TECHNISCHE UNIVERSITÄT BRAUNSCHWEIG
INFORMATIK-BERICHTE
93–11

Beiträge zu KorSo– und TROLL *light*–Fallstudien

H.-D. Ehrich
(Herausgeber)

Abt. Datenbanken
Techn. Universität Braunschweig
Postfach 3329
D-38023 Braunschweig, Germany

Braunschweig

Dezember 1993

Vorwort

Ziel des BMFT–Verbundprojektes KorSo[1] ist es, angemessene Sprachen, Werkzeuge und Methoden für die Entwicklung nachweisbar korrekter Software zu entwickeln. Ob und inwieweit dies Ziel erreicht ist, muß man anhand von Fallstudien überprüfen. Neben einer Vielzahl kleinerer Fallstudien, die die einzelnen Projektpartner unabhängig voneinander durchführen, gibt es zwei größere, an denen jeweils mehrere Projektpartner mitwirken:

- Die Fallstudie „HDMS-A“ ist benannt nach dem heterogenen und verteilten Datenbanksystem, das am Deutschen Herzzentrum in Berlin entwickelt wird. Dies Beispiel ist zu groß und zu vielschichtig, um als ganzes in irgendeinem der vertretenen Ansätze behandelt werden zu können. So wählt jeder einen für die eigene Sprache und Methodik geeigneten Teilaspekt aus. Auf diese Weise erhält man Aufschluß darüber, wo die Stärken der jeweiligen Spezifikationssprachen und –paradigmen liegen und bekommt u.a. auch Hinweise darauf, wie sie sich in einer Anwendung miteinander kombinieren lassen.

- Die Fallstudie „Fertigungszelle“ dient vor allem dazu, unterschiedliche Ansätze am gleichen Beispiel zu erproben und miteinander zu vergleichen: mehrere Arbeitsgruppen beschreiben die Fertigungszelle unabhängig voneinander jeweils auf ihre Weise mit ihrer Sprache und mit ihrer Methodik. Dabei werden neben den von den Projektpartnern entwickelten Sprachen auch andere verwendet.

In diesem Band haben wir einige Arbeiten der Braunschweiger KorSo–Gruppe zusammengestellt, die innerhalb von KorSo–Fallstudien oder als interne Studien entstanden sind. Vier der fünf in diesem Bericht zusammengefaßten Fallbeispiele verwenden die hier entwickelte Objekt-Spezifikationssprache TROLL *light*.

Die ersten beiden Beiträge betreffen die HDMS-A–Fallstudie. Im dritten Beitrag wird die Fertigungszelle modelliert. Im vierten Beitrag zeigen wir, wie man einen Template-Katalog mit TROLL *light* spezifizieren kann. Der Katalog dient dazu, Objektbeschreibungen innerhalb einer Entwicklungsumgebung strukturiert abzulegen. Anschließend stellen wir einige kleinere Beispiele zusammen, von denen die meisten der Literatur entnommen sind und so den Vergleich mit anderen Spezifikationsansätzen zulassen.

Nicht alle hier behandelten Beispiele entstammen dem Anwendungsbereich „Informationssysteme“, auf den hin TROLL *light* entworfen wurde. Diese untypischen Beispiele zeigen, daß TROLL *light* breiter verwendet werden kann, als es zunächst geplant war. Daraus soll jedoch kein universeller Anspruch erhoben werden: die Frage, wo man objektorientierte Ansätze im allgemeinen — und TROLL *light* im besonderen — anwenden *sollte*, ist damit nicht beantwortet, sie ist auch nicht Gegenstand dieses Berichts.

H.-D. Ehrich

[1] BMFT–Verbundprojekt KorSo(= „KOrrekte SOftware“), Förderungsnummer 01 IS 203 D.

Inhaltsverzeichnis

Einbindung eines bestehenden Datenbanksystems in einen formalen Software–Entwicklungsprozeß

— ein Beitrag zur HDMS-A–Fallstudie —

Stefan Conrad

Technische Universität Braunschweig
Informatik, Abt. Datenbanken
D–38092 Braunschweig

e-mail: conrad@idb.cs.tu-bs.de

Zusammenfassung

Dieser Beitrag beschäftigt sich mit dem Problem, wie ein bestehendes Datenbanksystem innerhalb eines formalen Software–Entwicklungsprozeß berücksichtigt werden kann. Konkret betrachten wir den Fall, daß ein relationales Datenbanksystem vorliegt. Dazu ist zunächst ein relationaler Datenbankentwurf durchzuführen, der die formal spezifizierte Welt repräsentiert. Ein Schwerpunkt der weiteren Betrachtungen ist die Behandlung von Integritätsbedingungen, die in der formalen Spezifikation formuliert wurden.

1 Einführung in die Problemstellung

Ein besonderes Problem der formalen Software–Entwicklung ist die Einbindung bereits bestehender Subsysteme. In der Regel sind solche Systeme nicht formal entwickelt worden, so daß geeignete Spezifikationen für sie nicht vorhanden sind. Eine Möglichkeit, dieses Manko zu beheben, wäre eine vollständige formale Nachentwicklung. Allerdings ist der Aufwand dafür meistens nicht vertretbar. Die Verwendung bestehender Subsysteme soll ja gerade die Entwicklungskosten in Grenzen halten. Eine vollständige formale Nachentwicklung ist aber in Hinblick auf ihren Arbeitsaufwand kaum von einer formalen Neuentwicklung zu unterscheiden.

Ohne formale Spezifikation eines bestehenden Subsystems bleibt als Alternative nur ein Vorgehen, das die Einbindung dieses Subsystems in den Entwicklungsprozeß derart berücksichtigt, daß die funktionalen Anforderungen an dieses System mit der tatsächlich vorhandenen Funktionalität des Systems in Übereinstimmung gebracht werden. Dieses Vorgehensweise werden wir im folgenden skizzieren. Dazu betrachten wir die HDMS-A–Fallstudie [CKL92, LCFW92], die im Rahmen des KorSo–Projektes

[Bro92] die Verwendung formaler Methoden in der Software–Entwicklung anhand einer realen Aufgabenstellung aufzeigen soll. Ein Aspekt dieser Fallstudie ist die Einbeziehung bestehender Datenbanksysteme in die formale Entwicklung.

Im folgenden werden wir annehmen, daß eine abstrakte, funktionale und formale Spezifikation des zu entwerfenden Gesamtsystems gegeben ist (im folgenden auch „funktionale Essenz"–Spezifikation genannt, siehe [SNM+93]). Die Realisierung dieses Gesamtsystems soll nun ein relationales Datenbanksystem einbeziehen, das schon vorher vorhanden war bzw. zur Senkung der Entwicklungskosten gekauft wurde. Für die Daten, die später von dem Datenbanksystem verwaltet werden sollen, ist ein konzeptioneller Entwurf auf der Grundlage des ER–Modells durchgeführt worden (vgl. [SNM+93]). Ferner besteht eine allgemeine, formale Übersetzung für ER–Schemata in algebraische Spezifikationen [Het93, für Spectrum]. Unter diesen Vorgaben werden wir nun einen relationalen Datenbankentwurf exemplarisch durchführen. Anschließend werden wir an einem Ausschnitt aus der „funktionalen Essenz"–Spezifikation die Verwendung und Berücksichtigung der relationalen Datenbank in der Implementierung der abstrakten, funktionalen Spezifikation demonstrieren.

2 Skizze des relationalen Datenbankentwurfs

Zur Modellierung des *Universe Of Discourse* (UoD) ist in [SNM+93] das ER–Modell in seiner Basisversion (vgl. [Che76]) verwendet worden. In [Het93] ist eine Übersetzung solcher ER–Schemata nach Spectrum entwickelt worden. Im folgenden werden wir uns auf diese Übersetzung beziehen und einen dazu passenden, relationalen Datenbankentwurf skizzieren. Hierfür sind folgende Schritte durchzuführen:

1. Generierung der Relationenschemata aus der Spectrum–Spezifikation,
2. Umsetzung der elementaren Datenmanipulationsoperationen in SQL,
3. Umsetzung der Integritätsbedingungen.

Zuvor wollen wir aber die wichtigsten Spezifikationselemente aufführen, die bei der Übersetzung von ER–Schemata in Spectrum–Spezifikationen erzeugt werden. Zunächst einmal gibt es zu jedem Entitytyp eine Sorte E und für die Attribute zu einem Entitytyp geeignete Sorten $A_1, \ldots, A_n$. Für die erlaubten Datenbankzustände gibt es eine Sorte Db. Die Beziehungstypen werden als Komponenten von Datenbankzuständen modelliert. Werte der Sorte Db werden durch Mengen von Werten zu den Entitytypsorten und durch Relationen über diesen Entities bestimmt.

Zu jedem Entitytyp gibt es Operationen zum Einfügen und Löschen einzelner Entities und zum Verändern der Attributwerte. Für jeden Beziehungstyp gibt es je eine Operation zum Herstellen einer Beziehung zwischen geeigneten Entities und zum Wiederaufheben solcher Beziehungen.

2.1 Generierung der Relationenschemata

Die Übersetzung eines ER–Schemas in das Relationenmodell bedingt die Erzeugung von Relationenschemata für Entitytypen als auch für Beziehungstypen aus dem ER–Schema.

Ein Relationenschema Rel$E$ für einen Entitytyp E mit Attributen $attr_1, \ldots, attr_n$ der Sorten $A_1, \ldots, A_n$ ist gegeben durch:

$$\mathrm{Rel}E(attr_1 : A_1, \ldots, attr_n : A_n)$$

Dieses läßt sich direkt in SQL als eine Anweisung zur Erzeugung einer neuen Tabelle formulieren:

$$\begin{array}{ll} \texttt{create table} & \texttt{Rel}E \\ & \texttt{(}\ attr_1 \quad A_1\,\texttt{,} \\ & \quad \vdots \\ & \quad attr_n \quad A_n\,\texttt{);} \end{array}$$

Für nicht–optionale Attribute $attr_i$ ist dabei jeweils noch die Option „`not null`“ hinter der Attributsorte A_i anzugeben.

Für jede Relation R des ER–Schemas, an der zwei Entitytypen E_1 und E_2 mit Schlüsselattributen $attr_{E_1,1}, \ldots, attr_{E_1,k_1}$ bzw. $attr_{E_2,1}, \ldots, attr_{E_2,k_2}$ beteiligt sind, läßt sich ein entsprechendes Relationenschema Rel$R$ angeben (dieses läßt sich natürlich auch auf mehr als zwei beteiligte Entitytypen verallgemeinern):

$$\mathrm{Rel}R(attr_{E_1,1} : A_{E_1,1}, \ldots, attr_{E_1,k_1} : A_{E_1,k_1}, attr_{E_2,1} : A_{E_2,1}, \ldots, attr_{E_2,k_2} : A_{E_2,k_2})$$

Auch hierfür sei wieder eine entsprechende SQL–Formulierung zur Erzeugung dieser Relation angegeben:

$$\begin{array}{ll} \texttt{create table} & \texttt{Rel}R \\ & \texttt{(}\ attr_{E_1,1} \quad A_{E_1,1}\,\texttt{,} \\ & \quad \vdots \\ & \quad attr_{E_1,k_1} \quad A_{E_1,k_1}\,\texttt{,} \\ & \quad attr_{E_2,1} \quad A_{E_2,1}\,\texttt{,} \\ & \quad \vdots \\ & \quad attr_{E_2,k_2} \quad A_{E_2,k_2}\,\texttt{);} \end{array}$$

2.2 Umsetzung der elementaren Datenmanipulationsoperationen in SQL–Statements

Zu den zuvor erzeugten Relationenschemata werden noch elementare Datenmanipulationsoperationen benötigt. Im folgenden werden wir für die in [Het93] in Spectrum spezifizierten elementaren Operationen für Entitytypen und Beziehungstypen SQL–Statements angegeben, die zu den zuvor erzeugten Relationenschemata passen.

2.2.1 Elementare Operationen für Entities

Zunächst betrachten wir die elementaren Operationen auf den Entitytypen. Für jeden Entitytyp E mit Attributen $attr_1, \ldots, attr_n$ ist eine n–stellige Operation create$E$: $A_1 \times \ldots \times A_n \rightarrow E$ gegeben, die ein neues Entity des Typs E erzeugt und seine Attribute mit Werte versieht. Mit put$E$: $E \times Db \rightarrow Db$ wird dieses Entity in die Datenbank eingefügt. Die folgende SQL–Anweisung realisiert nun die zusammengesetzte Operation put$E$(create$E(a_1, \ldots, a_n), Db$):[2]

```
insert into  RelE
     values  (a1, ..., an);
```

Ferner gibt es für jedes Attribut $attr_i$ eines Entitytyps E eine Operation setattr$_i E$: $E \times A_i \rightarrow E$, die dem Attribut einen neuen Wert zuweist. Eine SQL–Anweisung, die den Operationsaufruf setattr$_i E$(create$E(a_1, \ldots, a_n), a$) realisiert, ist:

```
update  RelE
   set  attri = a
 where  attr1 = a1 and ...and attrn = an ;
```

Zum Löschen eines Entities aus einer Relation Rel$E$ gibt es eine Operation del$E$: $E \times Db \rightarrow Db$. Diese läßt sich wie folgt in SQL realisieren (für den Aufruf del$E$(create$E(a_1, \ldots, a_n)$)):

```
delete from  RelE
      where  attr1 = a1 and ...and attrn = an ;
```

Für die letzten beiden Operationen reicht es auch aus, in der `where`–Klausel nur die Schlüsselattribute zu betrachten. Die Schlüsseleigenschaft garantiert dann, daß auch nur das durch die Werte der Schlüsselattribute bestimmte Entity betroffen ist.

Zum Auslesen eines Entities des Sorte E benötigen wir noch eine Funktion get$E$: $keyE \times Db \rightarrow E$, die sich durch eine einfache SQL–Anweisung realisieren läßt (seien $attr_{k_1}, \ldots, attr_{k_i}$ die Schlüsselattribute des betrachteten Entitytyps, und seien $a_{k_1}, \ldots, a_{k_i}$ die bei Aufruf der Funktion übergebenen Werte zu diesen Schlüsselattributen):

```
select  attr1,  ...,  attrn
  from  RelE
 where  attrk1 = ak1 and ...and attrki = aki ;
```

2.2.2 Elementare Operationen für Beziehungen zwischen Entities

Zum Herstellen und Auflösen von Beziehungen zwischen Entities in einer Datenbank werden folgende Funktionen für jede Relation R in der Spectrum–Spezifikation be-

[2]Wir verwenden hier und im folgenden solche zusammengesetzten Operationen, um eine einfache und verständliche Darstellung zu erhalten. Die gewählte Darstellung läßt sich durch Ersetzen des Terms create$E(a_1, \ldots, a_n)$ durch eine Variable *Entity* (der Sorte E) und Ersetzen der Attributwerte a_i durch $Attr_i(Entity)$ geeignet umformen, wobei $Attr_i : E \rightarrow A_i$ in [Het93] zu einem Entity den entsprechenden Attributwert bestimmt.

reitgestellt (E_1 und E_2 sind die beteiligten Entitytypen):

$\text{est}R : Db \times E_1 \times E_2 \to Db;$

$\text{rel}R : Db \times E_1 \times E_2 \to Db;$

Für diese beiden Arten von Operationen lassen sich folgende SQL–Statements als Entsprechung angeben (wobei $attr_{E_1,1}, \ldots, attr_{E_1,k_1}$ und $attr_{E_2,1}, \ldots, attr_{E_2,k_2}$ die Schlüsselattribute der Entitytypen E_1 bzw. E_2 seien):

- für est$R$-Operationen
  mit Aufruf est$R$(get$E_1(a_{1,1}, \ldots, a_{1,k_1})$, get$E_2(a_{2,1}, \ldots, a_{2,k_2})$):

  `insert into` `Rel`R
  `values` $(a_{1,1},\ \ldots,\ a_{1,k_1},\ a_{2,1},\ \ldots,\ a_{2,k_2})$ `;`

- für rel$R$-Operationen
  mit Aufruf rel$R$(get$E_1(a_{1,1}, \ldots, a_{1,k_1})$, get$E_2(a_{2,1}, \ldots, a_{2,k_2})$):

  `delete from` `Rel`R
  `where` $attr_{E_1,1} = a_{1,1}$ `and` ... `and` $attr_{E_1,k_1} = a_{1,k_1}$
  `and` $attr_{E_2,1} = a_{2,1}$ `and` ... `and` $attr_{E_2,k_2} = a_{2,k_2}$ `;`

2.3 Umsetzung der Integritätsbedingungen

Betrachtet man die Axiome der Spectrum–Spezifikation, so fällt auf, daß viele dieser Axiome auf einfache Weise von dem Datenbanksystem garantiert werden können. So kann z.B. die Erhaltung der Schlüsselintegrität durch Einrichten von geeigneten Indexen über die jeweiligen Schlüsselattribute der Relationen vom System geprüft werden, und Operationen, die die Schlüsselintegrität verletzen, können so zurückgewiesen werden.

Auch referentielle Integrität, d.h., jedes Entity, das an einer Beziehung beteiligt ist, existiert auch, kann zumeist vom Datenbanksystem überwacht werden. Dazu müssen dann Fremdschlüsselbeziehungen zwischen jeder Relation, die einen Beziehungstyp repräsentiert, und den Relationen, die die an dem Beziehungstyp beteiligten Entitytypen repräsentieren, hergestellt werden.

Inwieweit weitere Integritätsbedingungen direkt durch das Datenbanksystem überwacht werden können, hängt maßgeblich von den Möglichkeiten des jeweiligen Systems ab. Alle Teilaufgaben, die das Datenbanksystem leisten kann, sollten nicht außerhalb des Datenbanksystems realisiert werden, da dann in der Regel starke Effizienzverluste zu erwarten sind. Alle verbleibenden Bedingungen (resp. Axiome der Spectrum–Spezifikation) müssen im Laufe der Implementierung explizit außerhalb des Datenbanksystems realisiert werden. Dies dürfte insbesondere auf die Bedingungen zutreffen, die von der Modellierung abhängen: z.B. Beschränkungen der Grade von Beziehungen, sich gegenseitig ausschließende Teilnahme an verschiedenen Beziehungen, etc.

Um die Einhaltung solcher Integritätsbedingungen gewährleisten zu können, wird es dann notwendig sein, eine abstrakte Operation auf der Datenbank (in [SNM+93] auch elementare Transaktionen genannt) in der Implementierung durch eine Transaktion zu realisieren. Dadurch kann zum einen die abstrakte Operation durch eine Folge konkreter Anweisungen auf der relationalen Datenbank realisiert werden. Zum anderen kann und muß innerhalb der Transaktion (in der Regel vor Abschluß der Transaktion) explizit überprüft werden, ob Integritätsbedingungen verletzt wurden, die nicht vom Datenbanksystem selbst überprüft werden können. Diese explizite Überprüfung wird im allgemeinen weitere (Lese-) Operationen auf der Datenbank erfordern. Damit kann folgendes allgemeines Implementierungsschema für abstrakte Operationen (aus der „funktionalen Essenz"–Spezifikation) angegeben werden: `begin transaction;`

```
     ⋮
     SQL–Statements        Realisierung der abstrakten Operation
     ⋮
     SQL–Statements        explizite Integritätsprüfung
     ⋮
     if  Integrität erhalten
          then   commit_transaction
          else   abort_transaction
     endif;
end;
```

In Abhängigkeit davon, ob durch Ausführung der abstrakten Operation (realisiert durch eine Folge von SQL–Anweisungen) irgendeine Integritätsregel verletzt wurde, wird die Transaktion abgebrochen, d.h. es wird der zu Beginn der Transaktion bestehende Datenbankzustand wiederhergestellt, oder erfolgreich abgeschlossen. Ziel der formalen Software–Entwicklung sollte aber sein, durch geeignete, korrekte Implementierung Transaktionsabbrüche wegen Verletzung von Integritätsbedingungen zu vermeiden.

3 Anbindung an die funktionale Spezifikation

In diesem Abschnitt wollen wir einen Ausschnitt aus der „funktionalen Essenz"–Spezifikation betrachten. Dabei soll diskutiert werden, in welcher Form Axiome der funktionalen Spezifikation behandelt werden können, wenn wir in einem Entwicklungsschritt von einer abstrakten Datenbank zu einer konkreten übergehen.

Der Kern der Anbindungsproblematik ist darin zu sehen, daß in der funktionalen Spezifikation „naturgemäß" Bedingungen als first-order-Formeln notiert werden, während bestehende Datenbanksysteme in der Regel nicht in der Lage sind, beliebige first–order–Integritätsbedingungen zu überwachen. Daher ist es notwendig, diese Bedingungen derart zu zerlegen, daß Teile, die von der Integritätsüberwachung des Datenbanksystems behandelt werden können, ausgegliedert werden können. Für die übrigbleibenden Teilbedingungen muß dann in dem durch weitere Verfeinerungsschritte entstehenden Datenbankanwendungsprogramm die Integritätserhaltung garantiert werden (dies bedeutet, daß auf dem Wege zu einer lauffähigen Implementierung explizite Integritätstests eingeführt werden müssen; dies wurde schon im vorherigen Abschnitt diskutiert).

Unterschieden werden muß an dieser Stelle aber zwischen

- statischen Integritätsbedingungen und
- transitionalen Integritätsbedingungen.

Statische Integritätsbedingungen werden unabhängig von möglichen Aktionen auf der Datenbank angegeben. Sie müssen für jeden Datenbankzustand erfüllt sein (ggfs. mit der Ausnahme von Zwischenzuständen innerhalb von Transaktionen). In der funktionalen Spezifikation sind diese Integritätsbedingungen auf verschiedene Teilspezifikationen verteilt. Konjunktiv zusammengefaßt werden diese Bedingungen durch das OK–Prädikat. Dieses Prädikat ist genau für die Datenbankzustände erfüllt, die alle statischen Integritätsbedingungen erfüllen.

Transitionale Integritätsbedingungen gehen über die statischen insofern hinaus, daß sie für einzelne Operationen auf der Datenbank zusätzliche Einschränkungen treffen. Dies bedeutet, daß eine Operation auf der Datenbank nicht nur einen (beliebigen) neuen Zustand erzeugen soll, der die statischen Integritätsbedingungen erfüllt, sondern es kann zusätzlich eingeschränkt werden, welche Veränderungen in der Datenbank durch diese Operation geschehen dürfen.

Als kleines Beispiel sei hier die elementare Transaktion „Eintragen des kleinen Blutbildes in die Datenbank“ (enterKlBlutbild) aus dem Ablauf „Blutbilduntersuchung“ betrachtet (siehe [MN93]), deren Spectrum–Spezifikation wie folgt aussieht:

enterKlBlutbild: Db $\times$ AuftragNr $\times$ Status $\times$ KlBlutbildWert $\times$ LaborantId $\rightarrow$ Db;
enterKlBlutbild **strict;**

axioms
δ(enterKlBlutbild(db, nr, stat, bbw, lid)) $\Longleftrightarrow$
    δ(getLaborAuftrag(nr, db)) $\wedge$ δ(getLaborant(lid, db))
$\wedge$ ($\neg$ $\exists$ bb: KlBlutbild. (KlBb db (a, bb)));

δ(enterKlBlutbild(db, nr, stat, bbw, lid)) $\Rightarrow$
let bb =
    if stat <> unverwertbar **then**
        createKlBlutbild(nr, stat, bbw, time(db))
    else createKlBlutbild(nr, stat, UNDEF, time(db))
and a = getLabAuftrag(nr, db)
and laborant = getLaborant(lid, db)
and db’ = putKlBlutbild(bb, db)
in
    enterKlBlutbild(db, nr, stat, bbw, lid) =
    estKlbb(estFreigabeKl(db’, bb, laborant), a, bb) **endlet**
endaxioms

In den Axiomen ist deutlich zu erkennen, daß diese elementare Transaktion aus einer Reihe von Datenbankoperationen besteht. So werden hier die folgenden elementaren

Datenmanipulationsoperationen benutzt: getLaborAuftrag, getLaborant, putKlBlutbild, estKlbb und estFreigabeKl. Ferner sind noch einige Hilfsfunktionen (z.B. createKlBlutbild) notwendig.

Eine Realisierung unter Verwendung der im vorigen Kapitel angegebenen SQL–Anweisungen könnte folgende Gestalt haben (unter Auslassung vieler Details; eine konkretere Angabe der Transaktion findet sich in Anhang 1):

```
begin transaction enterKlBlutbild;
      ⋮
      SQL–Anweisung für getLabAuftrag
      SQL–Anweisung für getLaborant
      if  beide Ergebnisse definiert   -- erstes Axiom
          then  SQL–Anweisung für putBlutbild
                SQL–Anweisung für estFreigabeKl
                SQL–Anweisung für estKlbb
                if  Integrität erhalten
                    then   commit_transaction
                    else   abort_transaction
                endif;
          else  abort_transaction
      endif;
end;
```

Die abschließende Integritätsabfrage ist dann notwendig, wenn das verwendete Datenbanksystem nicht in der Lage ist, alle statischen Integritätsbedingungen selbst zu prüfen (und dies ist bei den zur Zeit kommerziell erhältlichen Systemen leider zu erwarten). Zu jeder SQL–Anweisung sind abhängig von der Einbettungsart in die Wirtssprache noch weitere Programmschritte notwendig.

4 Diskussion

Abschließend wollen wir zunächst die hier gewählte Vorgehensweise diskutieren und mit eventuellen Alternativen vergleichen. Danach werden wir noch kurz in allgemeiner Form die vorgegebene ER–Modellierung und die Übersetzung nach Spectrum hinterfragen.

4.1 Diskussion der Vorgehensweise

Die hier vorgestellte Vorgehensweise ist sicherlich nicht die einzig mögliche. Ihr besonderes Merkmal ist ihre pragmatische Ausrichtung. Im Gegensatz dazu kann man sich aber auch vorstellen, ein bestehendes Datenbanksystem als solches zu spezifizieren (dazu siehe auch [Con93]). Dies würde bedeuten, daß neben dem Datenmodell des Datenbanksystems auch Managementkomponenten, wie z.B. Transaktionsverwaltung und Anfrageauswertung, formal spezifiziert werden müßten. Allerdings handelt man sich dabei eine Reihe von Problemen ein. Insbesondere die verschiedenen Formen der

Generizität, die im Zusammenhang mit einem Datenbanksystem auftreten, sind formal nur unbefriedigend in den Griff zu bekommen. Eines dieser Probleme ist, daß man dann das Datenbanksystem schemaunabhängig beschreiben und alle Möglichkeiten der Schemaerzeugung und Schemamodifikation berücksichtigen muß. Daraus resultiert dann auch, daß für alle Datenbankmanagementkomponenten Sprachen mitspezifiziert werden müssen, die ebenfalls zunächst schemaunabhängig sind. Als Beispiele seien hier die Anfrageauswertung und Integritätsüberwachung genannt:

- Für die Anfrageauswertung benötigt man zunächst eine Anfragesprache. Diese muß für alle möglichen Schemata geeignet sein. Zum Zeitpunkt der Anfrageauswertung muß dann die konkrete Anfrage mit dem aktuellen Datenbankschema verglichen werden, um feststellen zu können, ob diese Anfrage überhaupt sinnvoll ist (und nicht auf etwas zugreifen will, das im Datenbankschema überhaupt nicht vorhanden ist).

- Für die Integritätsüberwachung ist eine Sprache zur Formulierung der Integritätsbedingungen notwendig. Diese Sprache muß dann zur Laufzeit interpretiert werden können, um neue Integritätsbedingungen bearbeiten zu können. In einer solchen Sprache formulierte Integritätsbedingungen müssen dann auch mit dem aktuellen Schema verglichen werden, und ihre Erfüllung muß nach jeder Datenmanipulation (bzw. Transaktion) überprüft werden.

Für andere Komponenten gibt es ähnliche Forderungen. Damit würde aber eine vollständig formale Spezifikation eines Datenbanksystems sehr umfangreich werden. Darüberhinaus hat der Versuch, ein einfaches Datenbanksystem formal zu spezifizieren (siehe [Con93]), uns auch die Erkenntnis gebracht, daß die bestehenden formalen Mittel (insbesondere Spezifikationssprachen) die hier unbedingt notwendige Generizität nur in unbefriedigender Weise darstellen lassen, d.h., hier müssen oft schon auf der Ebene der abstrakten Spezifikation Implementierungsdetails simuliert werden.

Diese Probleme haben uns dazu veranlaßt, das in diesem Papier beschriebene pragmatische Verfahren zu betrachten. Hier scheint der Aufwand in einem sinnvollen Rahmen zu stehen. Allerdings entstehen neue bzw. andere Probleme. Insbesondere die Behandlung von Integritätsbedingungen ist abhängig von den Möglichkeiten des verwendeten Datenbanksystems. Aber auch in Hinblick auf Korrektheit entstehen Probleme:

- Ist die Integritätsüberwachung des Datenbanksystems korrekt?

  Dies ist eine Voraussetzung dafür, daß wir Teilaufgaben an das Datenbanksystem abgeben dürfen.

- In welcher Form kann gezeigt werden, daß die konkret gewählte Aufgabenverteilung in bezug auf Integritätsüberwachung zwischen Datenbanksystem und expliziter Kontrolle in programmierten Transaktionen die volle Integrität (gemäß „funktionaler Essenz“–Spezifikation) garantiert?

Diese Fragen entziehen sich hier allerdings einer vollständig formalen Betrachtungsweise, da das Datenbanksystem, und damit die Integritätsüberwachungskomponente, selbst nicht formal beschrieben wird.

4.2 Zur Datenmodellierung

Für die Datenmodellierung sind in [SNM+93] ER–Diagramme gewählt worden. Dabei wurden nur Konzepte des Basis–ER–Modells [Che76] verwendet. Hierzu sind zwei Anmerkungen notwendig:

- Allgemein stellt sich die Frage, ob die Verwendung des „Basis"–ER–Modells adäquat ist. Da das verwendete ER–Modell nur minimal in seinen Konzepten ist, besteht die Gefahr, daß Zusammenhänge aus der realen Welt auf die wenigen Konzepte heruntergebrochen werden. Für komplexere Weltausschnitte eignen sich daher zur Modellierung erweiterte ER–Modelle bzw. semantische Datenmodelle sicher besser. In Hinblick auf die „funktionale Essenz"–Spezifikation bietet es sich somit an zu prüfen, ob an verschiedenen Stellen nicht abstrakter modelliert werden kann.
- Die Verwendung des „Basis"–ER–Modells bedingt eine offensichtliche Transformation ins Relationenmodell. Dadurch läßt sich das Ergebnis der Transformation von ER nach SPECTRUM (siehe [Het93]) in trivialerweise Weise z.B. in SQL–Statements überführen. Dies gilt für alle durch die Transformation entstandenen primitiven Datenmanipulationsoperationen.

Während prinzipiell ein Datenmodell zu bevorzugen ist, das eine bessere Abstraktion erlaubt, hätte dies aber auch zur Konsequenz, daß der Übersetzungsprozeß z.B. ins Relationenmodell für ein bestehendes Datenbanksystem aufwendiger werden würde.

4.3 Zur Übersetzung von ER–Schemata nach SPECTRUM

In [Het93] wurde die für den Datenbankbereich schon klassische Übersetzung der Konzepte des Basis–ER–Modells in Konzepte des Relationenmodells für die Spezifikationssprache SPECTRUM nachempfunden. Während diese Übersetzung auf den ersten Blick keine Überraschungen enthält, wirft eine nähere Betrachtung die Frage auf:

> Sollte die Übersetzung nicht von einigen „Implementierungsdetails" abstrahieren?

Insbesondere bei der Betrachtung von Beziehungstypen in der entsprechenden SPECTRUM–Spezifikation fällt auf, daß die Beziehung über Primärschlüssel der beteiligten Entitytypen realisiert wird. Dies kann aber als eine „Implementierungsentscheidung" angesehen werden, die auf die spätere Realisierung durch ein relationales Datenbanksystem abgestimmt ist. Statt ihrer Schlüssel könnten auch die Entities selbst an der Beziehung teilnehmen. Damit würde die SPECTRUM–Spezifikation die Konzepte des ER–Modells besser widerspiegeln. Ferner stellt sich sogar die Frage, ob in jedem Fall explizite Primärschlüssel vorhanden sein müssen. Es gibt durchaus Anwendungsbereiche, für die dies als nicht notwendig angesehen wird (z.B. im CAD–Bereich).

1 Detailliertere Betrachtung einer Transaktion

Hier wollen wir die Transaktion, die wir für das „Eintragen des kleinen Blutbildes“ in Abschnitt 3 schematisch skizziert haben, konkretisieren. Dazu werden wir ein *embedded SQL* benutzen, wie es z.B. für das Datenbanksystem INGRES (Release 6.4) angeboten wird. Als Wirtssprache legen wir die Sprache C zugrunde. Der Übersichtlichkeit halber haben wir einige programmiertechnische Details ausgelassen (z.B. Deklarationen von Variablen).

Bevor wir die Transaktion selbst betrachten, seien zum Verständnis noch die verwendeten Relationen deklariert:

```
/********************
*  Die Entitytypen  *
********************/
  EXEC SQL DECLARE LaborAuftrag TABLE
       (ANr                 int         not null, /* Schlüsselattribut */
        ErstellungsZeitpkt  varchar(20) not null,
        Differw             bool        not null,
        Laboreingang        varchar(20)          );

  EXEC SQL DECLARE Laborant TABLE
       (LaborantId          int         not null, /* Schlüsselattribut */
        Name                varchar(20) not null,
        Adresse             varchar(50) not null,
        Einstellung         varchar(20) not null,
        Ausstellung         varchar(20)          );

  EXEC SQL DECLARE KlBlutbild TABLE
       (ANr                 int         not null, /* Schlüsselattribut */
        Status              varchar(20) not null,
        Wert                varchar(50),
        Zeitpunkt           varchar(20) not null );

/************************
*  Die Beziehungstypen  *
************************/
  EXEC SQL DECLARE FreigabeKl TABLE
       (LaborantId          int         not null,
        KlBlutbildNr        int         not null ); /* ANr des Auftrags */

  EXEC SQL DECLARE KlBb TABLE
       (ANr                 int         not null,
        KlBlutbildNr        int         not null ); /* ANr des Auftrags */
```

Nun können wir die Transaktion angeben (dabei werden die folgenden Variablen verwendet: `nr`, `stat`, `bbw`, `lid`, `ANr`, `LabId` und `time`; werden diese Variablen in den eingebetteten SQL–Anweisungen benutzt, so wird ihnen ein ":" vorangestellt):

```
/*******************************************************
*           transaction  enterKlBlutbild              *
*******************************************************/

void enterKlBlutBild(nr,stat,bbw,lid)
{
  /*   Deklaration und Initialisierung lokaler Variablen  */
  ⋮
  /*
  ** Deklaration der Cursor zum Suchen in den Relationen
  */
  EXEC SQL DECLARE labauf CURSOR FOR
    SELECT   ANr
    FROM     LaborAuftrag
    WHERE    ANr = :nr     ;
  EXEC SQL DECLARE lab CURSOR FOR
    SELECT   LaborantId
    FROM     Laborant
    WHERE    LaborantId = :lid ;

  /*
  ** Öffnen der Datenbank 'hdms-a'
  */
  EXEC SQL CONNECT hdms-a;

  /*
  ** Öffnen der beiden Cursor (Ausführen der Anfragen)
  */
  EXEC SQL OPEN labauf;
  EXEC SQL OPEN lab;

  /*
  ** für den Fehlerfall Transaktionsabbruch
  */
  EXEC SQL WHENEVER NOT FOUND  STOP;

  EXEC SQL FETCH labauf
    INTO :ANr;

  EXEC SQL FETCH lab
    INTO :LabId;

  /*
  ** wenn beide Ergebnisse definiert sind, dann weiter !!
  ** sonst wurde automatisch die Transaktion abgebrochen !!
  */
```

```
/*
**  SQL–Anweisung für  putBlutbild
*/
time = system_time();
if( stat <> "unverwertbar" )
  {
    EXEC SQL INSERT INTO  KlBlutbild
                    VALUES (:ANr, :stat, :bbw, :time) ;
  };
else
  {
    EXEC SQL INSERT INTO  KlBlutbild
                    VALUES (:ANr, :stat, UNDEF, :time) ;
  };
/*
**  SQL–Anweisung für  estFreigabeKl
*/
EXEC SQL INSERT INTO  FreigabeKl  VALUES (:ANr, :LabId);

/*
**  SQL–Anweisung für  estKlbb
*/
EXEC SQL INSERT INTO  KlBb  VALUES (:nr, :ANr);

/*
**  abschließender Integritätstest
*/
if  ( db_OK() ) /*  Integrität erhalten ??
                **   diese Funktion prüft die Integritätsbedingungen,
                **   die nicht vom Datenbanksystem bearbeitet
                **   werden können
                */
  {
    /***  commit_transaction  ***/
    /*
    ** erfolgreicher Transaktionsabschluß
    */
    EXEC SQL COMMIT;
  };
else
  {
    /***  abort_transaction   ***/
    /*
    ** Rollback (Wiederherstellung des Ausgangszustandes) anstoßen:
    */
    EXEC SQL ROLLBACK;
  };
```

```
  EXEC SQL DISCONNECT;
};   /*  end of  'enterKlBlutbild'  */
```

Anmerkungen:

Gegenüber der abstrahierten Fassung aus Abschnitt 3 haben sich einige strukturelle Änderung ergeben. Diese beruhen im wesentlichen darauf, daß der Erfolg von Anfragen nicht explizit getestet werden muß, sondern durch `EXEC SQL WHENEVER NOT FOUND ...` vorab eine Fehlerbehandlung festgelegt werden kann (hier der Transaktionsabbruch).

Bei genauer Betrachtung erkennt man, daß die Relation `KlBb` hier überflüssig ist (beim Einfügen gilt immer `nr=ANr`). Wir haben sie hier zum besseren Vergleich mit der Spezifikation und dem abstrakten Schema nicht eliminiert.

Danksagung: Besonderer Dank gebührt Rudolf Hettler (TU München) und Michael Löwe (TU Berlin), die durch ihre sorgfältige Begutachtung wesentlich zur Verbesserung des vorliegenden Beitrages beigetragen haben.

Literaturverzeichnis

[Bro92] M. Broy. KORSO: Beschreibung der Gesamtziele. Tum-info, Institut für Informatik, TU München, Juni 1992.

[Che76] P. Chen. The Entity-Relationship Model – Towards a Unified View of Data. *ACM Trans. on Database Systems*, 1(1):9–36, 1976.

[CKL92] F. Cornelius, M. Klar, and M. Löwe. Ein Fallbeispiel für KORSO: Ist-Analyse für HDMS-A. Technischer Bericht 93-28, TU Berlin, 1992.

[Con93] S. Conrad. Spezifikation eines vereinfachten Datenbanksystems — ein Beitrag zur HDMS-A–Fallstudie. In H.-D. Ehrich, editor, *Beiträge zu* KorSo– *und* TROLL *light–Fallstudien*, pages 15–26. Technische Universität Braunschweig, Informatik–Bericht 93–11, 1993.

[Het93] R. Hettler. Zur Übersetzung von E/R-Schemata nach Spectrum. Technischer Bericht TUM-I9333, TU München, 1993.

[LCFW92] M. Löwe, F. Cornelius, J. Faulhaber, and R. Wessäly. Ein Fallbeispiel für KORSO: Ein Vorschlag. Technischer Bericht 93-45, TU Berlin, 1992.

[MN93] S. Merz and F. Nickl. Ablauf Blutbilduntersuchung mit Unterabläufen Kleines Blutbild und Differentialblutbild. Arbeitspapier zur HDMS-A Fallstudie, 1993.

[SNM+93] O. Slotosch, F. Nickl, S. Merz, H. Hußmann, and R. Hettler. Die funktionale Essenz von HDMS-A. Technischer Bericht TUM-I9335, TU München, 1993.

Spezifikation eines vereinfachten Datenbanksystems
— ein Beitrag zur HDMS-A–Fallstudie —

Stefan Conrad
Technische Universität Braunschweig
Informatik, Abt. Datenbanken
D–38092 Braunschweig

e-mail: conrad@idb.cs.tu-bs.de

1 Einleitung

Im Rahmen des KORSO-Projektes wird eine umfangreiche Fallstudie durchgeführt. Gegenstand der Betrachtung ist eine maschinelle Unterstützung der Abläufe im Deutschen Herzzentrum Berlin [LCFW92, CKL92]. Schon in der Anfangsphase dieser Fallstudie (kurz: HDMS-A) wird deutlich, daß eine vollständige Neuentwicklung aller Bestandteile des Systems nicht typisch für realistische Softwareentwicklungsprojekte ist. Daher muß auch die Einbeziehung bestehender Komponenten und bereits entwickelter (z.B. kommerzieller) Software berücksichtigt werden.

Für die HDMS-A Fallstudie bietet es sich an, zur Datenhaltung und -verwaltung auf existierende Datenbanksysteme zurückzugreifen. Eine komplette Eigenentwicklung einer Datenbankkomponente innerhalb eines größeren Systems würde zu großen Aufwand (zeitlich und damit auch finanziell) bedeuten.

Um aber ein verifizierbares Gesamtsystem entwickeln zu können, müssen natürlich alle Einzelkomponenten formal spezifiziert werden können. Diese Forderung wird aber in der Regel von kommerziellen Anbietern von Standardsoftware (Textverarbeitungssysteme, Datenbanksysteme, etc.) insofern nicht berücksichtigt, daß sie für ihre Produkte keine formale Spezifikation bereitstellen (für die sie dann garantieren müßten, daß ihr Produkt eine korrekte Implementierung ist).

Diese Problematik hat uns dazu veranlaßt, exemplarisch eine formale Spezifikation eines Datenbanksystems zu entwickeln. Dabei haben wir uns nicht an ein spezielles, verfügbares Datenbanksystem gehalten, sondern ein — zugegebenermaßen — primitives, fiktives relationales Datenbanksystem, das nur mit elementarer Funktionalität ausgestattet sein soll, ins Auge gefaßt. Diese Wahl ist begründet durch die Tatsache, daß zur Zeit relationale Datenbanksysteme im industriellen Einsatz sind und sich gegenüber den früheren Datenbanksystemen, die auf dem Netzwerk- oder Hierarchischen-Datenmodell beruhen, durch größere Benutzerfreundlichkeit (z.B. *ad hoc* Anfragemöglichkeit) auszeichnen. Neuere Datenbanksysteme (basierend auf NF^2- oder Objektmodellen) haben bis jetzt noch nicht den Stand erreicht, daß sie in großen Umfang eingesetzt werden können. Dies liegt einerseits daran, daß erst Prototypen entwickelt worden sind, und andererseits daran, daß Managementkonzepte für komplexe Objekte noch Gegenstand der Forschung sind.

Wesentliche Bestandteile des von uns betrachteten relationalen Datenbanksystems sind

- das relationale Datenmodell zur Datenbankbeschreibung,
- elementare Anfrage- und Datenmanipulationsdienste und
- ein einfaches Transaktionskonzept.

2 Informelle Beschreibung

Zunächst soll das zu modellierende Datenbanksystem informell beschrieben werden. Für ein real existierendes System müßte an dieser Stelle eine *System–Analyse* durchgeführt werden. Ferner wären die Anforderungen der beabsichtigten Anwendung(en) an ein zu verwendendes Datenbanksystem zu klären (im Rahmen der *Requirement–Analyse*. Durch Vergleich der beiden Analyseergebnisse ist dann zu klären, ob das in Betracht gezogen Datenbanksystem überhaupt sinnvoll mit den geplanten Anwendungen zu kombinieren ist. Anderenfalls müßte man sich nach einem besser geeigneten Datenbanksystem umschauen oder Teile der geforderten Datenbankfunktionalität in der Anwendung selbst realisieren.

Die im vorigen Kapitel aufgezählten Bestandteile unseres Systems sollen nun im einzelnen detaillierter erläutert werden.

Das relationale Datenmodell: Das Datenmodell erlaubt uns, Datenbankschemata zu beschreiben. Ein Datenbankschema besteht aus einer Menge benannter *Relationenschemata*. Ein Relationenschema besteht aus einer endlichen Liste benannter und typisierter *Attribute*. Als mögliche Typen für Attribute erlauben wir hier zur Vereinfachung nur `string` und `int`.

Mögliche *Datenbankzustände* zu einem gegebenen Datenbankschema bestehen aus einer Menge benannter *Relationen* (wobei die Benennung der Relationen mit der Benennung im Schema übereinstimmen muß). Jede dieser Relationen enthält eine Menge von Tupeln, die zu dem zugehörigen Relationenschema "passen" müssen (Anzahl, Reihenfolge und Typisierung der Komponenten).

Elementare Anfrage– und Datenmanipulationsdienste: Um eine Datenbank nutzen zu können, müssen Anfragen möglich sein. Dafür sind elementare Anfrageoperationen notwendig. Um hier nicht die relationale Algebra oder einen der klassischen Anfragekalküle spezifizieren zu müssen, beschränken wir uns auf einen simplen Anfragemechanismus, der in der Regel gerade für Datenbankprogrammierschnittstellen (Einbettung der Datenbankfunktionalität in eine Wirtssprache) in ähnlicher Form realisiert ist.

Der/Die AnwenderIn (das Anwendungsprogramm) eröffnet einen `Scan` für eine Relation in einer Datenbank (mittels einer Operation/eines Events `open_scan`).

Dann kann mit Hilfe zweier Operationen (`get_first` und `get_next`) die Relation tupelweise durchlaufen werden. Das jeweils aktuelle Tupel kann dann ausgelesen und verändert (`update`) oder gelöscht (`delete`) werden. Neue Tupel können natürlich auch eingefügt werden (`insert`). Nach der Bearbeitung der Relation wird der zugehörige `Scan` wieder geschlossen (`close_scan`).

Neben der tupelbasierten Manipulation bestehender Relationen ist aber auch das Erzeugen und Löschen einzelner Relationen (durch die Operationen `create_relation` und `delete_relation`) und ebenso von ganzen Datenbanken (`create_database`, `delete_database`) möglich.

Vereinfachtes Transaktionskonzept: Das betrachtete Transaktionskonzept umfaßt nur die minimale Funktionalität. Transaktionen können eröffnet werden (`begin_trans`), erfolgreich beendet werden (`commit`) oder abgebrochen werden (`abort`). Wir haben diese Funktionalität zunächst auf Ein-Benutzer-Betrieb beschränkt. Damit gibt es immer nur eine Transaktion zu einem Zeitpunkt. Bei Abbruch der Transaktion muß der Datenbankzustand wiederhergestellt werden, der zum Zeitpunkt des Transaktionsbeginns bestand. Bei erfolgreicher Beendigung der Transaktion können alle zur Gewährleistung der Wiederherstellbarkeit gesicherten Informationen gelöscht werden.

Dieses vereinfachte Datenbanksystem ist sicherlich nicht als komfortabel zu bezeichnen noch erlaubt es Mehrbenutzerbetrieb. Dafür wäre die Spezifikation eines Transaktionkonzepts z.B. auf der Basis eines strikten 2-Phasen-Sperr-Protokolls mit geeigneten Sperren notwendig. Andererseits stellt schon dieses einfache Datenbanksystem hohe Anforderungen, denn die Spezifikation des relationalen Datenmodells muß dynamische Schemaänderungen (Neuanlegen/Löschen von Relationen und Datenbanken) ermöglichen. Verzichtet man auf Schemaevolution, d.h. geht man von einem festen, unveränderlichen Schema aus, könnte man dieses direkt in der Spezifikation festschreiben (im Prinzip sogar ohne Rücksicht auf das vom Datenbanksystem verwendete Datenmodell).

3 Formale Spezifikation

Für die formale Spezifikation verwenden wir als Spezifikationssprache TROLL *light* [CGH92], da es sich bei Datenbanksystemen um reaktive und zustandsorientierte Systeme mit persistenten Daten handelt. Gerade diese Paradigmen finden sich im objektorientierten Ansatz von TROLL *light* wieder.

Entsprechend der Strukturvorstellung, die wir in der Abb. 1 grob skizziert haben, folgt nun eine Spezifikation dieses (vereinfachten und idealisierten) Systems in TROLL *light*. Auf die Angabe der erlaubten Lebensläufe (`BEHAVIOR`–Abschnitt in TROLL *light*) ist hier aus Gründen der Übersichtlichkeit verzichtet worden.

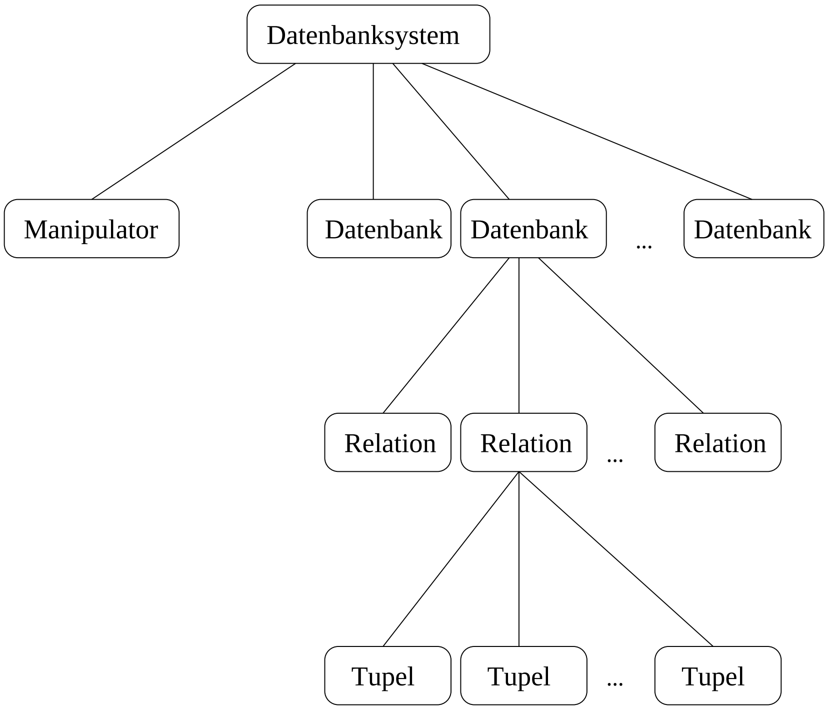


Abbildung 1: Strukturübersicht

```
TEMPLATE  Databasesystem
  DATA TYPES String;
  TEMPLATES  Manipulator, Database;
  SUBOBJECTS Manipulator            : manipulator;
             Database(Ident:string) : database;
  EVENTS BIRTH create;
               create_manipulator;
               create_database(Ident:string);
               destroy_database(Ident:string);
               begin_trans;
               commit;
               abort;
         DEATH destroy;
  INTERACTION
    create_manipulator               >> Manipulator.create(SELF) ;
    Manipulator.create_database(Id)  >> Database(Id).create ;
    Manipulator.destroy_database(Id) >> Database(Id).destroy ;
    Manipulator.create_relation(DB,RelId,RelSchema) >>
                    Database(DB).create_relation(RelId,RelSchema) ;
    Manipulator.destroy_relation(DB,RelId) >>
                    Database(DB).destroy_relation(RelId) ;
    begin_trans  >>  (SELECT PRJ(R,2) FROM R IN Database).begin_trans ;
    commit       >>  (SELECT PRJ(R,2) FROM R IN Database).commit ;
```

```
    abort         >>  (SELECT PRJ(R,2) FROM R IN Database).abort ;

END TEMPLATE;
```

```
TEMPLATE  Database
  DATA TYPES String;
  TEMPLATES  Relation;
  SUBOBJECTS Relation(Ident:string) : relation;
  EVENTS BIRTH create;
               create_relation(Ident     : string,
                               RelSchema : LIST(TUPLE(AttrId:string,
                                                      TypeId:string)));
               destroy_relation(Ident:string);
               begin_trans;
               commit;
               abort;
         DEATH destroy;
  INTERACTION
    create_relation(Id,Schema) >> Relation(Id).create(Schema) ;
    destroy_relation(Id)       >> Relation(Id).destroy ;
    begin_trans >> (SELECT PRJ(R,2) FROM R IN Relation).begin_trans ;
    commit      >> (SELECT PRJ(R,2) FROM R IN Relation).commit ;
    abort       >> (SELECT PRJ(R,2) FROM R IN Relation).abort ;
END TEMPLATE;
```

Die Interaction-Regeln, in denen `SELECT`-Terme auf der rechten Seite auftreten, sind folgendermaßen zu verstehen. Tritt das Event auf, das auf der linken Seite spezifiziert wurde, so wird in dem jeweiligen Zustand der `SELECT`-Term ausgewertet. Dieser liefert immer eine endliche Multi-Menge von Objekten, für die dann jeweils das Event, das auf der rechten Seite spezifiziert wurde, ausgeführt werden muß.

Für Relationen wird hier auf den (schreib-) aufwendigen Mechanismus verzichtet, der sicherstellt, daß Relationen, die in Transaktionen erzeugt oder gelöscht werden, im Falle des Transaktionsabbruchs auch wieder verschwinden bzw. wiederhergestellt werden. Dieser Mechanismus ist — eine Ebene tiefer — für Tupel in Relationen spezifiziert worden, so daß eine Adaption des Mechanismus für Relationen im wesentlichen nur Schreibarbeit wäre.

```
TEMPLATE  Relation
  DATA TYPES String, Bool, Nat;
  TEMPLATES  Tuple;
  SUBOBJECTS Tuple(No:nat) : tuple;
  ATTRIBUTES Schema        : LIST(TUPLE(AttrId:string,TypeId:string));
```

```
            -- Relationenschema als Liste von Attributnamen mit Typen
            IndexOfTuples : LIST(nat); -- enth"alt die aktuellen Tupel
            IndexShadow   : LIST(nat); -- enth"alt die Schattenversion
            NextNo        : nat;       -- zeigt auf die n"achste No

  EVENTS BIRTH create(Schema:LIST(TUPLE(AttrId:string,TypeId:string)));
              insert(Value: LIST(UNION(S:string,I:int)) );
              update(No:nat, Val:LIST(UNION(S:string,I:int)) );
              delete(No:nat);
              begin_trans;
              commit;
              abort;
        DEATH destroy;
  VALUATION
    [ create(Sch) ] Schema        = Sch,
                    IndexOfTuples = EMPTY,
                    NextNo = 0;
    { LEN(Schema) = LEN(V) AND     -- Bedingung (*)
      FORALL (N IN IND(Schema) )
         TypeID(SEL(Schema,N))="int" IMPLIES UNDEF( S( SEL(V,N))) AND
         TypeID(SEL(Schema,N))="string" IMPLIES UNDEF( I( SEL(V,N))) }
      [ insert(V) ] IndexOfTuples = append(IndexOfTuples,NextNo),
                    NextNo        = NextNo+1;
    [ delete(N) ]   IndexOfTuples = remove(IndexOfTuples,N);
    [ begin_trans ] IndexShadow   = IndexOfTuples;
    [ abort ]       IndexOfTuples = IndexShadow;
  INTERACTION
--  { (*) }                                    -- Bedingung s.o.
      insert(V) >> Tuple(NextNo).insert(V);
--  { (*) }                                    -- Bedingung s.o.
      update(N,V) >> Tuple(N).update(V);

    { NOT( Manipulation(Tuple(N)) = "inserted" ) }
        delete(N) >> Tuple(N).delete; -- log. L"oschen eines Tupels
    { Manipulation(Tuple(N)) = "inserted" }
        delete(N) >> Tuple(N).destroy;-- phys. L"oschen eines Tupels,
                                      -- das nur tempor"ar in einer
                                        -- Transaktion existierte
    begin_trans >> (SELECT Tuple(N)
                    FROM   N IN IndexOfTuples).begin_trans;
    commit >> (SELECT Tuple(N)
              FROM   N IN IndexOfTuples
              WHERE     Manipulation(Tuple(N)) = "changed"
                     OR Manipulation(Tuple(N)) = "inserted").commit;
    commit >> (SELECT Tuple(N)
```

```
               FROM   N IN IndexOfTuples
               WHERE  Manipulation(Tuple(N)) = "deleted").destroy;
    abort >>
         (SELECT Tuple(N)
          FROM   N IN IndexOfTuples
          WHERE  NOT(Manipulation(Tuple(N))="unmanipulated") ).abort;
    abort >> (SELECT Tuple(N)
              FROM   N IN IndexOfTuples
              WHERE  Manipulation(Tuple(N)) ="inserted" ).destroy;
END TEMPLATE;
```

Die mit (*) markierte Bedingung gehört eigentlich in den BEHAVIOR-Teil als Vorbedingung zum Eintreten von insert und update Events. Da wir hier aber auf den BEHAVIOR-Teil verzichtet haben, haben wir diese Bedingung sowohl in dem VALUATION- als auch dem INTERACTION-Teil eingeführt.

```
TEMPLATE  Tuple
  DATA TYPES String, Int;
  ATTRIBUTES Value : LIST( UNION(S:string, I:int) );
             Shadow: LIST( UNION(S:string, I:int) );
             Manipulation: string;   -- = "unmanipulated", "inserted",
                                     --    "deleted", or "changed"
  EVENTS BIRTH insert(Value: LIST(UNION(S:string, I:int)));
               update(Value: LIST(UNION(S:string, I:int)));
               delete;      -- logisches L"oschen in einer Transaktion
               begin_trans;
               commit;
               abort;
         DEATH destroy;
  VALUATION
    [ insert(L) ] Value        = L,
                  Manipulation = "inserted";
    [ update(L) ] Value        = L,
                  Manipulation = "changed";
    [ delete ] Manipulation = "deleted";
    [ begin_trans ] Shadow       = Value,   -- Schattenversion anlegen
                    Manipulation = "unmanipulated";
    [ commit ]  Manipulation = "unmanipulated";
    [ abort ]  Value        = Shadow,
               Manipulation = "unmanipulated";
END TEMPLATE;
```

```
TEMPLATE  Manipulator
  DATA TYPES String, Nat, Bool;
  TEMPLATES  Databasesystem;
  ATTRIBUTES DBS               :databasesystem;
             ScanStatus(No:nat):bool;  -- true  f"ur ge"offneten Scan
             ScanRel(No:nat)   :relation;
             ScanPtr(No:nat)   :nat;
             ScanVal(No:nat) :LIST(TUPLE(AttrId:string,
                                        Value:UNION(I:int,S:string)));
  EVENTS BIRTH create(Dbs:databasesystem);
               open_scan(DB_Id:string, RelId:string, ScanNo:nat);
               close_scan(No);
               get_first(ScanNo:nat);
               get_next(ScanNo:nat);
               insert_tuple(ScanNo:nat);
               update_tuple(ScanNo:nat,
                            Value:LIST(UNION(S:string,I:int)));
               delete_tuple(ScanNo:nat);
               create_database(Ident:string);
               delete_database(Ident:string);
               create_relation(DB_Id:string,
                               RelId:string,
                               RelSchema:LIST(TUPLE(AttrId:string,
                                                    TypeId:string)) );
               destroy_relation(DB_Id:string, RelId:string);
               begin_trans;
               commit;
               abort;
         DEATH destroy;
    -- die Events 'update_tuple' und 'delete_tuple' beziehen sich immer
    -- auf das Tupel, auf das der entsprechende 'ScanPtr' zeigt
  VALUATION
  [ create(Dbs) ] DBS = Dbs;
  [ open_scan(DB_Id,RelId,No) ]
         ScanStatus(No) = true,
         ScanRel(No)    = Relation(Database(DBS, DB_Id), RelId);
  [ close_scan(No) ]
         ScanStatus(No) = BOTTOM,
         ScanRel(No)    = BOTTOM,
         ScanPtr(No)    = BOTTOM,
         ScanVal(No)    = BOTTOM;
  [ get_first(No) ]
         ScanPtr(No) = FIRST(TupleList(ScanRel(No))),
         ScanVal(No) = Value(Tuple(ScanRel(No),
                                   FIRST(TupleList(ScanRel(No)))));
```

```
  [ get_next(No) ]
        ScanPtr(No) = SEL(TupleList(ScanRel(No)),
                          POS(TupleList(ScanRel(No)),ScanPtr(No))+1),
        ScanVal(No) = Value(Tuple(ScanRel(No),
                                  SEL(TupleList(ScanRel(No)),
                                      POS(TupleList(ScanRel(No)),
                                          ScanPtr(No))+1)));
  [ delete_tuple(ScanNo) ]
        ScanPtr(No) = SEL(TupleList(ScanRel(No)),
                          POS(TupleList(ScanRel(No)),ScanPtr(No))+1),
        ScanVal(No) = Value(Tuple(ScanRel(No),
                                  SEL(TupleList(ScanRel(No)),
                                      POS(TupleList(ScanRel(No)),
                                          ScanPtr(No))+1)));
        -- Cursor auf n"achstes Tupel setzen
  INTERACTION
    insert_tuple(ScanNo,Value) >> ScanRel(ScanNo).insert(Value);
    update_tuple(ScanNo,Value) >>
                      ScanRel(ScanNo).update(ScanPtr(ScanNo),Value);
    delete_tuple(ScanNo)       >>
                          ScanRel(ScanNo).delete(ScanPtr(ScanNo)) ;
    begin_trans >> DBS.begin_trans ;
    commit      >> DBS.commit ;
    abort       >> DBS.abort ;
END TEMPLATE;
```

4 Abschließende Bemerkungen

4.1 Anmerkungen zu den vorgenommenen Vereinfachungen

Eine Reihe von Vereinfachungen sind hier vorgenommen, um einerseits die prinzipielle Vorgehensweise nicht durch Unmengen von Details zu verdecken und um andererseits eine Reihe von Problemen zunächst zu vermeiden. Folgende Vereinfachungen wurden getroffen:

- Ein-Benutzer– statt Mehr-Benutzer-Betrieb

  In der Realität werden Datenbanken vor allem im Mehr-Benutzer-Betrieb interessant, um konkurrierende Datenmanipulation zu erlauben und zu organisieren. Da hier nur ein Manipulator-Objekt vorgesehen ist, kann eigentlich nur ein Benutzer auf die Datenbank zugreifen.

  Für ein Mehr-Benutzer-Datenbanksystem müßte auch ein Sperrverfahren spezifiziert werden, da kommerzielle Systeme in der Regel mit Sperrverfahren arbeiten.

Zu diesem Sperrverfahren kommt dann noch ein Sperrprotokoll hinzu, das von der Transaktionsverwaltung zu realisieren wäre.

- Keine Rechteverwaltung

  Gerade im Rahmen der HDMS-Fallstudie wird ein Datenbanksystem auch in der Lage sein müssen, für verschiedene BenutzerInnen (-gruppen) verschiedene Zugriffs- und Manipulationsrechte zu realisieren.

- Datenbank-Administration wird nicht unterstützt

  Die beiden ersten Einschränkungen bedeuten auch, daß in der Spezifikation eine Reihe von typischen Administrationsaufgaben nicht berücksichtigt werden mußten. Mehr-Benutzer-Betrieb bringt in der Regel auch die Administration von Benutzerdaten, insbesondere Einrichtung neuer Benutzer, mit sich. Bei Rechteverwaltung müßte zusätzlich unterschieden werden, welcher Benutzer welche Daten sehen darf und welche Daten ändern darf. Für diese Daten müßte es dann auch noch einen (oder mehrere?) ausgezeichnenten "Benutzer" geben, den Datenbank-Administrator.

- Wiederherstellbarkeit der Daten wird nur z.T. unterstützt

  In der angebenen Spezifikation ist die Wiederherstellbarkeit der Daten nur dann garantiert, wenn in der laufenden Transaktion nur die Inhalte von Relationen manipuliert wurden. Sobald es Schemaänderungen gibt, z.B. durch Einfügen von neuen Relationen oder Löschen von bestehenden Relationen, ist der Datenbank-Zustand, der zu Beginn der Transaktion galt, nicht wiederherstellbar.

- Es gibt keine *ad hoc* Anfragesprache

  Typischerweise besitzen relationale Datenbanksysteme eine *ad hoc* Anfragesprache (in der Regel `SQL`). Hier wurde darauf verzichtet, eine solche Anfragesprache in der Spezifikation zu berücksichtigen, da dann auch der vollständige Auswertungsmechanismus spezifiziert werden müßte.

- Integritätsregeln werden nicht unterstützt

  Wir haben hier den Aufwand gescheut, eine Form der Integritätsüberwachung in der Spezifikation zu berücksichtigen. Hierzu hätte z.B. ein Integritätsmonitor entworfen werden müssen. Da Integritätsbedingungen in der Regel schemaabhängig sind und wir Schemaevolution schon bis zu einem gewissen Grad berücksichtigt haben, müßten auch Integritätsbedingungen dynamisch (d.h. zur Laufzeit) erzeugt, geändert und gelöscht werden können. Dafür wäre aber erheblicher zusätzlicher Spezifikationsaufwand nötig.

Ferner sind noch eine Reihe kleinere Vereinfachungen vorgenommen worden, so erlauben wir z.B. nur zwei Basisdatentypen für Attribute in Relationen, und zwar `int` und `string`.

4.2 Anmerkungen zur Verwendung von TROLL *light*

Die Benutzung von TROLL *light* zur Spezifikation eines (wenn auch nur imaginär) bestehenden Datenbanksystems zeigt ein generelles Problem auf: Die Darstellung eines Datenmodelles in einem anderen ist zwar von der theoretischen Seite ein schönes Ergebnis, allerdings kann dies unter ästhetischen Gesichtspunkten nicht befriedigend sein, denn je verschiedener die Konzepte der beiden Datenmodelle sind, desto mehr Overhead ist nötig. Dies gilt insbesondere auch für die hier vorgenommene Darstellung des Relationenmodells in TROLL *light*. Die Realisierung von Tupeln durch Objekte scheint doch inadäquat zu sein, ist allerdings in TROLL *light* notwendig, um das Datenbankverhalten einigermaßen adäquat zu modellieren. Eine Reihe von Vereinfachungen könnten hier getroffen werden, indem zumindestens Teile des Relationenmodells in eine geeignete Datentyp-Spezifikation verlagert werden und dann TROLL *light* vorrangig zur Spezifikation des Verhaltens benutzt wird.[3]

Eine weitere allgemeine Bemerkung zur Verwendung von TROLL *light* muß hier auch getroffen werden: TROLL *light* ist nicht primär zur konzeptionellen Datenmodellierung auf hohem Abstraktionsniveau entworfen wurden. In diesem Bereich ist sicherlich TROLL [JSHS91] besser geeignet. TROLL *light* stellt eher einen ausführbaren Dialekt von TROLL dar. Allerdings lassen sich auch in TROLL *light* abstrakte Spezifikationen schreiben, so sind Events auch eine Form der Abstraktion, denn ein Event kann im Prinzip für eine Folge von Berechnungsschritten stehen, die nur vollständig oder überhaupt nicht ausgeführt werden darf (damit können Events auch als primitive Transaktionen betrachtet werden). Auch die Verwendung von abstrakten Datentypen, die außerhalb von TROLL *light* spezifiziert werden, dient zur Abstraktion in der Spezifikation und kann auch dazu führen, daß TROLL *light* Spezifikationen nicht mehr ausführbar sind, bzw. daß es zu einer TROLL *light* Spezifikation mehrere Modelle geben kann (durch eine lose Spezifikation der Datentypen z.B. in SPECTRUM [BFG$^+$92]).

Die Einführung des Recovery-Mechanismuses bereitete einige Probleme, da hier gefordert war, einen alten Datenbankzustand zu merken und, falls erforderlich, später diesen aufgehobenen alten Datenbankzustand wiederherzustellen. Diese Vorgänge kann man sich leicht als einfache Kopiervorgänge vorstellen. Objekte zu kopieren, wird aber in TROLL *light* nicht unterstützt. Daher wurden (begrenzt auf die Tupelebene) die Objekte so spezifiziert, daß sie sich selbst eine Kopie ihres aktuellen Wertes anlegen können. Damit wurde die Verwaltung des gemerkten Zustands in die Tupelobjekte hineinverlagert.

4.3 Methodischen Überlegungen und Schlußfolgerungen

Da es sich bei dieser formalen Spezifikation nicht um eine Neuentwicklung handelte, sondern um das nachträgliche Formalisieren einer bereits abgeschlossenen Entwicklung

[3] Wo hier die Grenzen zwischen Datentyp- und Objektspezifikation zu ziehen sind, ist ein sehr interessantes Problem, das auch in der KORSO-Arbeitsgruppe “Beschreibung und Modellierung” diskutiert wird.

(*reverse engineering*), mußte zuerst eine Systemanalyse erfolgen. Ergebnis dieser Analyse sind die Struktur des Systems (grob vereinfacht in Abb. 1 dargestellt) und die nach außen sichtbare Funktionalität, z.B. als Sammlung von zur Verfügung stehenden Operationen.

Wie schon im vorigen Abschnitt diskutiert, ist eine Kooperation verschiedener Spezifikationssprachen für die Spezifikation größere Systeme wohl sinnvoll. In dem vorliegenden Beispiel könnten wir uns eine zusammengesetzte Spezifikation derart vorstellen, daß z.B. in SPECTRUM geeignete Datentypen spezifiziert werden, so daß die TROLL *light* Spezifikation stärker von Datentyp-Details abstrahieren kann und sich vorrangig mit dem Verhalten beschäftigen kann. Ein derartiges Zusammenspiel zwischen einer "applikativen" Spezifikationssprache (SPECTRUM) und einer "zustandsbasierten" Spezifikationssprache (TROLL *light*) bietet sicher eine Reihe von Vorteilen, da die jeweiligen Sprachen in ihrem jeweils bevorzugten Anwendungsgebiet zum Einsatz kommen. Sinnvoll wäre es, dieses Zusammenspiel anhand einer Fallstudie weiter zu untersuchen.

Literaturverzeichnis

[BFG+92] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, and K. Stølen. The Requirement and Design Specification Language SPECTRUM — An Informal Introduction (Version 0.3). Technischer Bericht TUM–I9140, TU München, 1992.

[CGH92] S. Conrad, M. Gogolla, and R. Herzig. TROLL *light*: A Core Language for Specifying Objects. Informatik-Bericht 92–02, Technische Universität Braunschweig, 1992.

[CKL92] F. Cornelius, M. Klar, and M. Löwe. Ein Fallbeispiel für KORSO: Ist-Analyse für HDMS-A. Technischer Bericht 93-28, TU Berlin, 1992.

[JSHS91] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-Oriented Specification of Information Systems: The TROLL Language. Informatik-Bericht 91–04, Technische Universität Braunschweig, 1991.

[LCFW92] M. Löwe, F. Cornelius, J. Faulhaber, and R. Wessäly. Ein Fallbeispiel für KORSO: Ein Vorschlag. Technischer Bericht 93-45, TU Berlin, 1992.

Spezifikation einer Fertigungszelle mit TROLL *light*

Rudolf Herzig u. Nikolaos Vlachantonis

Technische Universität Braunschweig
Informatik, Abt. Datenbanken
D–38092 Braunschweig

e-mail: herzig@idb.cs.tu-bs.de

Zusammenfassung

Im Rahmen des KorSo-Verbundprojektes werden Fallbeispiele für die praxisnahe Untersuchung der theoretischen Ergebnisse der einzelnen Partner herangezogen. In dieser Arbeit wird TROLL *light* [CGH92] zur Spezifikation einer Fertigungszelle [LRS92] benutzt. Ein Modell dieser Fertigungszelle wird vom FZI Karlsruhe zur Verfügung gestellt.

1 Konzepte von TROLL *light*

TROLL *light* ist eine Sprache zur Beschreibung von strukturellen und dynamischen Eigenschaften von Objekten. Die Beschreibung von strukturellen Eigenschaften orientiert sich in weiten Teilen an semantischen Datenmodellen.

- Die in einem Zustand beobachtbaren Eigenschaften eines Objektes werden durch Attribute beschrieben. Einfache Attribute sind daten– oder objektwertig. Darüber hinaus erlauben vordefinierte Sortenkonstruktoren wie *set* oder *tuple* auch die Spezifikation komplexer Attributbereiche.

- TROLL *light*–Objekte sind in Aggregationshierarchien eingebettet. Das oberste Objekt einer solchen Hierarchie ist das Schemaobjekt.

- Normalerweise werden Attributwerte direkt durch das Eintreten gewisser Ereignisse bestimmt. Daneben ist es auch möglich, abgeleitete Attribute zu spezifizieren, d.h. Attribute, deren Inhalte sich aus anderer gespeicherter oder abgeleiteter Information bestimmen. Zur Formulierung von Ableitungsregeln stellt TROLL *light* einen SQL–ähnlichen Anfragekalkül zur Verfügung.

- Der Anfragekalkül von TROLL *light* unterstützt auch die Spezifikation statischer Integritätsbedingungen.

Die Beschreibung von dynamischen Eigenschaften basiert auf der Spezifikation von Ereignissen. Ereignisse sind Abstraktionen von zustandsverändernden Operationen auf Objekten.

- Objektereignisse werden durch eine endliche Menge von Ereignisgeneratoren beschrieben. Jeder Ereignisgenerator kann mit einer Liste von Parametersorten versehen sein. Ein Ereignisgenerator zusammen mit aktuellen Parameterwerten liefert ein Ereignis.
- Die Auswirkung von Ereignissen auf Attribute werden durch Auswertungsregeln beschrieben.
- Ereignisse in verschiedenen Objekten können durch Interaktionsregeln synchronisiert werden.
- Die möglichen Ereignisfolgen können mit Hilfe CSP-ähnlicher Prozeßbeschreibungen (*Objektmaschinen*) auf zulässige Folgen eingeschränkt werden.

Den Rahmen zur Beschreibung all dieser Objekteigenschaften bilden *Templates*. Templates besitzen folgenden Aufbau:

```
TEMPLATE
  DATA TYPES     Deklaration von verwendeten Datentypen
  TEMPLATES      Deklaration von anderen verwendeten Objektbeschreibungen
  SUBOBJECTS     Unterobjektbeziehungen
  ATTRIBUTES     Attribute
  EVENTS         Ereignisgeneratoren
  CONSTRAINTS    statische Integritätsbedingungen
  VALUATION      Auswirkung von Ereignissen auf Attribute
  DERIVATION     Ableitungsregeln für abgeleitete Attribute
  INTERACTION    Synchronisierung von Ereignissen in verschiedenen Objekten
  BEHAVIOR       Beschreibung zulässiger Lebensläufe
END TEMPLATE
```

2 Aufgabenstellung

Am Forschungszentrum Informatik Karlsruhe existiert ein Modell einer Fertigungszelle für Modulschienen [LRS92]. Die Fertigungszelle besteht aus mehreren Komponenten zur Bearbeitung von Metallrohlingen. Die Metallrohlinge gelangen über ein Zuführförderband auf einen Hubdrehtisch. Ein Roboter nimmt die Rohlinge dort auf und transportiert sie zu einer Presse. Die bearbeiteten Rohlinge werden vom Roboter zu einem Ablageförderband befördert. Dieses befördert die Teile zu einem Handhabungsgerät, das wiederum ankommende Teile zurück auf das Zuführförderband transportiert (siehe Abb. 2).

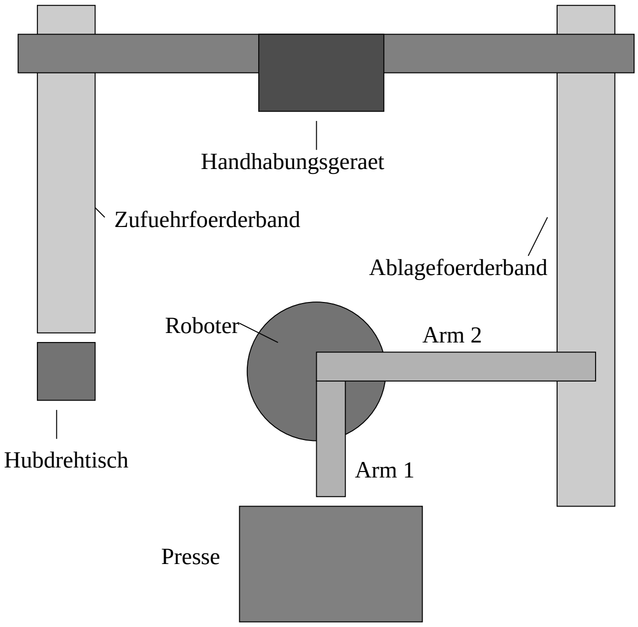


Abbildung 2: Aufsicht auf das Modell

Zuführförderband, Hubdrehtisch, Ablageförderband. Das Zuführförderband nimmt die Rohlinge vom Handhabungsgerät auf und transportiert sie zum Hubdrehtisch. Dieser nimmt immer nur ein ankommendes Teil auf, sofern er frei ist. Das Teil wird dann um 90° gedreht und auf die Höhe des ersten Roboterarms gebracht. Das Ablageförderband nimmt die bearbeiteten Teile vom Roboter auf und transportiert sie wieder zum Handhabungsgerät. Am Ende des Ablageförderbandes befindet sich eine Lichtschranke. Unterbricht ein bearbeitetes Teil die Lichtschranke, so hält das Ablageförderband an, bis das Handhabungsgerät das Teil aufnimmt und zurück auf das Zuführförderband bringt.

Dieses Handhabungsgerät dient im Modell dazu, einen Kreislauf von zu bearbeitenden Rohlingen bereitzustellen, da andernfalls ständig neue Teile auf das Zuführförderband aufgeladen werden müßten. In einer realen Fertigungszelle würde kein Handhabungsgerät vorhanden sein.

Handhabungsgerät. Das Handhabungsgerät nimmt auf dem Ablageförderband ankommende Teile auf und transportiert diese zum Zuführförderband, wo sie erneut in den Bearbeitungszyklus aufgenommen werden. Für das Modell wird es dadurch möglich, ohne manuelle Eingriffe und ohne Zuführung neuer Teile fortlaufend arbeiten zu können. Eine reale Fertigungszelle würde die Rohlinge natürlich nicht im selben Zyklus bearbeiten, sondern wäre in einem größeren Fertigungsprozeß eingebunden, wo Rohlinge bearbeitet und an nachfolgende Maschinen weitergereicht würden.

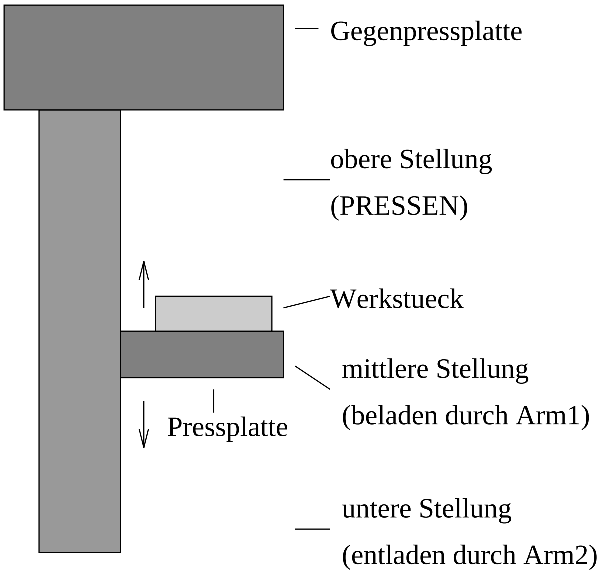


Abbildung 3: Seitenansicht der Presse

Roboter. Der Roboter hat die Aufgabe, Rohlinge vom Hubdrehtisch aufzunehmen und durch eine 90°–Drehung um die eigene Achse in die Presse zu transportieren. Er ist mit zwei Armen ausgestattet, um eine bessere Auslastung der Presse zu erreichen. Aus technischen Gründen liegen die beiden Arme übereinander, so daß sich zwei verschiedene Höhen für die Aufnahme bzw. Abgabe von Teilen für die Presse ergeben. Beide Arme lassen sich ein- und ausfahren, um die Presse bzw. den Hubdrehtisch entsprechend bedienen zu können. An ihrem Ende haben beide einen Magneten, mit dem die Metallteile aufgenommen werden können. Der erste Arm dient dem Transport von Teilen vom Hubdrehtisch zur Presse, wohingegen der zweite Arm bearbeitete Teile aus der Presse herausnimmt und auf das Ablageförderband transportiert.

Presse. Die Presse ist die eigentliche Bearbeitungseinheit der Fertigungszelle. Hier werden ankommende Rohlinge in eine bestimmte Form gepreßt. Die Presse besteht im wesentlichen aus einer feststehenden Gegenpressplatte, die oben auf der Presse sitzt, und einer Pressplatte, die auf einem Schlitten hoch- und runtergefahren werden kann. Ein Rohling wird vom ersten Roboterarm auf die Pressplatte gelegt, anschließend hochgefahren und gegen die Pressform in der Gegenpressplatte gedrückt. Die Pressplatte fährt dann wieder runter und das bearbeitete Teil wird vom zweiten Roboterarm wieder aufgenommen (siehe Abb. 3).

Für jedes Objekt der Fertigungszelle wird im folgenden eine entsprechende Objektbeschreibung in TROLL *light* gegeben. Wo es notwendig erscheint, sind einige Bemerkungen angefügt. Eine ausführlichere Sprachbeschreibung von TROLL *light* ist in [CGH92] enthalten.

3 Die Spezifikation

3.1 Fertigungszelle

```
TEMPLATE Fertigungszelle
  DATA TYPES
    Nat;
  TEMPLATES
    Zufuehrfoerderband, Hubdrehtisch, Roboter, Presse,
    Ablagefoerderband, Lichtschranke, Manipulator;
  SUBOBJECTS
    Zufuehrfoerderband : zufuehrfoerderband;
    Hubdrehtisch       : hubdrehtisch;
    Roboter            : roboter;
    Presse             : presse;
    Ablagefoerderband  : ablagefoerderband;
    Ablagesensor       : lichtschranke;
    Manipulator        : manipulator;
  ATTRIBUTES
    Teile# : nat;
  EVENTS
    BIRTH installiere;
    installiereKomponenten;
    initialisiere;
    fuehreTeilein;
    DEATH baueab;
  VALUATION
    [installiere]   Teile# = 0;
    [fuehreTeilein] Teile# = Teile#+1;
  INTERACTION
    installiereKomponenten >> Zufuehrfoerderband.installiere,
                              Hubdrehtisch.installiere,
                              Roboter.installiere,
                              Presse.installiere,
                              Ablagefoerderband.installiere,
                              Ablagesensor.installiere,
                              Manipulator.installiere;
    initialisiere >> Zufuehrfoerderband.starte,
                     Roboter.nehmeAnfangspositionein,
                     Presse.verbindemitUmgebung(Roboter),
                     Ablagefoerderband.starte;
    -- Zusammenspiel der Systemkomponenten:
    fuehreTeilein                   >> Zufuehrfoerderband.erhalteTeil;
    Zufuehrfoerderband.gebeTeilab >> Hubdrehtisch.erhalteTeil;
```

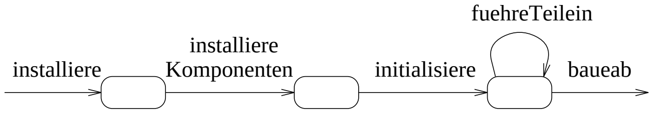


Abbildung 4: Objektmaschine `Fertigungszelle`

```
    Roboter.greifeTeilvonTisch    >> Hubdrehtisch.gebeTeilab;
    Roboter.gebeTeilabfuerPresse  >> Presse.erhalteTeil;
    Roboter.greifeTeilvonPresse   >> Presse.gebeTeilab;
    Roboter.gebeTeilabfuerBand    >> Ablagefoerderband.erhalteTeil;
    Ablagesensor.verliereStrahl   >> Ablagefoerderband.halteanzumLaden,
                                     Manipulator.erhalteAuftrag;
    Ablagesensor.entdeckeStrahl   >> Ablagefoerderband.starte;
    Manipulator.oeffneGreifer     >> Zufuehrfoerderband.erhalteTeil;
  BEHAVIOR
    PROCESS Fertigungszelleleben =
              (fuehreTeilein -> Fertigungszelleleben |
               baueab -> POSTMORTEM);
    (installiere -> installiereKomponenten ->
                    initialisiere -> Fertigungszelleleben);
END TEMPLATE;
```

Jede Objektbeschreibung impliziert eine Objektsorte. Zur Unterscheidung von Objektbeschreibungen und Objektsorten verwenden wir großgeschriebene Namen zur Bezeichnung von Objektbeschreibungen und die gleichen Namen kleingeschrieben zur Bezeichnung der zugehörigen Objektsorten.

Ein Objekt der Sorte `fertigungszelle` kann aus mehreren Unterobjekten (Komponenten) bestehen. Unter `SUBOBJECTS` werden allerdings nur die Namen dieser Unterobjekte deklariert. Diese Namen werden mit aktuellen Objekten verbunden, wenn Ereignisse der Fertigungszelle mit Geburtsereignissen der Unterobjekte synchronisiert werden (s. unten).

Ein Objekt der Sorte `fertigungszelle` besitzt ein Attribut, das die Anzahl der in einer Fertigungszelle gleichzeitig bearbeiteten Teile anzeigt. Nach der Installation einer Fertigungszelle hat das Attribut den Wert null. Wenn ein Teil in die Zelle eingeführt wird, erhöht sich der Wert entsprechend um eins.

Das Verhalten eines Objekts der Sorte `fertigungszelle` ist in Abb. 4 dargestellt.

Die Synchronisation des Ereignisses `installiereKomponenten` mit Geburtsereignissen der Unterobjekte bewirkt, daß den Unterobjektnamen, denen nach Geburt der Fertigungszelle zunächst keine Objekte zugewiesen sind, aktuelle Objekte zugeordnet werden. Das Ereignis `initialisiere` in Fertigungszelle bewirkt einige vorbereitende Ereignisse in den Komponenten. Schließlich wird auf der Ebene der Fertigungszelle das Zusammenspiel der Systemkomponenten spezifiziert.

3.2 Das Zuführförderband

```
TEMPLATE Zufuehrfoerderband
  DATA TYPES  Nat;
  ATTRIBUTES  Teile# : nat;
  EVENTS      BIRTH installiere;
                    starte;
                    haltean;
                    erhalteTeil;
                    gebeTeilab;
  VALUATION   [installiere] Teile# = 0;
              [erhalteTeil] Teile# = Teile#+1;
              [gebeTeilab]  Teile# = Teile#-1;
  BEHAVIOR    PROCESS StehendesBand =
                (starte -> LaufendesBand);
              PROCESS LaufendesBand =
                (haltean -> StehendesBand |
                 erhalteTeil,
                 { Teile#>0 } gebeTeilab -> LaufendesBand);
              (installiere -> StehendesBand);
END TEMPLATE;
```

Das Attribut `Teile#` gibt die Anzahl der Teile auf dem Förderband wieder. Im `BEHAVIOR`-Abschnitt ist gezeigt, wie Zustandsüberänge des Objektautomaten mit Vorbedingungen versehen werden können. So ist das Ereignis `gebeTeilab` nur erlaubt, sofern die Anzahl der Teile auf dem Band größer als null ist.

Der Einfachheit halber bleiben die Aktoren wie Elektromotor und Elektromagnet in der aktuellen Spezifikation unberücksichtigt. Im Zuge einer Verfeinerung könnte man natürlich die Objektbeschreibung `Zufuehrfoerderband` um ein entsprechendes Unterobjekt `Motor` mit Template `Elektromotor` erweitern und die Ereignisse `starte` mit `Motor.starte` und `haltean` mit `Motor.haltean` synchronisieren.

3.3 Der Hubdrehtisch

```
TEMPLATE Hubdrehtisch
  DATA TYPES  Hubposition,     -- { unten, oben }
              Schalterzustand, -- { an, aus }
              Real;
  TEMPLATES   Schalter, Potentiometer;
  SUBOBJECTS  SO : schalter;   -- Schalter oben;
              SU : schalter;   -- Schalter unten;
              Drehwinkelmesser : potentiometer;
  ATTRIBUTES  Position : hubposition;
              DERIVED Drehwinkel : real;
```

```
  EVENTS      BIRTH installiere;
                    installiereKomponenten;
                    erhalteTeil;
                    hebeunddrehe;
                    oberePositioneingenommen;
                    gebeTeilab;
                    senkeunddrehe;
                    unterePositioneingenommen;
  VALUATION   [installiere]               Position = unten;
              [oberePositioneingenommen]  Position = oben;
              [unterePositioneingenommen] Position = unten;
  DERIVATION  Drehwinkel = f1(Messwert(Drehwinkelmesser));
  INTERACTION installiereKomponenten >> SU.installiere(an),
                                        SO.installiere(aus),
                                        Drehwinkelmesser.installiere;
              SU.schaltean >> unterePositioneingenommen;
              SO.schaltean >> oberePositioneingenommen;
  BEHAVIOR PROCESS Arbeitszyklus =
             ({ Drehwinkel=Zufuehrbandpos } erhalteTeil ->
                  hebeunddrehe -> oberePositioneingenommen ->
              { Drehwinkel=Roboterpos } gebeTeilab -> senkeunddrehe ->
                  unterePositioneingenommen -> Arbeitszyklus);
           (installiere -> installiereKomponenten -> Arbeitszyklus);
END TEMPLATE;
```

Die Werte von Attributen ergeben sich im Normalfall aus der Anwendung von Auswertungsregeln. Die obige Objektbeschreibung gibt ein Beispiel für ein abgeleitetes Attribut. Der Wert eines abgeleiteten Attributs bestimmt sich aus einer entsprechenden Ableitungsregel. Hier bezeichne `f1` eine fest vorgegebene Funktion, die aus dem Messwert des Potentiometers den momentanen Drehwinkel berechnet. Ähnliches gilt in den folgenden Templates für die Funktionen `f2`, `f3` und `f4`. Man beachte, daß `Zufuehrbandpos` und `Roboterpos` fest vorgegebene Winkelpositionen bezeichnen.

Im nächsten Template bezeichnen `l1, l2, l3` und `l4` sowie `Anfangspos1` und `Anfangspos2` bestimmte Längenkonstanten, und `Anfangspos`, `Tischpos`, `Pressepos1`, `Pressepos2` und `Ablagebandpos` feste Drehwinkel.

3.4 Der Roboter

```
TEMPLATE Roboter
  DATA TYPES
    Real;
  TEMPLATES
    Roboterarm, Potentiometer;
  SUBOBJECTS
```

```
  Arm1 : roboterarm; Arm2 : roboterarm;
  Drehwinkelmesser : potentiometer;
ATTRIBUTES
  DERIVED Drehwinkel : real;
EVENTS
  BIRTH installiere;
  installiereKomponenten;
  nehmeAnfangspositionein;
  drehezuTisch;    fahreArmauszuTisch;    greifeTeilvonTisch;
  drehezuPresse1; fahreArm1auszuPresse; gebeTeilabfuerPresse;
  fahreArm1ein;
  drehezuPresse2; fahreArm2auszuPresse; greifeTeilvonPresse;
  fahreArm2ein;
  drehezuBand;     fahreArmauszuBand;     gebeTeilabfuerBand;
DERIVATION
  Drehwinkel = f2(Messwert(Drehwinkelmesser));
INTERACTION
  installiereKomponenten  >> Arm1.installiere, Arm2.installiere,
                             Drehwinkelmesser.installiere;
  nehmeAnfangspositionein >> Arm1.stelleaufLaenge(Anfangspos1),
                             Arm2.stelleaufLaenge(Anfangspos2);
  fahreArmauszuTisch      >> Arm1.stelleaufLaenge(l1);
  fahreArm1auszuPresse    >> Arm1.stelleaufLaenge(l2);
  fahreArm2auszuPresse    >> Arm2.stelleaufLaenge(l3);
  fahreArmauszuBand       >> Arm2.stelleaufLaenge(l4);
  fahreArm1ein            >> Arm1.stelleaufLaenge(Anfangspos1);
  fahreArm2ein            >> Arm2.stelleaufLaenge(Anfangspos2);
  greifeTeilvonTisch      >> Arm1.schliesseGreifer;
  greifeTeilvonPresse     >> Arm2.schliesseGreifer;
  gebeTeilabfuerPresse    >> Arm1.oeffneGreifer;
  gebeTeilabfuerBand      >> Arm2.oeffneGreifer;
BEHAVIOR
  PROCESS Startphase =
    ({ Laenge(Arm1)=Anfangspos1 AND Laenge(Arm2)=Anfangspos2 AND
       Drehwinkel=Anfangspos } drehezuTisch ->
     { Drehwinkel=Tischpos } fahreArmauszuTisch ->
     { Laenge(Arm1)=l1 } greifeTeilvonTisch -> fahreArm1ein ->
     { Laenge(Arm1)=Anfangspos1 } drehezuPresse1 ->
     { Drehwinkel=PressePos1 } fahreArm1auszuPresse ->
     { Laenge(Arm1)=l2 } gebeTeilabfuerPresse -> fahreArm1ein ->
     { Laenge(Arm1)=Anfangspos1 } nehmeAnfangspositionein
         -> Arbeitszyklus);
  PROCESS Arbeitszyklus =
    ({ Laenge(Arm1)=Anfangspos1 AND Laenge(Arm2)=Anfangspos2 AND
       Drehwinkel=Anfangspos } drehezuTisch ->
```

```
      { Drehwinkel=Tischpos } fahreArmauszuTisch ->
      { Laenge(Arm1)=l1 } greifeTeilvonTisch -> fahreArm1ein ->
      { Laenge(Arm1)=Anfangspos1 } drehezuPresse2 ->
      { Drehwinkel=PressePos2 } fahreArm2auszuPresse ->
      { Laenge(Arm2)=l3 } greifeTeilvonPresse -> fahreArm2ein ->
      { Laenge(Arm2)=Anfangspos2 } drehezuPresse1 ->
      { Drehwinkel=PressePos1 } fahreArm1auszuPresse ->
      { Laenge(Arm1)=l2 } gebeTeilabfuerPresse -> fahreArm1ein ->
      { Laenge(Arm1)=Anfangspos1 } drehezuBand ->
      { Drehwinkel=Ablagebandpos } fahreArmauszuBand ->
      { Laenge(Arm2)=l4 } gebeTeilabfuerBand -> fahreArm2ein ->
      { Laenge(Arm2)=Anfangspos2 } nehmeAnfangspositionein
           -> Arbeitszyklus);
   (installiere -> installiereKomponenten -> nehmeAnfangspositionein
      -> Startphase);
END TEMPLATE;
```

Die Sicherheitsbedingungen drücken sich in den Vorbedingungen für das Eintreten von Ereignissen aus.

3.5 Der Roboterarm

```
TEMPLATE Roboterarm
  DATA TYPES  Real;
  TEMPLATES   Potentiometer;
  SUBOBJECTS  Laengenmesser : potentiometer;
  ATTRIBUTES  DERIVED Laenge : real;
  EVENTS      BIRTH installiere;
                    installiereKomponenten;
                    stelleaufLaenge(Laenge : real);
                    oeffneGreifer;
                    schliesseGreifer;
  DERIVATION  Laenge = f3(Messwert(Laengenmesser));
  INTERACTION installiereKomponenten >> Laengenmesser.installiere;
  BEHAVIOR  PROCESS Roboterarmleben =
              (oeffneGreifer, schliesseGreifer, stelleaufLaenge
                -> Roboterarmleben);
            (installiere -> installiereKomponenten -> Roboterarmleben);
END TEMPLATE;
```

Ereignisgeneratoren können Parameter haben. Die Parameter können bei der Anwendung von Auswertungs- oder Synchronisationsregeln von Bedeutung sein.

3.6 Die Presse

```
TEMPLATE Presse
  DATA TYPES  Presseposition, -- { unten, mitte, oben }
              Schalterzustand;
  TEMPLATES   Schalter, Roboter;
  SUBOBJECTS  SO : schalter; SM : schalter; SU : schalter;
  ATTRIBUTES  Bediener : roboter;
              Position : presseposition;
  EVENTS      BIRTH installiere;
                    installiereKomponenten;
                    verbindemitUmgebung(R : roboter);
                    erhalteTeil;
                    schliesseundpresse;
                    oberePositioneingenommen;
                    senkeundoeffne;
                    unterePositioneingenommen;
                    gebeTeilab;
                    bewegezumittlererPosition;
                    mittlerePositioneingenommen;
  VALUATION   [installiere] Position = unten;
              [verbindemitUmgebung(R)] Bediener = R;
              [oberePositioneingenommen]    Position = oben;
              [mittlerePositioneingenommen] Position = mitte;
              [unterePositioneingenommen]   Position = unten;
  INTERACTION installiereKomponenten >> SO.installiere(aus),
                                        SM.installiere(aus),
                                        SU.installiere(an);
              SO.schaltean >> oberePositioneingenommen;
              SM.schaltean >> mittlerePositioneingenommen;
              SU.schaltean >> unterePositioneingenommen;
  BEHAVIOR  PROCESS Arbeitszyklus =
             ( erhalteTeil ->
               { Laenge(Arm1(Bediener))<p2 } schliesseundpresse ->
                 oberePositioneingenommen -> senkeundoeffne ->
                 unterePositioneingenommen -> gebeTeilab ->
               { Laenge(Arm2(Bediener))<p1 } bewegezumittlererPosition ->
                 mittlerePositioneingenommen -> Arbeitszyklus);
            ( installiere -> installiereKomponenten ->
              verbindemitUmgebung -> Arbeitszyklus );
END TEMPLATE;
```

p1 und p2 bezeichnen fest vorgegebene Längenangaben. Bislang erfolgte die Kommunikation zwischen Objekten immer nur entlang von Unterobjektbeziehungen. Unterobjektbeziehungen bewirken hierarchische Objektstrukturen. In manchen Fällen erscheint es allerdings notwendig, daß Objekte auch Zugriff auf andere als Unterobjekte haben. Solche Objektreferenzen können über objektwertige Attribute aufgebaut werden. So ist es im gegebenen Fall notwendig, daß die Presse mit dem Roboter kommuniziert, damit sichergestellt wird, daß die Ereignisse `bewegezumittlererPosition` und `schliesseundpresse` erst erfolgen dürfen, wenn der jeweilige Roboterarm die Presse verlassen hat. Ohne eine direkte Referenz von Presse zu Roboter könnte eine solche Bedingungen nur auf der Ebene des gemeinsamen Oberobjekts, sprich der Fertigungszelle, ausgedrückt werden. Dies würde aber auch voraussetzen, daß die genannten Ereignisse mit Ereignissen der Fertigungszelle synchronisiert werden. Eine dritte Lösung ergäbe sich natürlich durch den Einbau eines geeigneten Sensors.

3.7 Das Ablageförderband

```
TEMPLATE Ablagefoerderband
  EVENTS      BIRTH installiere;
                    starte;
                    haltean;
                    halteanzumLaden;
                    erhalteTeil;
                    gebeTeilab;
  BEHAVIOR    PROCESS StehendesBand =
                (starte -> LaufendesBand);
              PROCESS LaufendesBand =
                (erhalteTeil -> LaufendesBand |
                 haltean -> StehendesBand |
                 halteanzumLaden -> Ladephase);
              PROCESS Ladephase =
                ( gebeTeilab -> starte -> LaufendesBand);
              (installiere -> StehendesBand);
END TEMPLATE;
```

3.8 Das Handhabungsgerät

```
TEMPLATE Manipulator
  DATA TYPES  Real;
  TEMPLATES   Lichtschranke, Potentiometer;
  SUBOBJECTS  Lsensor : lichtschranke;
              Rsensor : lichtschranke;
              Laengenmesser  : potentiometer;
  ATTRIBUTES  DERIVED Laenge : real;
  EVENTS      BIRTH installiere;
                    installiereKomponenten;
                    linkePositioneingenommen;
                    rechtePositioneingenommen;
```

```
                    erhalteAuftrag;
                    bewegenachrechts;
                    fahrezuAblagefoerderband;
                    schliesseGreifer;
                    fahrehoch;
                    bewegenachlinks;
                    fahrezuzufuehrfoerderband;
                    oeffneGreifer;
  DERIVATION  Laenge = f4(Messwert(Laengenmesser));
  INTERACTION installiereKomponenten >> Lsensor.installiere,
                                        Rsensor.installiere,
                                        Laengenmesser.installiere;
              Lsensor.verliereStrahl >> linkePositioneingenommen;
              Rsensor.verliereStrahl >> rechtePositioneingenommen;
  BEHAVIOR  PROCESS WarteAufArbeit = (erhalteAuftrag -> Arbeite);
            PROCESS Arbeite =
              ({ Laenge=hAnfangspos} bewegenachrechts ->
               rechtePositioneingenommen -> fahrezuAblagefoerderband ->
               { Laenge=h1 } schliesseGreifer -> fahrehoch -> S11);
            PROCESS S11 =
              ({ Laenge=hAnfangspos } bewegenachlinks -> S12 |
               erhalteAuftrag -> S21);
            PROCESS S12 =
              (linkePositioneingenommen -> S13 |
               erhalteAuftrag -> S22);
            PROCESS S13 =
              (fahrezuzufuehrfoerderband -> S14 |
               erhalteAuftrag -> S23);
            PROCESS S14 =
              ({ Laenge=h2 } oeffneGreifer -> S15 |
               erhalteAuftrag -> S24);
            PROCESS S15 =
              (fahrehoch -> WarteAufArbeit | erhalteAuftrag -> S25);
            PROCESS S21 = ({ Laenge=hAnfangspos }
                            bewegenachlinks -> S22);
            PROCESS S22 = (linkePositioneingenommen -> S23);
            PROCESS S23 = (fahrezuzufuehrfoerderband -> S24);
            PROCESS S24 = ({ Laenge=h2 } oeffneGreifer -> S25);
            PROCESS S25 = (fahrehoch -> Arbeite);
            (installiere -> installiereKomponenten -> WarteAufArbeit);
END TEMPLATE;
```

Mit `h1`, `h2` und `hAnfangspos` haben wir fest vorgegebene Längenangaben bezeichnet. Die im `BEHAVIOR`-Abschnitt spezifizierte Objektmaschine ist in Abb. 5 dargestellt.

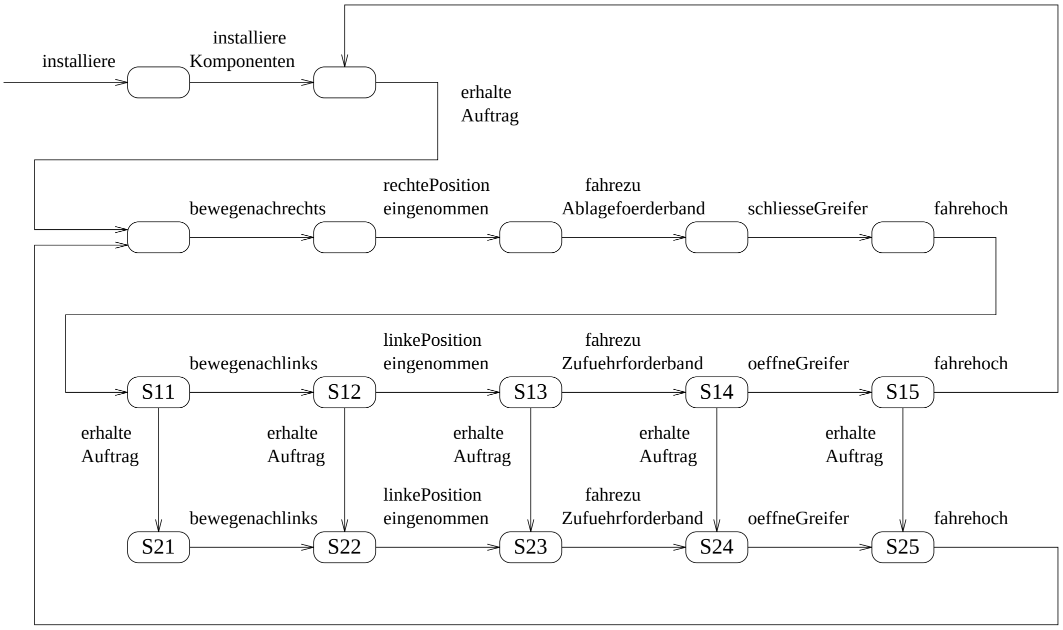


Abbildung 5: Objektmaschine `Manipulator`

3.9 Die Sensoren

```
TEMPLATE Schalter
  DATA TYPES  Schalterzustand; -- { an, aus }
  ATTRIBUTES  Zustand : schalterzustand;
  EVENTS      BIRTH installiere(Anfangszustand : schalterzustand);
                    schaltean;
                    schalteaus;
  VALUATION   [installiere(S)] Zustand = S;
              [schaltean]  Zustand = an;
              [schalteaus] Zustand = aus;
  BEHAVIOR    PROCESS Schalterleben =
                ({ Zustand=an } schalteaus -> Schalterleben,
                 { Zustand=aus } schaltean -> Schalterleben);
              (installiere -> Schalterleben);
END TEMPLATE;

TEMPLATE Potentiometer
  DATA TYPES  Real;
  ATTRIBUTES  Messwert : real;
  EVENTS      BIRTH installiere;
                    messe(Messwert : real);
  VALUATION   [messe(W)] Messwert = W;
  BEHAVIOR    PROCESS Potileben =
```

```
                (messe -> Potileben);
              (installiere -> Potileben);
END TEMPLATE;

TEMPLATE Lichtschranke
  DATA TYPES  Lichtschrankezustand; -- { Strahl, keinStrahl }
  ATTRIBUTES  Zustand : lichtschranke;
  EVENTS      BIRTH installiere;
                    verliereStrahl;
                    entdeckeStrahl;
  VALUATION   [installiere] Zustand = Strahl;
              [verliereStrahl] Zustand = keinStrahl;
              [entdeckeStrahl] Zustand = Strahl;
  BEHAVIOR  PROCESS Lichtschrankeleben =
              (verliereStrahl -> entdeckeStrahl -> Lichtschrankeleben);
            (installiere -> Lichtschrankeleben);
END TEMPLATE;
```

Literaturverzeichnis

[CGH92] S. Conrad, M. Gogolla, and R. Herzig. TROLL light: A Core Language for Specifying Objects. Informatik-Bericht 92–02, Technische Universität Braunschweig, 1992.

[LRS92] T. Lindner, A. Rüping, and E. Sekerinski. Aufgabenbeschreibung für die Fallstudie “Fertigungszelle”. Arbeitspapier, FZI Karlsruhe, 1992.

Spezifikation der abstrakten Syntax von TROLL *light* mit TROLL *light*

Rudolf Herzig

Technische Universität Braunschweig
Informatik, Abt. Datenbanken
D–38092 Braunschweig

e-mail: herzig@idb.cs.tu-bs.de

1 Vorbemerkungen

Einen wichtigen Bestandteil der TROLL *light*-Entwicklungsumgebung [VHG$^+$93] bildet das *TemplateDictionary*. Dies ist ein persistenter Speicher zur strukturierten Ablage von TROLL *light*-Objektbeschreibungen [CGH92]. Als Vorbereitung der Implementierung eines solchen Speichers wurde die abstrakte Syntax der Sprache TROLL *light* mit TROLL *light* selbst spezifiziert.

Die Beschreibung einer Sprache mit Mitteln derselben ist nicht unüblich. Zum Beispiel ist in [Gog94] die Struktur eines ER-Diagramms mit einem ER-Diagramm modelliert worden, und [Har92] enthält Ansätze zur Beschreibung eines TROLL Data-Dictionary mit TROLL. Natürlich kann die Beschreibung jeweils auch in anderen Sprachen erfolgen. So zeigt [MVL92] eine Z-Spezifikation eines ER-Dictionary, und in [GC93] wurde TROLL *light* zur Beschreibung eines Dictionary für die algebraische Spezifikationssprache ACT ONE verwendet.

Die Fallstudie ist in bestimmer Hinsicht untypisch für objektorientierte Spezifikation, da die Modellierung struktureller Aspekte im Vordergrund steht. Die Fallstudie zeigt aber insbesondere, in welcher Weise sich die Spezifikation von Objekten mit der Modellierung komplexer Werte ergänzen kann [LR89].

2 Die Spezifikation

Die Beschreibung des Speichers zur Ablage von Objektbeschreibungen erfolgt unter folgenden vereinfachenden Annahmen:

- Dynamische Aspekte werden unter der Annahme, daß es sich bei Objektbeschreibungen um Objekte ohne besonderes Verhalten handelt, vernachlässigt.
- Da davon ausgegangen wird, daß nur syntaktisch korrekte Objektbeschreibungen abgelegt werden, bleiben kontextsensitive Bedingungen unerwähnt. Prinzipiell könnten solche als statische Integritätsbedingungen formuliert werden.

- Es wird nur die Information zu einer Objektbeschreibung betrachtet, die unverzichtbar abgespeichert werden muß. Information, die sich prinzipiell aus vorhandener Information ableiten läßt, deren physische Ablage aber möglicherweise aus Effizienzgründen angezeigt ist, bleibt im vorliegenden Entwurf unberücksichtigt.

```
TEMPLATE TemplateDictionary
  DATA TYPES
    Nat, String;
  TEMPLATES
    Sortexpr, Term, Formula, Template;
  SUBOBJECTS
    SortExpressions(Id:nat):sortexpr;
    Terms(Id:nat):term;
    Formulas(Id:nat):formula;
    Templates(Name:string):template;
  EVENTS
    BIRTH create;
    addSortexpr(Id:nat, Contents:UNION(...));
    addTerm    (Id:nat, Contents:UNION(...));
    addFormula (Id:nat, Contents:UNION(...));
    addTemplate(Name:string, Contents:TUPLE(...));
  INTERACTION addSortexpr(I,C) >> SortExpressions(I).create(C);
              addTerm(I,C)     >> Terms(I).create(C);
              addFormula(I,C)  >> Formulas(I).create(C);
              addTemplate(N,C) >> Templates(N).create(C);
END TEMPLATE;
```

Abbildung 6: Objektbeschreibung `TemplateDictionary`

Abbildung 6 enthält die Beschreibung eines Speichers zur Ablage von Objektbeschreibungen. In einem Objekt, welches der Objektbeschreibung `TemplateDictionary` genügt, lassen sich Sortenausdrücke, Terme, Formeln und Objektbeschreibungen abspeichern. Bis auf Objektbeschreibungen, welche über ihre Namen unterschieden werden, werden alle anderen Unterobjekte jeweils der Einfachheit halber durch natürliche Zahlen unterschieden. Zur Vereinfachung wurden nur Einfügeoperationen spezifiziert. Die Bereiche zu den einzelnen `Contents`-Parametern sind den nachfolgenden Objektbeschreibungen zu entnehmen.

Abbildung 7 zeigt die Objektbeschreibung für Sortenausdrücke. Ein Sortenausdruck ist entweder eine Datensorte, eine Objektbezeichnersorte, ein Mengenausdruck, ein Listenausdruck, ein Multimengenausdruck, ein Tupelausdruck, ein Mapausdruck oder ein Unionausdruck.

```
TEMPLATE Sortexpr
  DATA TYPES
    String;
  ATTRIBUTES
    Contents:UNION(DSort : string,
                   OSort : string,
                   Set   : sortexpr,
                   List  : sortexpr,
                   Bag   : sortexpr,
                   Tuple : LIST(TUPLE(Id:string,
                                      Domain:sortexpr)),
                   Map   : TUPLE(SrcId:string,
                                 Src:sortexpr,
                                 DstId:string,
                                 Dst:sortexpr),
                   Union : LIST(TUPLE(Id:string,
                                      Domain:sortexpr)) );
  EVENTS
    BIRTH create(Contents:UNION(...));
  VALUATION
    [create(C)] Contents=C;
END TEMPLATE;
```

Abbildung 7: Objektbeschreibung zu Sortenausdrücken

Der Einfachheit halber werden Daten- und Objektbezeichnersorten über ihre Namen referenziert. Falls diese Sorten als Objekte mit zusätzlicher Information modelliert werden sollten, dann ist die Datensorte string durch entsprechende Objektbezeichnersorten zu ersetzen. Als wichtige Integritätsbedingung ist zu nennen, daß Verweise auf Untersortenausdrücke immer azyklisch sein müssen. Entsprechendes wird auch für alle weiteren Objektbeschreibungen angenommen.

Abbildung 8 zeigt die Objektbeschreibung für Terme. Ein Term ist entweder ein self-Term, eine Variable, eine Operation, ein if-Term oder ein select-Term.

Da zu self-Termen keine weitere Information abgespeichert werden muß, wurde als Bereich eine Datensorte dummy mit leerer Interpretation eingesetzt. Operationssymbole werden über ihre Namen referenziert. Falls Operationen als Objekte mit zusätzlicher Information modelliert werden sollten, dann ist die Datensorte string durch eine entsprechende Objektbezeichnersorte zu ersetzen. In select-Termen werden Deklarationen direkt als Liste von Tupeln von Variablennamen und Variablenbereichen modelliert.

```
TEMPLATE Term
  DATA TYPES
    Dummy, String;
  TEMPLATES
    Formula;
  ATTRIBUTES
    Contents:UNION(Self   : dummy,
                   Var    : string,
                   Op     : TUPLE(Name:string,
                                  Arguments:LIST(term)),
                   If     : TUPLE(Condition:formula,
                                  Then:term,
                                  Else:term),
                   Select : TUPLE(Select:term,
                                  From:LIST(TUPLE(Var:string,
                                                  Range:term)),
                                  Where:formula));
  EVENTS
    BIRTH create(Contents:UNION(...));
  VALUATION
    [create(C)] Contents=C;
END TEMPLATE;
```

Abbildung 8: Objektbeschreibung zu Termen

Als Erweiterung erscheint die separate Behandlung von Datenkonstanten sinnvoll, welche statt als nullstellige Operationssymbole direkt als Werte entsprechender Typen wie integer oder string abgespeichert werden können. Darüber hinaus sollte die Sorte eines Terms entweder flach als Zeichenkette oder strukturiert als Zeiger auf einen Sortenausdruck vermerkt werden.

Abbildung 9 zeigt die Objektbeschreibung für Formeln. Eine Formel ergibt sich entweder als eine konstante Formel, ein Prädikat, ein Test auf Gleichheit, ein Test auf Definiertheit, die Negation einer Formel, die logische Verküpfung zweier Formeln oder Anwendung eines Quantors auf einer Formel.

Abbildung 10 zeigt schließlich die Objektbeschreibung für Objektbeschreibungen. Eine Objektbeschreibung umfäst die Definition von Unterobjektsymbolen, Attributsymbolen, Ereignisgeneratoren, abgeleiteten Attributen, Integritätsbedingungen, Auswertungsregeln, Interaktionsregeln und Verhaltenszustandsübergängen.

```
TEMPLATE Formula
  DATA TYPES
    Dummy, String;
  TEMPLATES
    Term;
  ATTRIBUTES
    Contents:UNION(True   : dummy,
                   False  : dummy,
                   Pred   : TUPLE(Name:string,
                                  Arguments:LIST(term)),
                   Eq     : TUPLE(L:term, R:term),
                   Def    : term,
                   Not    : formula,
                   Or     : TUPLE(L:formula, R:formula),
                   And    : TUPLE(L:formula, R:formula),
                   Exists : TUPLE(Decl:LIST(TUPLE(Var:string,
                                                  Range:term)),
                                  Formula:formula),
                   Forall : TUPLE(Decl:LIST(TUPLE(Var:string,
                                                  Range:term)),
                                  Formula:formula));
  EVENTS
    BIRTH create(Contents:UNION(...));
  VALUATION
    [create(C)] Contents=C;
END TEMPLATE;
```

Abbildung 9: Objektbeschreibung zu Formeln

3 Schlußbemerkungen

Die Objektbeschreibungen zeigen beispielhaft die Spezifikation von Objekten mit komplexen Attributen. Alle Objektbeschreibungen zusammen ergeben eine Beschreibung der Syntax von TROLL *light* mit TROLL *light*. Unter Annahme eines vorhandenen Animationssystems könnten die gegebenen Objektbeschreibungen selbst animiert werden, was die direkte Eingabe von Objektbeschreibungen sowie Anfragen an Objektbeschreibungen erlauben würde.

Das beschriebene TemplateDictionary wurde als Teil zweier Diplomarbeiten mit Hilfe des objektorientierten Datenbankmanagement ObjectStore [LLOW91] auf der Basis von C++ implementiert [Ale93, Bri93].

```
TEMPLATE Template
  DATA TYPES
    String, Nat;
  TEMPLATES
    Sortexpr, Term, Formula;
  ATTRIBUTES
    Contents:
      TUPLE(SubobjSymb:   LIST(TUPLE(Name:string,
                                     Parameter:sortexpr,
                                     Osort:string)),
            AttrSymb:     LIST(TUPLE(Name:string,
                                     Domain:sortexpr)),
            EventGen:     LIST(TUPLE(Name:string,
                                     Parameter:sortexpr,
                                     Type:(birth,life,death))),
            DrvdAttr:     LIST(TUPLE(Name:string,
                                     DerivationRule:term)),
            Constraints:  LIST(formula),
            Valuation:    LIST(TUPLE(Precondition:formula,
                                     EventGen:string,
                                     AttrSymb:string,
                                     DstTerm:term)),
            Interaction:  LIST(TUPLE(Precondition:formula,
                                     SrcObject:term,
                                     SrcEventGen:string,
                                     DstEventTerm:
                                       TUPLE(DstObject:term,
                                             DstEventGen:string,
                                             Parameter:term) )),
            Transition:   LIST(TUPLE(Precondition:formula,
                                     EventGen:string,
                                     PrevState:nat,
                                     NextState:nat)));
  EVENTS
    BIRTH create(Contents:TUPLE(...));
  VALUATION
    [create(C)] Contents=C;
END TEMPLATE;
```

Abbildung 10: Objektbeschreibung zu Objektbeschreibungen

Literaturverzeichnis

[Ale93] M. Alexander. *Implementierung eines Term- und Formelauswerters für TROLL light-Spezifikationen.* Diplomarbeit, TU Braunschweig, 1993.

[Bri93] J. Brinckmann. *Implementierung des TROLL light Ausführungsmodells.* Diplomarbeit, TU Braunschweig, 1993.

[CGH92] S. Conrad, M. Gogolla, and R. Herzig. TROLL *light*: A Core Language for Specifying Objects. Informatik-Bericht 92–02, Technische Universität Braunschweig, 1992.

[GC93] M. Gogolla and I. Claßen. An Object-Oriented Design for the ACT ONE Environment. In G. Scollo, editor, *Proc. 3rd Int. Conf. on Algebraic Methodology and Software Technology AMAST.* Springer, Workshops in Computing, 1993.

[Gog94] M. Gogolla. *An Extended Entity-Relationship Model — Fundamentals and Pragmatics.* Springer LNCS 767, 1994.

[Har92] T. Hartmann. Vom konzeptionellen Modell zur Implementierung: Transformation des TROLL Data-Dictionary. In *Kurzfassungen des 4. GI-Workshops "Grundlagen von Datenbanken", Barsinghausen*, pages 56–60. ECRC, München, Technical Report ECRC-92-13, 1992.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreib. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, 1991.

[LR89] C. Lécluse and P. Richard. Modeling Complex Structures in Object-Oriented Databases. In *Proc. 8th ACM Symp. Principles of Database Systems PODS*, pages 360–368, 1989.

[MVL92] V. Mišić, D. Velašević, and B. Lazarević. Formal Specification of a Data Dictionary for an Extended ER Data Model. *The Computer Journal*, 35(6):610–622, 1992.

[VHG+93] N. Vlachantonis, R. Herzig, M. Gogolla, G. Denker, S. Conrad und H.-D. Ehrich. Towards Reliable Information Systems: The KORSO Approach. In C. Rolland, F. Bodart und C. Cauvet, Herausgeber, *Proc. 5th Int. Conf. on Advanced Information Systems Engineering (CAiSE'93)*, S. 463–482. Springer, LNCS 685, 1993.

Some Examples for TROLL *light* Templates

Martin Gogolla

Technische Universität Braunschweig
Informatik, Abt. Datenbanken
D–38092 Braunschweig

e-mail: gogolla@idb.cs.tu-bs.de

Abstract

We introduce some small examples for templates which are given in the language TROLL *light* [CGH92, GCH93, HCG94] which is intended for the specification of conceptual objects in information systems. The spectrum of our examples ranges however from rather untypical information systems like clocks, counters, or flipflops over stacks and trees to more specific information system applications like libraries.

1 Clocks

First we give a description of a normal clock, i.e., an ideal clock which is able to tick an infinite number of times (like this has been described for instance in [Hoa78]). The `Time` attribute is given by a natural number.

```
TEMPLATE Clock
  DATA TYPES Nat;
  ATTRIBUTES Time:nat;
  EVENTS     BIRTH create;
                   tick;
  VALUATION  [ create ] Time=0;
             [ tick ] Time=Time+1;
  BEHAVIOR   PROCESS ClockLife = ( tick -> ClockLife );
             ( create -> ClockLife );
END TEMPLATE;
```

Next we want to characterize a modification of the above clock by a somewhat strange clock, which is able to tick and tack. The overall idea is that ticks and tacks come in pairs. The tick is a little bit too ambitious and increases the time by two. Therefore the tack decreases time by one. But, because ticks and tacks come in pairs, the `Time` attribute will give the “correct” time if one considers the average values.

```
TEMPLATE StrangeClock
  DATA TYPES Nat;
  ATTRIBUTES Time:nat;
  EVENTS     BIRTH create;
                   tick;
                   tack;
  VALUATION  [ create ] Time=1;
             [ tick ] Time=Time+2;
             [ tack ] Time=Time-1;
  BEHAVIOR   PROCESS ClockLife =
               ( tick -> tack -> ClockLife |
                 tack -> tick -> ClockLife );
             ( create -> ClockLife );
END TEMPLATE;
```

A generalization of the above strange clock is the following parametrized strange clock. It requires that the number of ticks and tacks must be equal in a given intervall.

```
TEMPLATE ParametrizedStrangeClock
  DATA TYPES Nat;
  ATTRIBUTES Time:nat;
             Delta:nat;
             Ticks:nat;
             Tacks:nat;
  EVENTS     BIRTH create(nat);
                   tick;
                   tack;
  VALUATION  [ create(D) ] Time=0, Delta=D, Ticks=0, Tacks=0;
             [ tick ] Time=Time+2, Ticks=Ticks+1;
             [ tack ] Time=Time-1, Tacks=Tacks+1;
  BEHAVIOR   PROCESS ClockLife =
               ( { ABS((Ticks+1+Tacks)-(Time+2))<=Delta }
                     tick -> ClockLife |
                 { ABS((Ticks+Tacks+1)-(Time-1))<=Delta AND Time>0 }
                     tack -> ClockLife );
             ( { D>0 } create(D) -> ClockLife );
END TEMPLATE;
```

An interesting thing one can ask about the above specification is whether it is deadlock free, i.e., whether `Enabled(tick) OR Enabled(tack)` is true in every state of an object of the above template. Thus the question is whether `NOT(Enabled(tick)) IMPLIES Enabled(tack)` and `NOT(Enabled(tack)) IMPLIES Enabled(tick)` holds. The answer to both questions is yes.

2 Counters

Here we introduce different kinds of counters. First we give an infinite counter. The `increment` event is always allowed, whereas the `decrement` event requires that the value of the counter is greater than zero.

```
TEMPLATE InfiniteCounter
  DATA TYPES Nat;
  ATTRIBUTES Value:nat;
  EVENTS     BIRTH create;
                   increment;
                   decrement;
  VALUATION  [ create ] Value=0;
             [ increment ] Value=Value+1;
             [ decrement ] Value=Value-1;
  BEHAVIOR   PROCESS CounterLife =
               ( increment -> CounterLife |
                 { Value>0 } decrement -> CounterLife );
             ( create -> CounterLife );
END TEMPLATE;
```

It might be an interesting question to ask what happens if one drops the pre-condition in the behavior part. In this situation the value of the counter is determined by the effects of the operations of the data types. We assume that `0-1` yields `BOTTOM` and that this undefined value is propagated by the operations `_+1` and `_-1`. The next example is a finite counter.

```
TEMPLATE FiniteCounter
  DATA TYPES  Nat;
  ATTRIBUTES  Value:nat;
  EVENTS      BIRTH create;
                    increment;
                    decrement;
  CONSTRAINTS Value<42;
  VALUATION   [ create ] Value=0;
              [ increment ] Value=Value+1;
              [ decrement ] Value=Value-1;
  BEHAVIOR    PROCESS CounterLife =
                ( { Value<42 } increment -> CounterLife |
                  { Value>0 } decrement -> CounterLife );
              ( create -> CounterLife );
END TEMPLATE;
```

The `increment` event is only allowed when the value is less than `42`, the `decrement` event only when the value is greater zero; thus the finite counter always has one state out of 42 possible ones. The constraint or the pre-condition for `increment` (but not both) may be dropped without affecting the semantics. Another solution would be to introduce a special data type `NatModulo42` with carrier `0, 1, 2, ..., 39, 40, 41` and with operations `_+1` and `_-1` which yield `0-1=0` and `41+1=41`. One can then drop the pre-conditions in the behavior part; the difference is of course that in a state with `value=41` the event `increment` is not allowed in the first case, but permissible in the second one. This example demonstrates nicely that there are always alternatives in order to achieve a desired behavior. A variation of the finite counter is a cyclic counter.

```
TEMPLATE CyclicCounter
  DATA TYPES Nat;
  ATTRIBUTES Value:nat;
  EVENTS     BIRTH create;
                   increment;
                   decrement;
  VALUATION  [ create ] Value=0;
             { Value<41 } [ increment ] Value=Value+1;
             { Value=41 } [ increment ] Value=0;
             { Value>0 } [ decrement ] Value=Value-1;
             { Value=0 } [ decrement ] Value=41;
  BEHAVIOR   PROCESS CounterLife =
               ( increment -> CounterLife |
                 decrement -> CounterLife );
             ( create -> CounterLife );
END TEMPLATE;
```

Again, we could use a data type `NatModulo42'` but this time with `0-1=41` and `41+1=0` and with `[ increment ] Value=Value+1` and `[ decrement ] Value=Value-1` without pre-conditions. The templates `FiniteCounter` and `CyclicCounter` clearly point out the difference between pre-conditions in the valuation and behavior parts.

3 Flipflop

As a variation of clocks and counters we show how flipflops can be specified in TROLL *light*.

```
TEMPLATE FlipFlop
  DATA TYPES Bool;
  ATTRIBUTES Value:bool;
  EVENTS     BIRTH create(Init:bool);
                   flip;
                   flop;
  VALUATION  [ create(B) ] Value=B;
```

```
            [ flip ] Value=NOT(Value);
            [ flop ] Value=NOT(Value);
  BEHAVIOR  PROCESS FlipFlopLife =
              ( flip -> flop -> FlipFlopLife );
            ( create -> FlipFlopLife );
END TEMPLATE;
```

Flipflops have only one boolean attribute `Value`, which is modified by `flip` and `flop` events occurring alternatively in the life of a flipflop.

4 Stacks

The most popular type in computer science is without doubt the stack [BT87]. We first describe unbounded stacks of strings with list-valued attributes. Exception handling, namely the question what the result of top and pop applied to the empty stack is, is realized via the undefined element `BOTTOM` for the derived attribute `Top` and via a pre-condition in the behavior part for the event `pop`, i.e., `pop` is not allowed to occur when there are no elements is the stack.

```
TEMPLATE Stack
  DATA TYPES String;
  ATTRIBUTES Elems:LIST(string);
             DERIVED Top:string;
  EVENTS     BIRTH create;
                   push(Elem:string);
                   pop;
             DEATH destroy;
  VALUATION  [ create ] Elems=EMPTY;
             [ push(E) ] Elems=APPEND(Elems,E);
             [ pop ] Elems=DELETE(Elems,CNT(Elems));
  DERIVATION Top=IF CNT(Elems)>0 THEN GET(Elems,CNT(Elems))
                                 ELSE BOTTOM FI;
  BEHAVIOR   PROCESS StackLife =
               ( push(E) -> StackLife |
                 { CNT(Elems)>0 } pop -> StackLife |
                 { CNT(Elems)=0 } destroy -> POSTMORTEM );
             ( create -> StackLife );
END TEMPLATE;
```

Secondly, we specify bounded stacks of strings with stack entries as sub-objects. Stack entries are created with an initial value and are destroyed when they are no longer needed.

```
TEMPLATE StackEntry
  DATA TYPES String;
  ATTRIBUTES Value:string;
  EVENTS     BIRTH create(Elem:string);
             DEATH destroy;
  VALUATION  [ create(E) ] Value=E;
  BEHAVIOR   ( create -> destroy -> POSTMORTEM );
END TEMPLATE;
```

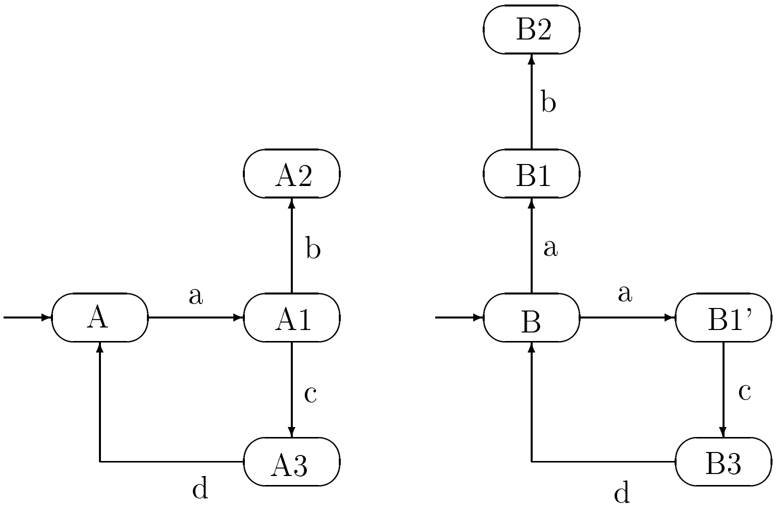
Objects of sort `stack` have stack entries as sub-objects. A `push` event in the stack object calls for a `create` event in a stack entry object; analogously, a `pop` event in the stack object calls for a `destroy` event in a stack entry object.

```
TEMPLATE Stack
  DATA TYPES  String, Nat;
  TEMPLATES   StackEntry;
  SUBOBJECTS  Elems(nat):stackEntry;
  ATTRIBUTES  DERIVED Top:string;
  EVENTS      BIRTH create;
                    push(Elem:string);
                    pop;
              DEATH destroy;
  CONSTRAINTS CNT(Elems)<42;
  INTERACTION push(E) >> Elems(CNT(Elems)).create(E);
              pop >> Elems(CNT(Elems)).destroy;
  DERIVATION  Top=IF CNT(Elems)>0 THEN Value(Elems(CNT(Elems)))
                                  ELSE BOTTOM FI;
  BEHAVIOR    PROCESS StackLife =
                ( push(E) -> StackLife |
                  { CNT(Elems)>0 } pop -> StackLife |
                  { CNT(Elems)=0 } destroy -> POSTMORTEM );
              ( create -> StackLife );
END TEMPLATE;
```

A nice declarative feature of TROLL *light* is pointed out here: The boundedness requirement is given simply by an appropriate constraint. Of course, in this case it could be given alternatively by a pre-condition for a `push` event.

5 Milner's Counter Example

In [Mil80] Robin Milner explains why the two processes $a.(P+Q)$ and $a.P+a.Q$ are not equivalent although in classical automata theory for example the two following non-deterministic machines are.



We take the same view as Milner. But let us first specify both above automata in TROLL *light*.

```
TEMPLATE A
  EVENTS      BIRTH create;
                    a; c; d;
              DEATH b;
  BEHAVIOR    PROCESS ALife =
                ( a -> ( b -> POSTMORTEM |
                         c -> d -> ALife ) )
              ( create -> ALife );
END TEMPLATE;
```

```
TEMPLATE B
  EVENTS      BIRTH create;
                    a; c; d;
              DEATH b;
  BEHAVIOR    PROCESS BLife =
                ( a -> b -> POSTMORTEM |
                  a -> c -> d -> BLife )
              ( create -> BLife );
END TEMPLATE;
```

The above two specifications are not equivalent with respect to the semantics sketched in [GCH93]: For instance, objects for template `A` will only have one transition from the initial state to an appropriate successor state with label {`a`}; but for objects of template `B` there will be two transitions from the initial state to appropriate successor states with label {`a`}.

6 Trees

If we want to specify binary trees then the following template is appropriate for this task. The specification introduces a template called `Node` and a corresponding object sort `node`. Objects belonging to this template are the nodes of the binary tree. We have (for each object of sort `node`) an infinite number of sub-object names, for instance `Children("BlueNode")` and `Children("RedNode")`. We have decided to use a parameterized sub-object in order to demonstrate this possibility. For the binary tree example a reasonable alternative would be to introduce two non-parameterized sub-objects `Left, Right:node`. We also have two non-parameterized, data-valued attributes, namely `Content` and `Total`. In this example nodes can be created and their attribute `Content` can be updated; in nodes children can be created, and nodes can be destroyed.

```
TEMPLATE Node
  DATA TYPES  String, Nat;
  SUBOBJECTS  Children(Name:string):node;
  ATTRIBUTES  Content:nat;
              DERIVED Total:nat;
  EVENTS      BIRTH create;
                    updateContent(Content:nat);
                    createChild(Name:string);
              DEATH destroy;
  CONSTRAINTS CNT(Children)<=2;
  VALUATION   [ updateContent(C) ] Content=C;
  DERIVATION  Total = SUM ( SELECT Total(PRJ2(C))
                            FROM   C IN Children ) + Content;
  INTERACTION createChild(N) >> Children(N).create;
  BEHAVIOR    PROCESS NodeLife =
                ( updateContent -> NodeLife |
                  createChild -> NodeLife |
                  { CNT(Children)=0 } destroy -> POSTMORTEM );
              ( create -> updateContent -> NodeLife );
END TEMPLATE
```

The above constraint requires that nodes can have at most two children. This is established in the following way. Only a finite number of sub-object names take a defined value. Therefore the term `Children` can also be viewed as a term of sort `SET(TUPLE(string,node))` and in each state this term is evaluated to the finite set of `(string,node)`-tuples representing the children of the current node. The cardinality of this set is restricted above (`CNT` stands for count). In the example we even have a recursively defined attribute because `Total` appears on the right-hand side of its defining equation. The inner `SELECT` term is a (bag-valued) term of our query calculus. In the `SELECT` expression, `Children` is used as a term of sort `SET(TUPLE(nat,node))`. The term describes that we have to sum up the values of the attribute `Total` of all existing sub-objects of the considered object and have to add the value of the attribute `Content`. The interaction formula expresses that whenever the event `createChild` occurs in the super-object the event `create` occurs in the sub-object. The example shows event sequences (for instance `create -> updateContent -> NodeLife`) as well as event dependent branchings (for instance `updateContent -> NodeLife | createChild -> NodeLife`).

7 Dining Philosophers

One of the most prominent examples for a concurrent system is the college with the dining philosophers [Hoa78]. The original system consists of five philosophers spending their time thinking and eating. In order to eat they need two (from five available) forks. We model the system with forks and philosophers as subobjetcs. Forks have a simple structure: they can be taken and afterwards dropped.

```
TEMPLATE Fork
  EVENTS   BIRTH create;
                 beTaken;
                 beDropped;
  BEHAVIOR PROCESS ForkLife =
             ( beTaken -> beDropped -> ForkLife );
           ( create -> ForkLife );
END TEMPLATE;
```

In the description of philosophers we abstract from the two important events `think` and `eat` and only mention the events concerning the forks. The sharing of forks is modeled via object-valued attributes.

```
TEMPLATE Philosopher
  TEMPLATES   Fork;
  ATTRIBUTES  LeftFork:fork;
              RightFork:fork;
  EVENTS      BIRTH sitDown(LF:fork, RF:fork);
                    takeLeft;
                    takeRight;
                    dropLeft;
                    dropRight;
  CONSTRAINTS DEF(LeftFork) AND DEF(RightFork);
  VALUATION   [sitDown(LF,RF)] LeftFork = LF, RightFork = RF;
  INTERACTION takeLeft  >> LeftFork.beTaken;
              takeRight >> RightFork.beTaken;
              dropLeft  >> LeftFork.beDropped;
              dropRight >> RightFork.beDropped;
```

```
  BEHAVIOR    PROCESS PhiloLife =
                ( takeLeft -> takeRight ->
                  dropLeft -> dropRight -> PhiloLife );
              ( sitDown -> PhiloLife );
END TEMPLATE;
```

The complete system now involves the philosophers and the forks as subobjects. Before a new philosopher enters the scene both his forks must be available, i.e., both fork objects must exist; otherwise the event `sitDown` cannot take place because it would violate the above constraint in template `Philosopher`.

```
TEMPLATE DiningTable
  DATA TYPES  Nat;
  TEMPLATES   Philosopher, Fork;
  SUBOBJECTS  Philo(No:nat):philosopher;
              Fork(No:nat):fork;
  EVENTS      BIRTH begin;
                    addFork(No:nat);
                    addPhilo(No:nat, N:nat, M:nat);
  INTERACTION addFork(No)       >> Fork(No).create;
              addPhilo(No,N,M) >> Philo(No).sitDown(Fork(N), Fork(M));
  BEHAVIOR    PROCESS Dinner =
                ( addFork  -> Dinner |
                  { N<>M } addPhilo(No,N,M) -> Dinner );
              ( begin -> Dinner );
END TEMPLATE;
```

We have decided to allow for a rather liberal order of events during dinner. A more specific configuration would be introduced by the following `Dinner` process.

```
PROCESS
  Dinner =
    ( addFork(0) -> addFork(1) -> addFork(2) ->
      addFork(3) -> addFork(4) ->
      addPhilo(0,0,1) -> addPhilo(1,1,2) -> addPhilo(2,2,3) ->
      addPhilo(3,3,4) -> addPhilo(4,4,0) -> () );
  ( begin -> Dinner );
END TEMPLATE;
```

Thus we have exactly five philosophers; philosopher 0 uses forks 0 and 1, philosopher 1 uses forks 1 and 2, and so on. One interesting question concerning the `DiningTable` description is where to put the requirement that philosophers need two distinct forks. Above we have required that the two natural number paramenters for the `addPhilo` event must be distinct. Another (in this case equivalent) solution would be to specify the constraint `LeftFork<>RightFork` in the `Philosopher` template.

8 Libraries

We give two descriptions of libraries where data about books and their authors is to be stored. The first version of library is in some sense a non-first-normal-form schema [SS86], because we introduce a single relation `Books` which has a list-valued attribute `Authors`. In order to keep the specification small we require that a book has at most two authors.

```
TEMPLATE Library
  DATA TYPES  String;
  ATTRIBUTES  Books:SET(TUPLE(Authors:LIST(string),Title:string));
  EVENTS      create;
              addBook(Authors:LIST(string),Title:string);
  CONSTRAINTS FORALL T IN Books
                ( CNT(Authors(T))=1 OR CNT(Authors(T))=2 );
  VALUATION   [create] Books=EMPTY;
              [addBook(AL,T)] Books=ADD(Books,(AL,T));
  BEHAVIOR    PROCESS LibraryLife =
                ( {CNT(AL)=1 OR CNT(AL)=2} addBook(AL,T) -> LibraryLife );
              create -> LibraryLife;
END TEMPLATE
```

The second description of our library can be looked at as the first-normal-form [Ken83] version of the first description. It introduces two relations `Authors` and `Titles` which are joined by an artificial attribute `No` representing a surrogate value for a single book entry. The attribute `No` is not directly modifiable from outside.

```
TEMPLATE Library
  DATA TYPES  String, Bool, Nat;
  ATTRIBUTES  Authors(No:nat,Author:string):bool;
              Titles(No:nat,Title:string):bool;
              Next:nat;
              DERIVED Books:SET(TUPLE(Authors:SET(string),Title:string));
  EVENTS      create;
              addBook(Authors:LIST(string),Title:string);
  CONSTRAINTS FORALL T IN Books
                ( CNT(Authors(T))=1 OR CNT(Authors(T))=2 );
  VALUATION   [create] Next=0;
              [addBook(AL,T)] Titles(Next,T)=TRUE, Next=Next+1;
              { CNT(AL)=1 }
                [addBook(AL,T)] Authors(Next,AL[1])=TRUE;
              { CNT(AL)=2 }
                [addBook(AL,T)] Authors(Next,AL[1])=TRUE,
                                Authors(Next,AL[2])=TRUE;
```

```
  DERIVATION  Books = (SELECT (SELECT Author(A)
                               FROM   A IN Authors
                               WHERE  No(A)=No(T) ), Title(T)
                       FROM   T IN Titles);
  BEHAVIOR    PROCESS LibraryLife =
                ( {CNT(AL)=1 OR CNT(AL)=2} addBook(AL,T) -> LibraryLife );
              create -> LibraryLife;
END TEMPLATE
```

Both versions of the library are to a large extend equivalent, because both have the same events `create` and `addBook` and the same attribute `Books` which serves as the main observation for the library. One difference is that the `Books` attribute in the second version is of sort `SET(TUPLE(SET(string),string))` which is in contrast to `SET(TUPLE(LIST(string),string))` in the first version; this is because the query calculus for TROLL *light* does not include a sorting feature yet. The second version can be regarded as a kind of implementation of the first version on a lower level.

Bibliography

[BT87] J. Bergstra and J.V. Tucker. The Inescapable Stack: An Exercise in Algebraic Specification with Total Functions. Technical report, University of Amsterdam, 1987.

[CGH92] S. Conrad, M. Gogolla, and R. Herzig. TROLL *light*: A Core Language for Specifying Objects. Informatik-Bericht 92–02, TU Braunschweig, 1992.

[GCH93] M. Gogolla, S. Conrad, and R. Herzig. Sketching Concepts and Computational Model of TROLL *light*. In A. Miola, editor, *Proc. 3rd Int. Conf. Design and Implementation of Symbolic Computation Systems (DISCO'93)*, pages 17–32. Springer, Berlin, LNCS 722, 1993.

[HCG94] R. Herzig, S. Conrad, and M. Gogolla. Compositional Description of Object Communities with TROLL light. In C. Chrisment, editor, *Proc. Basque Int. Workshop on Information Technology (BIWIT'94)*. Cepadues Society Press, France, 1994.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Ken83] W. Kent. A Simple Guide to Five Normal Forms in Relational Database Theory. *Communications of the ACM*, 26:120–125, 1983.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Springer, Berlin, 1980.

[SS86] H.-J. Schek and M.H. Scholl. The Relational Model with Relation-Valued Attributes. *Information Systems*, 2(2):105–133, 1986.

Technische Universität Braunschweig
Informatik-Berichte ab Nr. 89-02

89-02	H.-D.Ehrich, G.Engels, M.Gogolla, G.Saake	Abstracts des Workshops “Grundlagen von Datenbanken”
89-03	H.Langendörfer, R.Cordes, K.Laufer, T.Leipnitz	Implementation einer Dateischnittstelle für laser-optische Speicherplatten unter UNIX
89-04	K.Hülsmann, G.Saake	Theoretical Foundations of Handling Large Substitution Sets in Temporal Integrity Monitoring
89-05	P.Tillert, T.Worsch	A&P - Eine einfache Programmiersprache für Parallelrechner
89-06	H.Langendörfer, M.Hofmann	Das Hypertextsystem CONCORDE - Aktivitäten der Forschungsgruppe Bürokommunikation im Bereich Hypertext 1988 - 1989
90-01	U.Karge, M.Gogolla	Formal Semantics of SQL Queries
90-02	U.W.Lipeck, S.Braß, G.Saake	Kurzfassungen des 2.Workshops “Grundlagen von Datenbanken”, Volkse 5.-8. Juni 1990
90-03	B.Meyer, G.D.Westerman, M.Gogolla	QUEER: A Prolog-based Prototype for an Extended ER Approach
90-04	P.Löhr-Richter	Validation: Ein methodischer Schritt zu fehlerfreien Datenbankentwürfen
90-05	G.Engels, M.Gogolla, U.Hohenstein, K.Hülsmann, P.Löhr-Richter, G.Saake, H.-D.Ehrich	Conceptual Modelling of Database Applications Using an Extended ER Model
91-01	M.Hofmann, H.Langendörfer	User Interface and Navigation Facilities of the Hypertext System CONCORDE
91-02	U.Hohenstein, G.Engels	SQL/EER - Syntax and Semantics of an Entity-Relationship-Based Query Language
91-03	G.Saake, A.Sernadas	Information Systems - Correctness and Reusability (Workshop IS-CORE’91, Selected Papers)
91-04	R.Jungclaus, G.Saake, T.Hartmann, C.Sernadas	Object-Oriented Specification of Information Systems: The TROLL Language
92-01	F.-J.Grosch, G.Snelting	Polymorphic Components for Monomorphic Languages
92-02	S.Conrad, M.Gogolla, R.Herzig	TROLL *light*: A Core Language for Specifying Objects
93-01	B.Fischer	A New Feature Unification Algorithm
93-02	G.Snelting	Perspektiven der Softwaretechnologie
93-03	G.Snelting, A.Zeller	Inferenzbasierte Werkzeuge in NORA
93-04	W.Rönsch, J.Schüle	Parallelisierung im Wissenschaftlichen Rechnen
93-05	P.Löhr-Richter, G.Reichwein	Object Oriented Life Cycle Models
93-06	M.Krone, G.Snelting	On the Inference of Configuration Structures from Source Code
93-07	S.Schwiderski, T.Hartmann, G.Saake	Monitoring Temporal Preconditions in a Behaviour Oriented Object Model
93-08	T.Hartmann, G.Saake	Abstract Specification of Object Interaction
93-09	G.Snelting, B.Fischer, F.-J.Grosch, M.Kievernagel, A.Zeller	Die inferenzbasierte Softwareentwicklungsumgebung NORA
93-10	C.Lindig	STYLE – A Practical Type Checker for SCHEME
93-11	H.-D.Ehrich	Beiträge zu KORSO- und TROLL *light*-Fallstudien