Zur Übersetzung von E/R-Schemata nach SPECTRUM[‡]

Rudolf Hettler[*]

12. November 1993

Zusammenfassung

Dieser Bericht zeigt, wie sich Entity/Relationship-Schemata in die axiomatische Spezifikationssprache SPECTRUM übersetzen lassen. Damit wird Anwendern von SPECTRUM die Möglichkeit eröffnet, statische Anteile der zu spezifizierenden Software in leicht verständlicher und übersichtlicher Form als E/R-Diagramme zu beschreiben, ohne dabei den formalen Rahmen der Spezifikationssprache zu verlassen. Der Ansatz stellt eine Möglichkeit dar, die formale Entwicklung (datenbankorientierter) Informationssysteme überschaubar und praktikabel zu gestalten. Er wurde bei der Spezifikation der funktionalen Anforderungen an das Patientendaten-Verwaltungssystem HDMS-A ([LCFW92, SNM+93]) eingesetzt.

Inhaltsverzeichnis

[‡]Diese Arbeit wird vom Bundesministerium für Forschung und Technik als Teil des Verbundprojekts "Korrekte Software (KORSO)" gefördert.

[*]Fakultät für Informatik der Technischen Universität München, D-80290 München

1 Einleitung

Diese Arbeit zeigt auf, wie die in der funktionalen Essenz von HDMS-A [SNM+93] verwendeten Entity/Relationship-Schemas in die Spezifikationssprache SPECTRUM übersetzt werden können. Dadurch wird das Datenmodell der formalen Spezifikation und Verifikation in SPECTRUM zugänglich. Eine detaillierte Beschreibung und Motivation der zur Erstellung der funktionalen Essenz gewählten Vorgehensweise findet sich in [Huß93].

Ausgehend von der Annahme, daß die im E/R-Diagramm verwendeten Attributtypen bereits als abstrakte Datentypen in SPECTRUM spezifiziert sind, gibt Abschnitt 2 an, wie die Spezifikation von Entitytypen aussieht. Darauf aufbauend präsentiert Abschnitt 3 die Spezifikation einer E/R-Datenbanksorte, die es erlaubt Entities dieser Entitytypen und Beziehungen zwischen ihnen zu speichern. Abschnitt 4 schließlich beschäftigt sich mit der Rolle und Übersetzung statischer Integritätsbedingungen auf der E/R-Datenbank.

2 Entitytypen

2.1 Vorbemerkung

Entities sind Verbunde, wie sie z.B. von den Records der Programmiersprache PASCAL her bekannt sind. Die Felder dieser Verbunde heißen *Attribute*. Die Bezeichner dieser Felder heißen *Attributbezeichner*, ihre Werte heißen *Attributwerte*. Die Sorten der Attributwerte werden als *Attributsorten* bezeichnet. Die Sorte einer Entity wird als *Entitytyp* bezeichnet.

Zur Beschreibung von Entitytypen gibt es viele verschiedene Darstellungen. Die in [SNM+93] verwendete Notation listet unter dem Bezeichner des Entitytyps alle Attributbezeichner zusammen mit ihren Sorten auf. Besondere Eigenschaften eines Attributs wie *mandatory* (das Attribut muß immer definiert sein) oder *primary key* (das Attribut ist Teil des Primärschlüssels) werden in Klammern hinter das Attribut geschrieben.

Als Beispiel wird hier die Beschreibung der Entity HK_Befund aus [SNM+93] angegeben:

HK_Befund

BefundId:BefundId;	(*mandatory; primary key*)
Befundungsdatum:DateTime;	(*mandatory*)
Befund:Befund;	(*mandatory*)
Befundbrief:Befundbrief;	

2.2 Modellierung in SPECTRUM

Wie oben bereits erwähnt, können Attribute einer Entity optional oder zwingend sein. Zwingende Attribute müssen immer mit einem definierten Wert besetzt sein, optionale Attribute dürfen auch undefiniert sein. Undefinierte Attribute werden in diesem Ansatz jedoch nicht durch $\perp$ modelliert, da es möglich sein soll, auch solche Attribute durch implementierbare Funktionen abzufragen. Vielmehr wird zu jeder

Attributsorte ein von allen Elementen dieser Sorte unterschiedliches, definiertes Element UNDEF hinzugenommen. Für zwingende Attribute wird gefordert, daß diese nie den Wert UNDEF annehmen dürfen.

```
Attr ={

data Attr α = UNDEF | attr(! ↓ α);

}
```

Im folgenden soll der Typ E von Entities spezifiziert werden, die aus den Attributen mit den Bezeichnern $\mathtt{attr}_1, \ldots, \mathtt{attr}_n$ bestehen. Die Sorten dieser Attribute sind $\mathtt{A}_1, \ldots, \mathtt{A}_n$ (diese Sorten müssen nicht unbedingt alle unterschiedlich sein). Die zwingenden Attribute von E seien $\mathtt{attr}_{i_1}, \ldots, \mathtt{attr}_{i_m}$ $(1 \leq i_j \leq n)$, der Primärschlüssel von E bestehe aus den Attributen $\mathtt{attr}_{k_1}, \ldots, \mathtt{attr}_{k_l}$ $(1 \leq k_j \leq n)$. Es wird angenommen, daß die Attributsorten bereits in einer Spezifikation Attribute definiert sind. Dann ergibt sich für den Entitytyp E folgende SPECTRUM-Spezifikation:

```
 E = { enriches Attr + Attribute;

-- Entitysorte
sort E;

-- Schlüsselsorte
sortsyn KeyE = A_k1 × ... × A_kl ;

createE : Attr A_1 × ... × Attr A_n → E;
createE strict;
E freely generated by createE;

attr_1 : E → Attr A_1 ;
           ...
attr_n : E → Attr A_n ;
attr_1, ..., attr_n strict total;

axioms ∀ a_1, ..., a_n in
-- attr_i1 bis attr_im sind zwingende Attribute
δ(createE(a_1, ..., a_n)) ⇔ a_i1 ≠ UNDEF ∧ ... ∧  a_im ≠ UNDEF;

-- Selektoren
δ(createE(a_1, ..., a_n)) ⇒ attr_1(createE(a_1, ..., a_n)) = a_1 ;
                ...
δ(createE(a_1, ..., a_n)) ⇒ attr_n(createE(a_1, ..., a_n)) = a_n ;
endaxioms;

--Zugriff auf einzelne Attribute
setattr_1 : E × Attr A_1 → E;
                ...
setattr_n : E × Attr A_n → E;
setattr_1, ..., setattr_n strict;

axioms ∀ e, a_1, ..., a_n in
setattr_1(e,a_1) = createE(a_1, attr_2 e, ..., attr_n e);
```

```
                ...
setattr_n(e,a_n) = createE(attr_1 e, ..., attr_{n-1} e, a_n);
endaxioms;


--Zugriff auf Schlüssel
keyE : E → KeyE;
keyE strict total;

axioms ∀ e in
keyE e = (↓(attr_{k_1} e), ..., ↓(attr_{k_l} e));
endaxioms;
}
```

Bemerkungen

- Da auf die Attribute einer Entity über ihre Bezeichner (in der Spezifikation ist jedem Bezeichner ein Selektor zugeordnet) zugegriffen wird, ist die Reihenfolge der Attribute ohne Belang. Die Anordnung der einzelnen Attribute im Aufruf von `createE` ist daher willkürlich.

- In dieser Spezifikation wird zwar bereits eine Schlüsselsorte eingeführt, es wird aber noch nicht gefordert, daß die Schlüsselsorte Entities eindeutig identifiziert (Schlüsseleigenschaft). Über Schlüsseleigenschaften zu reden macht erst im Zusammenhang mit E/R-Schemata bzw. Datenbanken Sinn.

3 Entity/Relationship-Diagramme

3.1 Vorbemerkung

Bei der Entity-Relationship-Modellierung wird ein Datenbestand als eine Menge von Entities gesehen, die zueinander in Beziehung stehen können. Ein E/R-Diagramm beschreibt die Struktur eines solchen Datenbestandes, d.h. es gibt an, welche Entitytypen existieren und welche Entitytypen Beziehungen eingehen können (*Relationshiptypen*). Ein solches E/R-Diagramm läßt sich in SPECTRUM durch die Spezifikation einer zugehörigen *E/R-Datenbank* modellieren. Eine E/R-Datenbank ist eine Struktur, die es erlaubt, Entities der im E/R-Diagramm angegebenen Entitytypen und ihre Beziehungen gemäß der angegebenen Relationshiptypen zu speichern. In einer solchen E/R-Datenbank muß für jeden beteiligten Entitytyp die Schlüsseleigenschaft gelten, d.h. seine Entities müssen innerhalb der Datenbank über ihren Schlüssel eindeutig identifizierbar sein. Abschnitt 3.2 zeigt auf, wie aus einem E/R-Diagramm die Spezifikation der zugehörigen E/R-Datenbank gewonnen werden kann.

Neben der reinen Strukturinformation enthält ein E/R-Diagramm zusätzliche Bedingungen, die der modellierte Datenbestand erfüllen muß. Dabei handelt es sich um *Grade* der Relationshiptypen. Diese Grade sind ein Spezialfall statischer Integritätsbedingungen auf der E/R-Datenbank. Statische Integritätsbedingungen und ihre Auswirkungen auf die SPECTRUM-Spezifikation der E/R-Datenbank werden in Abschnitt 4 behandelt.

3.2 Modellierung in SPECTRUM

In diesem Abschnitt wird die schematische Übersetzung eines E/R-Diagramms mit den Entitytypen $\mathtt{E}_1, \ldots, \mathtt{E}_n$ und den Relationshiptypen $\mathtt{R}_1, \ldots, \mathtt{R}_m$ in die SPECTRUM-Spezifikation der zugehörigen E/R-Datenbank angegeben. Dazu wird zunächst eine Sorte für die Datenbank eingeführt.

```
sort Db;
```

Diese Datenbanksorte wird als Tupel bestehend aus Mengen von Entities der beteiligten Entitytypen und Relationships betrachtet. Relationships zwischen Entities werden als Teilmenge des Kreuzprodukts der Schlüsselsorten der am jeweiligen Relationshiptyp beteiligten Entitytypen dargestellt. Es wird also folgender Konstruktor für die Sorte Db eingeführt:

```
mkdb : (Set E1,                          -- Entitytyp E1
          ⋮
        Set En,                          -- Entitytyp En
        Set (KeyE_R11 × KeyE_R12),       -- Relationshiptyp R1
          ⋮
        Set (KeyE_Rm1 × KeyE_Rm2)        -- Relationshiptyp Rm
       ) → Db;

mkdb strict;

Db freely generated by mkdb;
```

Dieser Konstruktor ist partiell, da die an einem E/R-Schema beteiligten Entitymengen und Relationships folgende Bedingungen erfüllen müssen:

1. Die in der Datenbank enthaltenen Entities sind durch ihren Primärschlüssel eindeutig gekennzeichnet (Schlüsseleigenschaft).

2. In den Relationships kommen keine Entities vor, die nicht in der Datenbank enthalten sind.

Diese Bedingungen für die Datenbank lassen sich in SPECTRUM durch folgendes Axiom ausdrücken:

```
axioms ∀se1,...,sen,rel1,...,relm in
δ(mkdb(se1,...,sen,rel1,...,relm)) ⇔
  -- Bedingung 1.
  (∀e,e'. e ∈ se1 ∧ e'∈ se1 ∧ e≠e' ⇒ keyE1 e ≠ keyE1 e') ∧
                                ⋮
  (∀e,e'. e ∈ sen ∧ e'∈ sen ∧ e≠e' ⇒ keyEn e ≠ keyEn e') ∧
  -- Bedingung 2.
  (∀k,k'. (k,k') ∈ rel1 ⇒ ∃e,e'. e ∈ se_R11 ∧ e' ∈ se_R12 ∧
                           keyE_R11 e = k ∧ keyE_R12 e' = k'
  ) ∧
                                ⋮
  (∀k,k'. (k,k') ∈ relm ⇒ ∃e,e'. e ∈ se_Rm1 ∧ e' ∈ se_Rm2 ∧
```

```
                                keyE_{R_m1} e = k ∧ keyE_{R_m2} e' = k'
  );
endaxioms;
```

Auf der so spezifizierten Datenbanksorte werden nun eine Reihe von abgeleiteten Funktionen spezifiziert, die ein bequemes Umgehen mit dem E/R-Schema ermöglichen. Dazu werden zunächst Zugriffsfunktionen auf die einzelnen Komponenten der Datenbank definiert.

```
entE_1 : Db → Set E_1;
          ⋮
entE_n : Db → Set E_n;
R_1 : Db → (E_{R_11} × E_{R_12} → Bool);
          ⋮
R_m : Db → (E_{R_m1} × E_{R_m2} → Bool);

entE_1, ..., entE_n, R_1, ..., R_m strict total;

axioms ∀db,se_1,...,se_n,rel_1,...,rel_m in
db = mkdb(se_1,...,se_n,rel_1,...,rel_m) ⇒
   entE_1 db = se_1 ∧
                  ⋮
   entE_n db = se_n ∧
   R_1 db = (λ(e,e'). (keyE_{R_11} e,keyE_{R_12} e') ∈ rel_1) ∧
               ⋮
   R_m db = (λ(e,e'). (keyE_{R_m1} e,keyE_{R_m2} e') ∈ rel_m);
endaxioms;
```

Man beachte, daß die Funktionen $R_1, \ldots, R_m$ nicht einfach nur den Zugriff auf die jeweilige Komponente der Datenbank erlauben, sondern diesen Zugriff bereits etwas benutzerfreundlicher verpacken.

In einem nächsten Schritt werden Funktionen zum Eintragen und Löschen von Entities sowie eine Konstante für die leere Datenbank spezifiziert.

```
emptydb : Db;

putE_1 : E_1 × Db → Db;
          ⋮
putE_n : E_n × Db → Db;

putE_1, ..., putE_n strict total;

delE_1 : E_1 × Db → Db;
          ⋮
delE_n : E_n × Db → Db;

delE_1, ..., delE_n strict;

axioms ∀db,se_1,...,se_n,rel_1,...,rel_m,e_1,...,e_n in
emptydb = mkdb(∅,...,∅,∅,...,∅);
```

```
db = mkdb(se_1,...,se_n,rel_1,...,rel_m) ⇒
   putE_1(e_1,db)  = mkdb(add(e_1,se_1),se_2,...,se_n,rel_1,...,rel_m) ∧
                              ⋮
   putE_n(e_n,db)  = mkdb(se_1,...,se_{n-1},add(e_n,se_n),rel_1,...,rel_m) ∧
   delE_1(e_1,db)  = mkdb(del(e_1,se_1),se_2,...,se_n,rel_1,...,rel_m) ∧
                              ⋮
   delE_n(e_n,db)  = mkdb(se_1,...,se_{n-1},del(e_n,se_n),rel_1,...,rel_m);
endaxioms;
```

Darüberhinaus werden Funktionen zum Herstellen (establish) und Auflösen (release) von Relationships zwischen Entities der Datenbank eingeführt.

```
estR_1 : Db × E_{R_11} × E_{R_12} → Db;
              ⋮
estR_m : Db × E_{R_m1} × E_{R_m2} → Db;

estR_1, ..., estR_m strict;

relR_1 : Db × E_{R_11} × E_{R_12} → Db;
              ⋮
relR_m : Db × E_{R_m1} × E_{R_m2} → Db;

relR_1, ..., relR_m strict total;

axioms ∀db,se_1,...,se_n,rel_1,...,rel_m in
db = mkdb(se_1,...,se_n,rel_1,...,rel_m) ⇒
   (∀e,e'. estR_1(db,e,e') =
    mkdb(se_1,...,se_n,add((keyE_{R_11} e,keyE_{R_12} e'),rel_1),rel_2,...,rel_m)) ∧
              ⋮
   (∀e,e'. estR_m(db,e,e') =
    mkdb(se_1,...,se_n,rel_1,...,rel_{m-1},add((keyE_{R_m1} e,keyE_{R_m2} e'),rel_m))) ∧
   (∀e,e'. relR_1(db,e,e') =
    mkdb(se_1,...,se_n,del((keyE_{R_11} e,keyE_{R_12} e'),rel_1),rel_2,...,rel_m)) ∧
              ⋮
   (∀e,e'. relR_m(db,e,e') =
    mkdb(se_1,...,se_n,rel_1,...,rel_{m-1},del((keyE_{R_m1} e,keyE_{R_m2} e'),rel_m)));
endaxioms;
```

Um mit der so spezifizierten Datenbank praktisch arbeiten zu können, sind Funktionen nötig, die den Zugriff auf Entities über ihre Schlüssel erlauben.

```
getE_1 : KeyE_1 × Db → E_1;
        ⋮
getE_n : KeyE_n × Db → E_n;

getE_1, ..., getE_n strict;

axioms ∀db,ke_1,...,ke_n,e_1,...,e_n in
δ(getE_1 (ke_1,db)) ⇔ ∃e_1. e_1 ∈ entE_1 db ∧ keyE_1 e_1 = ke_1;
```

```
getE1 (ke1,db) = e1 ⇔ e1 ∈ entE1 db ∧ keyE1 e1 = ke1;
                              ⋮
δ(getEn (ken,db)) ⇔ ∃en. en ∈ entEn db ∧ keyEn en = ken;
getEn (ken,db) = en ⇔ en ∈ entEn db ∧ keyEn en = ken;
endaxioms;
```

Oft werden Funktionen benötigt, die "frische", d.h. noch nicht in der Datenbank vorhandene, Schlüsselwerte generieren. Viele Datenbanksysteme stellen solche Funktionen in Form von Zahlengeneratoren zur Verfügung. Will man eine derartige Funktion spezifizieren, reicht es jedoch nicht zu fordern, daß sie einen noch nicht verwendeten Schlüsselwert abliefert. Oft werden an diese Schlüsselwerte noch zusätzliche Anforderungen gestellt, z.B. daß aus datenschutzrechtlichen Gründen der Inhalt einer Entity nicht aus ihrem Schlüsselwert berechenbar sein darf. Deshalb erhalten die hier spezifizierten Schlüsselgeneratorfunktionen als Argument ein Prädikat, das der generierte Schlüssel erfüllen muß.

```
genkeyE1 : Db × (KeyE1 → Bool) → KeyE1;
                ⋮
genkeyEn : Db × (KeyEn → Bool) → KeyEn;
genkeyE1,...,genkeyEn strict;

axioms ∀db,p in
δ(genkeyE1(db,p)) ⇔ ∃k:KeyE1. (¬∃e:E1. e ∈ entE1 db ∧ keyE1 e = k) ∧ p k;
δ(genkeyE1(db,p)) ⇒ ¬∃e:E1. e ∈ entE1 db ∧ keyE1 e = genkeyE1(db,p);
δ(genkeyE1(db,p)) ⇒ p(genkeyE1(db,p));
                  ⋮
δ(genkeyEn(db,p)) ⇔ ∃k:KeyEn. (¬∃e:En. e ∈ entEn db ∧ keyEn e = k) ∧ p k;
δ(genkeyEn(db,p)) ⇒ ¬∃e:En. e ∈ entEn db ∧ keyEn e = genkeyEn(db,p);
δ(genkeyEn(db,p)) ⇒ p(genkeyEn(db,p));
endaxioms;
```

Als letztes werden nun noch Funktionen definiert, die es erlauben Entities in der Datenbank zu verändern. Dabei darf sich der Primärschlüssel der Entity nicht ändern, da es sich sonst um keine Veränderung, sondern um ein Ersetzen einer Entity durch eine andere handeln würde.

```
updateE1 : KeyE1 × E1 × Db  → Db;
                ⋮
updateEn : KeyEn × En × Db  → Db;

updateE1, ..., updateEn strict;

axioms ∀db,se1,...,sen,rel1,...,relm,ke1,...,ken,e1,...,en in
δ(updateE1(ke1,e1,db)) ⇔
   ∃e'. e' ∈ entE1 db ∧ ke1 = keyE1 e' ∧ ke1 = keyE1 e1;
                    ⋮
δ(updateEn(ken,en,db)) ⇔
   ∃e'. e' ∈ entEn db ∧ ken = keyEn e' ∧ ken = keyEn en;

db = mkdb(se1,...,sen,rel1,...,relm) ⇒
```

```
   (δ(updateE1(ke1,e1,db)) ⇒ updateE1(ke1,e1,db) =
      mkdb(add(e1,del(getE1(ke1,db),se1)),se2,...,sen,rel1,...,relm)) ∧
                          ⋮
   (δ(updateEn(ken,en,db)) ⇒ updateEn(ken,en,db) =
      mkdb(se1,...,sen−1,add(en,del(getEn(ken,db),sen)),rel1,...,relm));
endaxioms;
```

4 Statische Integritätsbedingungen

4.1 Klassifizierung

Statische Integritätsbedingungen sind Eigenschaften, deren Gültigkeit anhand eines Zustands des Datenbestands überprüfbar ist. Insbesondere ist die Kenntnis der Geschichte des Datenbestands zur Prüfung statischer Integritätsbedingungen nicht erforderlich.

Im Rest dieses Kapitels wird zwischen zwei verschiedenen Arten von statischen Integritätsbedingungen unterschieden:

Grade von Beziehungstypen Hier wird angegeben, wie oft eine Entity an einer Relationship mindestens teilnehmen muß bzw. höchstens teilnehmen darf. Zur Formulierung dieser Art von Bedingungen gibt es mehrere Möglichkeiten (z.B. 1:n-Notation, (min,max)-Notation, siehe [SS83]). Auch die graphische Darstellung dieser Notationen in E/R-Diagrammen ist nicht einheitlich (verschiedene Arten von Kanten, Annotation der Kanten).

Allgemeine Integritätsbedingungen In diese Klasse fallen alle (komplexeren) statischen Integritätsbedingungen, die nicht durch den vorigen Punkt abgedeckt sind. Diese Bedingungen können in E/R Diagrammen nicht formuliert werden. In algebraischen Spezifikationen von E/R-Schemata lassen sie sich jedoch mit den Mitteln der Spezifikationssprache beschreiben.

4.2 Gültigkeit

Integritätsbedingungen sind Eigenschaften, die der Datenbestand eines Informationssystems immer erfüllen sollte. Es ist jedoch nicht zu erwarten, daß dies auf der Ebene primitiver Operationen wie `putE` (Einfügen einer Entity) oder `estR` (Herstellen einer Beziehung zwischen Entities) garantiert werden kann.

Im Datenbankbereich führte unter anderem dieses Problem zur Verwendung des *Transaktionskonzepts*. Integritätsbedingungen müssen dabei am Ende jeder Transaktion erfüllt sein, sie dürfen jedoch im Inneren einer Transaktion zeitweilig verletzt werden.

Im Bereich des requirements engineering, in dem bereits E/R Datenmodellierung eingesetzt wird, sollte man jedoch nicht verpflichtet sein, die Granularität von Transaktionen festzulegen. Die Integritätsbedingungen müssen aber auf jeden Fall nach Ausführung jeder dem Benutzer des Informationssystem zugänglichen Funktion (*Toplevel-Funktion*) erfüllt sein.

4.3 Modellierung in SPECTRUM

Jede statische Integritätsbedingung läßt sich als (striktes und totales) Prädikat $\texttt{C}_i$ ($1 \leq i \leq k$, k ist Anzahl der Integritätsbedingungen) über dem aktuellen Zustand des Datenmodells auffassen:

```
C_i : Db → Bool;
C_i strict total;
```

Zusätzlich wird ein Prädikat `OK` definiert, das alle $\texttt{C}_i$ (mit $\wedge$ verknüpft) zusammenfaßt:

```
OK : Db → Bool;
OK strict total;

axioms
OK db = C_1 db ∧ ... ∧ C_k db;
endaxioms;
```

Wie bereits in Abschnitt 4.2 erläutert, müssen die statischen Integritätsbedingungen nach jeder Toplevel-Funktion erfüllt sein. Deshalb muß bei der Spezifikation jeder solchen Funktion als Axiom die Forderung hinzugenommen werden, daß diese Funktion die statischen Integritätsbedingungen des Datenbestandes invariant läßt.

Beispiel: Die Toplevel-Funktion `Mitarbeiter_eintragen` eines betrieblichen Informationssystems habe die Funktionalität

```
Mitarbeiter_eintragen : Db × Mitarbeiter → Db;
```

Da es sich um eine dem Benutzer zugängliche Funktion handelt, muß sie die statischen Integritätsbedingen des Datenbestands `Db` invariant lassen, d.h. sie muß das Gesetz

```
OK db ⇒ OK(Mitarbeiter_eintragen(db,m));
```

erfüllen. Dieses Gesetz wird zur Spezifikation von `Mitarbeiter_eintragen` hinzugenommen und muß daher von jeder Implementierung dieser Funktion garantiert werden. Es stellt somit eine Beweisverpflichtung für die Entwicklung der ausführbaren Implementierung der Funktion dar.

In Abschnitt 4.1 wurde bei den statischen Integritätsbedingungen unterschieden zwischen Graden von Relationshiptypen und allgemeinen Integritätsbedingungen. Während erstere in den zu übersetzenden E/R-Diagrammen grafisch dargestellt sind, können letztere in herkömmlichen Ansätzen der E/R-Modellierung bestenfalls als erklärende informelle Kommentare zu den E/R-Diagrammen angegeben werden. Erst die Übersetzung in eine (algebraische) Spezifikationssprache erlaubt es, diese Art der Integritätsbedingungen formal zu fassen. Hier kann die volle Mächtigkeit der Sprache zur Formulierung der oben angegebenen $\texttt{C}_i$-Prädikate eingesetzt werden.

Der Rest dieses Abschnitts beschäftigt sich nun mit der Übersetzung der in den E/R-Diagrammen dargestellten Grade von Relationshiptypen in die Sprache SPECTRUM.

Grade von Beziehungstypen

Wie bereits in Abschnitt 4.1 erläutert, gibt es für diese Klasse von Integritätsbedingungen eine Vielzahl verschiedener Notationen und graphischer Darstellungen. In diesem Kapitel werden zweistellige Relationshiptypen vorausgesetzt, deren Grade in 1:N Notation angegeben werden (siehe [SS83]). Die Teilnahme von Entities an Relationships kann zwingend oder optional sein. Darüberhinaus können sich Relationships gegenseitig ausschließen. Zur graphischen Darstellung von Beziehungstypen mit ihren Graden werden wir im folgenden die SSADM-Notation verwenden (siehe z.B. [DCC92]).

Wird ein Relationshiptyp (ohne zusätzliche Axiome) so in SPECTRUM modelliert wie in Abschnitt 3 angegeben, so hat er den Grad m:n und ist für beide teilnehmenden Entitytypen optional. Von diesem allgemeinsten Fall abweichende Anforderungen an den Relationshiptyp müssen durch zusätzliche Axiome gefordert werden.

Zwingende Partizipation Die Teilnahme an einer Relationship `R` kann für jeden der beiden beteiligten Entitytypen zwingend sein, d.h. es kann keine Entity dieses Entitytyps im Informationssystem existieren, ohne an der Beziehung R teilzunehmen. Abbildung 1 zeigt einen Relationshiptyp `R`, der an seinem $\mathtt{E}_1$-Ende optional und an seinem $\mathtt{E}_2$-Ende zwingend ist.

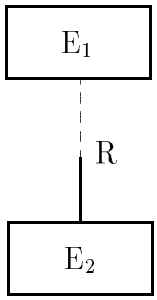


Abbildung 1: Zwingende Partizipation an einem Relationshiptyp

Die Tatsache, daß die Teilnahme des Entitytyps $\mathtt{E}_2$ an `R` zwingend ist, wird nun in SPECTRUM folgendermaßen ausgedrückt:

```
Ci : Db → Bool;
Ci strict total;
axioms ∀ db:Db in
Ci db = ∀e2:E2. e2 ∈ entE2 db ⇒ ∃e1:E1. R db (e1,e2);
endaxioms;
```

1:n-Beziehung Bei 1:n-Relationships heißt der Entitytyp an der 1-Seite der Beziehung (gemäß SSADM) auch *Master*-Entitytyp, der Entitytyp an der n-Seite *Detail*-Entitytyp. Die 1:n-Bedingung bedeutet, daß jeder Detail-Entity nur noch maximal eine Master-Entity zugeordnet sein kann. Abbildung 2 zeigt einen (an der Master-Seite optionalen) 1:n-Relationshiptyp `R` mit Master $\mathtt{E}_1$ und Detail $\mathtt{E}_2$.

Die 1:n-Bedingung für `R` wird in SPECTRUM wie folgt ausgedrückt:

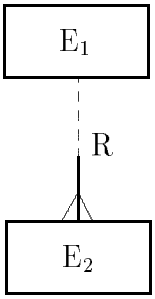


Abbildung 2: 1:n Relationshiptyp

```
Ci : Db → Bool;
Ci strict total;
axioms ∀ db:Db in
Ci db = ∀e1,e1':E1,e2:E2. R db (e1,e2) ∧ R db (e1',e2) ⇒ e1=e1';
endaxioms;
```

1:1-Beziehung Im Fall von 1:1-Relationshiptypen (Abbildung 3 zeigt einen auf beiden Seiten zwingenden 1:1-Relationshiptyp R) kann jede Entity des Typs E_1 mit höchstens einer (im Fall zwingender Partizipation genau einer) Entity des Typs E_2 in der Beziehung R stehen und umgekehrt.

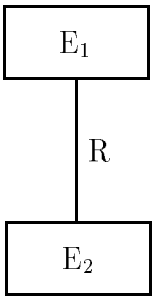
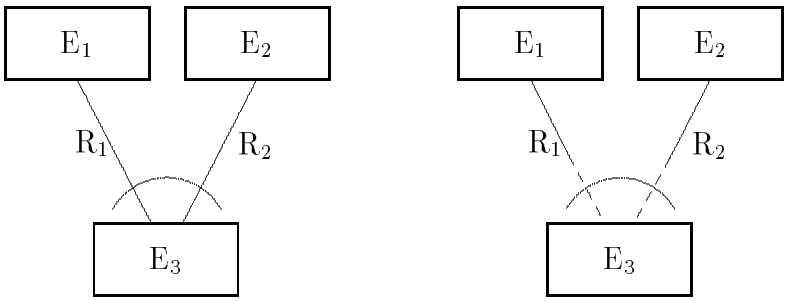


Abbildung 3: 1:1 Relationshiptyp

```
Ci : Db → Bool;
Ci strict total;
axioms ∀ db:Db in
Ci db = ∀e1,e1':E1,e2,e2':E2. (R db (e1,e2) ∧ R db (e1',e2) ⇒ e1=e1')∧
                              (R db (e1,e2) ∧ R db (e1,e2') ⇒ e2=e2');
endaxioms;
```

Sich ausschließende Relationshiptypen In SSADM besteht die Möglichkeit, die Beteiligung eines Entitytypen an mehreren Relationshiptypen als exklusiv zu kennzeichnen. In diesem Fall kann keine Entity dieses Typs an mehr als einer der betroffenen Relationships gleichzeitig teilnehmen. Dargestellt wird dies im E/R-Diagramm durch einen Kreisbogen, die betroffenen Relationshiptypen

verbindet. Es besteht die Einschränkung, daß exklusive Beteiligung nur zwischen Relationshiptypen mit zwingender Partizipation oder zwischen Relationshiptypen mit optionaler Partizipation gefordert werden kann. Mischformen sind nach SSADM nicht möglich. Die beiden erlaubten Varianten sind in Abbildung 4 a) und b) für den Fall zweier sich ausschliessender Relationshiptypen dargestellt.



a) b)

Abbildung 4: Gegenseitiger Ausschluß von Beziehungstypen

Für den Fall von m sich gegenseitig ausschliessenden Relationshiptypen $\mathtt{R}_1, \ldots, \mathtt{R}_m$ bedeutet das, daß jede Entity des Typs `E` an genau einem der Relationshiptypen $\mathtt{R}_1, \ldots, \mathtt{R}_m$ teilnehmen muß. In SPECTRUM ergibt sich dafür folgendes Prädikat[1]:

```
Ci : Db → Bool;
Ci strict total;
axioms ∀db in
Ci db = ∀e. (∃e1. R1 db (e,e1)) xor ... xor (∃em. Rm db (e,em));
endaxioms;
```

Sind die sich gegenseitig ausschliessenden Relationshiptypen optional, so besteht zusätzlich die Möglichkeit, daß Entities des Typs `E` an keinem der Relationshiptypen $\mathtt{R}_1, \ldots, \mathtt{R}_m$ teilnehmen, d.h. es ergibt sich folgendes Prädikat:

```
Ci : Db → Bool;
Ci strict total;
axioms ∀db in
Ci db = ∀e. (∀e1,...,em. ¬(R1 db (e,e1) )∧ ... ∧ ¬(Rm db (e,em))) ∨
            ((∃e1. R1 db (e,e1)) xor ... xor (∃em. Rm db (e,em)));
endaxioms;
```

Da die Optionalität der beteiligten Relationshiptypen hier bereits in die Spezifikation der Integritätsbedingung eingegangen ist, findet bei sich ausschliessenden Relationshiptypen die weiter oben unter dem Punkt 'Zwingende Partizipation' angegebene Regel keine Anwendung. Die anderen Übersetzungsregeln

[1] Das in dieser Spezifikation verwendete Infix-Symbol `xor` steht für das (hier nicht spezifizierte) logische 'exclusive oder'.

(Punkte '1:n-Beziehung' und '1:1-Beziehung') werden jedoch genau wie bei allen anderen Relationshiptypen angewendet.

Literatur

[DCC92] Ed Downs, Peter Clare, and Ian Coe. *Structured Systems Analysis and Design Method — Application and Context.* Prentice Hall, 1992.

[Huß93] H. Hußmann. Zur formalen Beschreibung der funktionalen Anforderungen an ein Informationssystem. Technical Report TUM-I9332, Technische Universität München, 1993.

[LCFW92] M. Löwe, F. Cornelius, J. Faulhaber, and R. Wessäly. Ein Fallbeispiel für KORSO: Das heterogene verteilte Managementsystem HDMS der Projektgruppe Medizin-Informatik (PMI) am Deutschen Herzzentrum Berlin und an der TU Berlin — Ein Vorschlag. Technical Report 92-45, TU Berlin, December 1992.

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

[SS83] G. Schlageter and W. Stucky. *Datenbanksysteme: Konzepte und Modelle.* Teubner, Stuttgart, 1983.