeitet werden.

Die begriffliche Unterscheidung zwischen Daten und Objekten ist wesentlich für unseren Ansatz. Objekte sind Dinge, über die Informationen in der Datenbank gespeichert werden sollen, während Daten Dinge sind, die diese Information selbst ausdrücken. Ein komplexes Datum wie z.B. eine Adresse wird eine andere, wenn sich nur eine Komponente, z.B. die Straße oder die Hausnummer ändert. Ein Objekt wie z.B. eine Person soll dagegen durchaus seine Identität behalten dürfen, wenn sich ein beschreibendes Datum, z.B. die Adresse, ändert. In unserem Dreieck-Szenen-Beispiel wird dies am Beispiel der Dreiecke deutlich : wir haben einen Datentyp triangle und einen Objekttyp TRIANGLE (s.u.).

Informationen über Objekte werden im funktionalen Datenmodell durch Funktionen dargestellt : datenwertige Funktionen stellen Attribute dar, und objektwertige Funktionen stellen Beziehungen dar. Analog zu den Datentypen werden Objekttypen durch die auf ihnen erklärten Funktionen (Attribute und Beziehungen) bestimmt. Dies führt dazu, daß sich die Struktur der Objekte formal - ebenso wie die der Daten - durch eine Signatur $\Sigma' = (S', \Omega')$ ausdrücken läßt. Hierbei ist S' eine Menge von Objekt-Sorten und Ω' eine Menge von Objekt-Funktionen der Form $\omega : s_1 x \ldots s_n \rightarrow s_0$, wobei die s_i, $1 \leq i \leq n$, Daten- und Objektsorten sein können.

Insgesamt erhalten wir so eine hierarchische Signatur $\Sigma + \Sigma'$, die eine Erweiterung der Datensignatur Σ ist.

Die Semantik von Σ' ist in der Tat verschieden von der von Σ : Interpretationen von Σ' sind Datenbankzustände, und diese sollen im Laufe der Zeit variieren.

Um die zulässigen Zustände und Zustandsfolgen einzugrenzen, ist die Angabe von Integritätsbedingungen erforderlich. Diese formulieren wir in einer um die temporalen Quantoren always und sometime angereicherten Prädikatenlogik. Bzgl. der präzisen Semantik dieser Temporalen Logik sei auf [LEG85] verwiesen. Wir begnügen uns hier mit erläuternden Kommentaren zum Beispiel.

Neben Objekt-Sorten, Objekt-Funktionen und Integritätsbedingungen enthält die folgende Objekt-Spezifikation noch definierte Funktionen. Diese stellen eine rudimentäre deduktive Komponente dar und wurden aufgenommem, um auf Erweiterungsmöglichkeiten in dieser Richtung aufmerksam zu machen. Wir benutzen dann definierte Funktionen wie andere Funktionen, z.B. in den Integritätsbedingungen.

Nach diesen Vorbemerkungen sollte die Spezifikation der Objekttypen unseres Dreieck-Szenen-Beispiels verständlich sein :

```
4.1  TRIANGLE
     sorts      TRIANGLE, text, color, triangle
     functions  tname     : TRIANGLE -> text      key
                tcolor    : TRIANGLE -> color
                triang    : TRIANGLE -> triangle
     define     lines     : TRIANGLE -> set line
          by    lines(t) = { createl (A(u),B(u)),
                             createl (B(u),C(u)),
                             createl (C(u),A(u))  }
          where      u = triang(t)
     end TRIANGLE
```

Bemerkungen : Das Symbol key bestimmt das Attribut tname zum Schlüssel. TRIANGLE hat nur Attribute, keine Beziehungen (ist demnach ein "relationaler" Objekttyp : Objekte können durch Tripel (tname, tcolor, triang) dargestellt werden). Das Symbol set bestimmt lines zu einer mengenwertigen Funktion. Zur formal sauberen Einfügung in unseren Rahmen muß man sich die Datenschicht um einen weiteren Datentyp "set line" erweitert denken, der aus Linienmengen besteht. Dann ist lines eine Funktion mit Datenelementen dieses Typs als Werten.

4.2 SCENE

```
sorts      SCENE, TRIANGLE, int, real, text
functions  sname      : SCENE -> text      key
           version    : SCENE -> int
           current    : -> SCENE
           components : SCENE -> set TRIANGLE
           actual     : SCENE -> TRIANGLE
define     sarea      : SCENE -> real
  by  sarea(s) = SUM [area(triang(t))|t∈components(s)]
constraints
     actual(s) ∈ components(s)
     t,t' ∈ components(s) ∧ l∈lines(t) ∧ l'∈lines(t')
               => ¬ intsect (l,l')
     sarea(s) = a => always sarea(s) ≥ a
     t ∈ components(s)
        => sometime ∃ u ∈ components(s) : touch(t',s')
        where t' = triang(t), s' = triang(u)
end SCENE
```

Bemerkungen : Hier haben wir neben den Attributen sname und version auch Beziehungen : current ist (als nullstellige Beziehung) eine Variable der Sorte SCENE; components ordnet jeder Szene die Menge aller Dreiecks-Objekte zu, die gerade (d.h. im aktuellen Zustand) in der Szene sind. Die Mengenwertigkeit von components kann analog zum Fall lines (s.o.) auf Einwertigkeit in einem neuen Objekttyp set TRIANGLE zurückgeführt werden. Die angegebenen Integritätsbedingungen formalisieren die oben im Abschnitt 2 angegebenen Bedingungen IB1 bis IB4.

4.3 STAR sub SCENE

```
sorts STAR, TRIANGLE, triangle, color
functions centercolor    : STAR -> color
          t1             : STAR -> TRIANGLE
          t2             : STAR -> TRIANGLE
          t3             : STAR -> TRIANGLE
define    center         : STAR -> triangle
     by   center(s) = createt( A(triang(t1(s))),
                               A(triang(t2(s))),
                               A(triang(t3(s))) )
```

```
constraints
    components(s) = { t1(s),t2(s),t3(s) }
    A(triang(t1(s))) = B(triang(t3(s)))
    A(triang(t2(s))) = B(triang(t1(s)))
    A(triang(t3(s))) = B(triang(t2(s)))
    equi(center(s))
    iso(C(triang(t1(s))),triang(t1(s)))
    iso(C(triang(t2(s))),triang(t2(s)))
    iso(C(triang(t3(s))),triang(t3(s)))
end STAR
```

Bemerkungen : Das Symbol sub drückt eine Generalisierungs-/Spezialisierung-Beziehung aus. Alle Objektfunktionen von SCENE werden dadurch auf STAR vererbt. Der aktuelle Wert von current kann z.B. auch ein STAR sein. Die Integritätsbedingungen für STAR formalisieren die Bedingungen IB5 bis IB8 aus Abschnitt 2.

Allgemein besteht die Spezifikation der Objektschicht aus der Angabe einer Objektsignatur Σ' und der Angabe von Integritätsbedingungen C :

$$O = (\Sigma', C)$$

Auf den Objekt-Sorten S' ist außerdem eine Halbordnung $\leq$, die Generalisierungsbeziehung, erklärt. Zusammen mit der Datenspezifikation ergibt dies eine hierarchische Schemaspezifikation

$$SCH = D + O$$

Die Semantik von SCH ist eine Klasse von Zustandsfolgen und charakterisiert so das zulässige Verhalten der Datenbank. Ein Zustand ist eine Interpretation von SCH. Er besteht aus der festen Interpretation DATA für D, erweitert um eine Interpretation

- der Objektsorten s durch Mengen $\sigma(s)$ "aktueller" Objekte
- der Objektfunktionen $\omega : s_1 x \ldots x\ s_n \rightarrow s_0$ durch "aktuelle" Funktionen $\sigma(\omega) : \sigma(s_1) x \ldots x\ \sigma(s_n) \rightarrow \sigma(s_0)$

Für eine Datensorte s ist hierbei $\sigma(s)$ die feste Interpretation in DATA. Ansonsten kann σ durch Transaktionen (s.u.)

verändert werden.

Um die möglichen Zustände generell einzugrenzen, ist es zweckmäßig, zu jeder Objektsorte s ein Universum $\pi(s)$ der "möglichen" Objekte dieser Sorte anzunehmen, so daß dann stets $\sigma(s) \subseteq \pi(s)$ ist. Eine weitere Einschränkung auf "zulässige" Zustände und Zustandsfolgen wird durch die Angabe der Integritätsbedingungen erreicht. In [LEG85] ist die Klasse der durch eine Schemaspezifikation SCH charakterisierten zulässigen Zustandsfolgen präzise beschrieben.

5. Transaktionen

Im Unterschied zu Objektfunktionen, die einzelne Objekte bzw. Datenelemente als Argumente und Werte haben, hängen Transaktionen vom Gesamt-Zustand ab. Wir unterscheiden zwei Arten von Transaktionen, Änderungs-Transaktionen und Wert-Transaktionen. Diese entsprechen den echten Prozeduren und den Funktions-Prozeduren in Programmiersprachen. Wert-Transaktionen ermitteln einen vom Gesamt-Zustand abhängigen Wert durch eine "Aggregierungs-Funktion" (Summe, Maximium, Mittelwert, etc.). Wir geben ein Beispiel (s.u. 5.3), konzentrieren jedoch unser Hauptaugenmerk auf Änderungstransaktionen.

Die Wirkung von Transaktionen läßt sich in den meisten Fällen einfach durch die Angabe von Vor- und Nachbedingungen beschreiben. Diese Spezifikationstechnik ist weit verbreitet und ohne Zweifel akzeptabel und praktikabel. Sie ist jedoch nicht immer anwendbar. Es gibt Transaktionen mit gleichsam "verborgener" Wirkung, die sich am nächsten Zustand nicht oder nicht vollständig ablesen läßt.

Hierzu gehört z.B. die Anlage einer neuen Version : man wird in unserem Beispiel die Versionsnummer um eins erhöhen, aber ansonsten bleiben alle Komponenten, alle Attribute und alle Beziehungen unverändert. Die beabsichtigte Wirkung zeigt sich erst später, unbestimmt lange nach der Anlage der Version, nämlich wenn diese Version wiederhergestellt werden soll.

Man kann solche Transaktionen durchaus auf das klassische Vorher-/Nachher-Modell "trimmen". Dazu ist es nötig, eine Art Hintergrundspeicher für Versionen zu einem Bestandteil des Zustands zu machen. Die Anlage einer Version läßt sich dann vollständig im so erweiterten Nachfolgezustand ablesen : die Version ist in den Hintergrundspeicher eingetragen.

Diese Vorgehensweise erscheint jedoch recht implementierungsnah. Bei der konzeptionellen Modellierung sollte der Zustand möglichst keine Verwaltungsinformation ohne direkten Anwendungsbezug enthalten. Es sollte möglich sein, die beabsichtigte Wirkung direkt zu spezifizieren, ohne bei zusätzlichen Hilfs-Objekten Zuflucht nehmen zu müssen.

Eine attraktive Möglichkeit, diesem Ziel zu entsprechen, bietet die Trace-Spezifikation [BP77, McL84]. Hierbei werden verschiedene Folgen von Transaktionen (traces) als gleich erklärt, d.h. sie sollen die gleiche Wirkung haben. Trace-Spezifikationen sind mit den algebraischen Spezifikationen abstrakter Datentypen verwandt, jedoch ist die semantische Grundlage unterschiedlich.

Nachfolgend geben wir eine Auswahl von Spezifikationen für Transaktionen in einer nach den gemachten Erläuterungen wohl für sich verständlichen Ad-hoc-Notation.

5.1 ADDTR1 (n:text, c:color, t:triangle)

pre –

post ∃T : TRIANGLE :

components(current) = components(current[pre])∪{T}

∧ T = actual(current)

∧ tname(T) = n

∧ tcolor(T) = c

∧ triang(T) = t

Bemerkung : Der Zusatz [pre] bezieht sich auf den Vorher-Zustand. es wird also ein neues Dreiecks-Objekt T eingefügt, mit den mitgegebenen Attributen versehen und zum aktuellen Dreieck der Szene gemacht.

5.2 SHIFTACT (p:point)
pre components(current) ≠ ∅
post actual(current) = shift(actual(current[pre]),p)

5.3 SIZEOFMAXSTAR () returns max : real
pre ∃s : STAR
post max = MAX{sarea(s) | s : STAR}

Bemerkung : Dies ist ein Beispiel einer Wert-Transaktion, die den Zustand nicht verändert.

5.4 NEWVERSION ()
pre -
post version(current) = version(current[pre]) + 1

Bemerkung : Die "eigentliche" Wirkung ergibt sich erst durch das Zusammenwirken mit der nächsten Transaktion, und dies wird durch "Trace-Gleichungen" beschrieben.

5.5 OLDVERSION ()
pre version(current) > 0
traces ADDTRI(n,c,t); OLDVERSION = OLDVERSION
SHIFTACT(p); OLDVERSION = OLDVERSION
NEWVERSION; OLDVERSION = DUMMY

Bemerkung : DUMMY bezeichnet die triviale Transaktion ohne jede Wirkung. Die Gleichungen besagen z.B., daß nach dem Aufbau einer Szene mit der Transaktionsfolge

$$\tau = \tau_1; \tau_2; \ldots ; \tau_n$$

die Transaktionsfolge

$$\tau; \text{NEWVERSION}; \text{ADDTRI}(n_1,c_1,t_1); \text{SHIFTACT}(p_1); \text{SHIFTACT}(p_2); \text{ADDTRI}(n_2,c_2,t_2); \text{OLDVERSION}$$

die gleiche Wirkung haben soll wie z.B.

$$\tau; \text{NEWVERSION}; \text{ADDTRI}(n_1,c_1,t_1); \text{SHIFTACT}(p_1); \text{OLDVERSION}$$

oder wie

τ; NEWVERSION; OLDVERSION
= τ; DUMMY
= τ

Allgemein gilt für Folgen τ^* von Änderungstransaktionen :

τ = τ; NEWVERSION; τ^*; OLDVERSION

Insofern wird also die beabsichtigte Wirkung spezifiziert, daß OLDVERSION alle Änderungen bis zum letzten NEWVERSION rückgängig macht.

Bei unserer Spezifikation der Transaktionen haben wir noch nicht darauf geachtet, ob nicht Transaktionsfolgen, angewandt auf zulässige Zustände, unzulässige Zustandsfolgen produzieren können. Hierauf sollte in einem späteren Verfeinerungsschritt des Entwurfs eingegangen werden, in dem dann die Einhaltung der Integritätsbedingungen durch Verfeinern der Vor- und Nachbedingungen der Transaktionen soweit möglich zu gewährleisten ist. Mit dieser Thematik beschäftigt sich [Li85].

6. Schlußbemerkungen

Unser hier anhand eines Beispiels geschilderter Ansatz zur konzeptionellen Modellierung befindet sich noch in der Entwicklung. Manches Detail, insbesondere bei der Spezifikation der Transaktionen, bedarf der weiteren Ausgestaltung und Verfeinerung. Auch besteht noch eine Lücke bei der Darstellbarkeit partieller Information. Diese soll in künftigen Arbeiten geschlossen werden. Es genügt nicht, Partialität der Objektfunktionen zuzulassen, was ohne größere Schwierigkeiten möglich wäre. Hiermit ließen sich nur Sachverhalte der Art darstellen, daß eine Objektfunktion auf ein bestimmtes Objekt nicht anwendbar ist, z.B. das Attribut "Hubraum" auf ein Fahrrad. Die Darstellung fehlender oder unvollständiger Information, z.B. daß der Hubraum eines Autos zwischen 1600 und 2000 ccm liegt, bedarf jedoch einer wesentlichen Erweiterung des Ansatzes.

Besonderes Gewicht legen wir auf die saubere Klärung der semantischen Grundlagen unserer Konzepte. Die daraus resultierende Präzision und innere Konsistenz unserer Methodik erachten wir als wesentliche Vorraussetzung für die Entwicklung brauchbarer und zuverlässiger automatischer Hilfsmittel zur Unterstützung des Entwurfsvorgangs und zur Entwicklung einer korrekten und zuverlässigen Implementierung aus dem konzeptionellen Modell. Bislang teilt unser Ansatz die verbreitete Schwäche, daß keine implementierten Hilfsmittel existieren. Wir planen jedoch ein Forschungsprojekt, in dem derartige Hilfsmittel entwickelt und implementiert werden sollen.

Man kann sich fragen, ob unsere drei Schichten der Daten, Objekte und Transaktionen bereits den Gesamtbereich der konzeptionellen Modellierung ausschöpfen. Vorstellbar ist es, eine weitere Schicht darüberzulagern, in der dargestellt wird, wie bestimmte Ereignisse bestimmte Transaktionen anstoßen, deren Ausführung wieder zu neuen Ereignissen führt, die wiederum weitere Transaktionen anstoßen, u.s.w. Dies ist die natürliche Domäne der Petri-Netz-orientierten Modellierungsmethoden. Eine solche dynamische Gesamt-System-Schicht ließe sich auch über mehrere "lokale" Daten-Objekt-Transaktionen-Schemata legen, so daß sich auch verteilte Situationen modellieren ließen, in denen es lediglich lokale, nicht jedoch einen globalen Zustand gibt.

Literatur

BF79 Buneman, P. / Frankel, R.E. : FQL - A Functional Query Language. Proc. ACM SIGMOD Int. Conf. on Management of Data 1979, 52-58

BMS84 Brodie, M.L. / Mylopoulos, J. / Schmidt, J.W. (eds.) : On Conceptual Modelling. Springer-Verlag, New York 1984

BP77 Bartussek, A.W. / Parnas, D.L. : Using Traces to Write Abstract Specifications for Software Modules. UNC Report TR77-012, Univ. North Carolina, Chapel Hill, N.C., 1977

CCF82 Castilho, J.M.V. / Casanova, M.A. / Furtado, A.L. : A Temporal Framework for Database Specifications. Proc. 8th Int. Conf. on Very Large Data Bases, Mexico 1982

CF82 Casanova, M.A. / Furtado, A.L. : A Family of Temporal Languages for the Description of Transition Constraints. Proc. Workshop on Logical Bases for Data Bases, Toulouse 1982

Eh82 Ehrich, H.-D. : On the Theory of Specification, Implementation, and Parameterization of Abstract Data Types. Journ. ACM 29 (1982), 206-227

EGL84 Ehrich, H.-D. / Lipeck, U.W. / Gogolla, M. : Specification, Semantics, and Enforcement of Dynamic Database Constraints. Proc. 10th Int. Conf. on Very Large Data Bases, Singapore 1984

EM85 Ehrig, M. / Mahr, B. : Fundamentals of Algebraic Specification 1. Springer-Verlag, Berlin 1985

GMS83 Golshani, F. / Maibaum, T.S.E. / Sadler, M.R. : A Modal System of Algebras for Database Specification and Query /Update Language Support. Proc. 9th Int. Conf. on Very Large Data Bases, Florence 1983

Kl83 Klaeren, H. : Algebraische Spezifikation. Springer-Verlag, Berlin 1983

Ku84 Kung, C.H. A Temporal Framework for Database Specification and Verification. Proc. 10th Int. Conf. on Very Large Data Bases, Singapore 1984

LEG85 Lipeck, U.W. / Ehrich, H.-D. / Gogolla, M. :Specifying Admissibility of Dynamic Database Behaviours Using Temporal Logic. Proc. IFIP Working Conf. on Theoretical and Formal Aspects of Information Systems, A.Sernadas et al. (eds.), North Holland, Amsterdam 1985

Li85 Lipeck, U.W. : Schrittweise Spezifikation des dynamischen Verhalten von Datenbanken (dieser Tagungsband)

McL84 McLean, J. : A Formal Method for the Abstract Specification of Software. Journal ACM 31 (1984), 600-627

Se80 Sernadas, A. : Temporal aspects of Logical Procedure Definition. Inform. Sys. 5(1980), 167-187

Sh81 Shipman, D,W. : The Functional Data Model and the Data Language DAPLEX. ACM TODS 6(1981), 140-173