HDMS–A und OBSCURE in KORSO

Die Funktionale Essenz von HDMS–A aus Sicht

der algorithmischen Spezifikationsmethode

Teil 2: Schablonen zur Übersetzung eines E/R-Schemas in eine OBSCURE Spezifikation

Serge Autexier

Technischer Bericht **A/05/93**

Dezember 1993

Serge Autexier (Autor)

serge@dfki.uni–sb.de

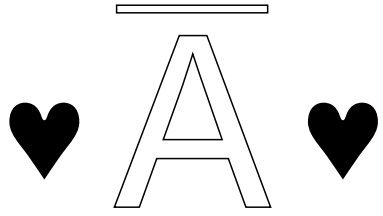
unter Mitarbeit von

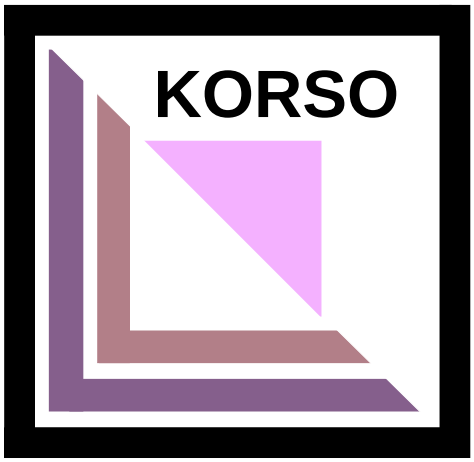
Christoph Benzmüller und Ramses A. Heckler

und unter Beratung durch

Stefan Conrad (Uni Braunschweig)
Rudi Hettler (TU München)

HDMS





Serge Autexier

serge@dfki.uni–sb.de

HDMS–A und OBSCURE in KORSO

Die Funktionale Essenz von HDMS–A aus Sicht der algorithmischen Spezifikationsmethode

Teil 2: Schablonen zur Übersetzung eines E/R-Schemas in eine OBSCURE Spezifikation

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführende Bemerkungen

1.1 Das allgemeine Vorwort

The following report is part of the central case study HDMS-A[1] within the german national project KORSO[2]. This study is dedicated to the development of a complex information system for the support of the patient data administration in the specialized heart disease clinic DHZB[3]. While the developpers group PMI[4] develops the real system for the clinic called HDMS, the project KORSO's aim was the rigorous development of an abstracted version of HDMS by exclusive use of pure formal methods. The abstraction refers both to number of modelled documents and depth of treatment, while still considering the relevant aspects in a partly parameterized way.
The investigation of HDMS-A has been done by 11 partners within KORSO. An extended overview provides [CHL94a, CHL94b]. The main topics are: an actual state analysis of the selected documents in the patient record as well as of the existing and motivating problems concerning safety and security, distribution and effectiveness (see [CKL93]); the requirements analysis and specification, beginning with a description of the chosen policy ([Hußß93]) and two technical formalisms providing means for the translation of entity-relationship diagrams as well as data flow diagrams into SPECTRUM (see [Het93, Nic93]), finally the requirements specification itself ([SNM+93]); the main field of safety and security treated and in general ([GH93]) and concretely: [Ren94, Ste93]; finally the investigation of the integration of existing software components into a formal development: [Con93, Dam93, Sch94, Shi94, MZ94]. Apart from those there were few specialized contributions to selected topics: [Hec93, Aut93, Ben93, Fuc94, SH94].

1.2 Über diesen Bericht

Dieser Bericht ist der zweite Teil des Abschlußberichts des Lehrstuhls von Prof. Loeckx zum KORSO Fallbeispiel HDMS-A. Die drei Teile wurden zusammen entwickelt und stellen eine Einheit dar. Diese Berichte sind Bestandteil einer umfangreichen Reihe von Abschlußberichten zur Fallstudie HDMS-A des KORSO–Projekts. Einen ausführlichen Überblick über diese Reihe gibt [CHL94a] (bzw. [CHL94b]). Unsere drei Berichte sind sowohl un-

[1] Heterogeneous Distributed Information Management System

[2] Korrekte (=correct) Software, sponsored by the german ministry for research and technology

[3] Deutsches Herzzentrum Berlin

[4] Projektgruppe Medizin Informatik am DHZB und der TU Berlin

tereinander als auch mit den anderen Veröffentlichungen der HDMS-A–Reihe inhaltlich eng verwoben. Dieser Bericht vermittelt also nur in Zusammenhang mit den beiden anderen Berichten [Hec93] und [Ben93] einen allgemeinen Überblick zu den Arbeiten am HDMS-A Fallbeispiel. Deshalb wird zu Beginn dieses Berichtes ein kurzer Überblick zu den drei Teilen gegeben:

- **Teil 1: [Hec93]**
  In [Hec93] wird das Fallbeispiel HDMS-A als Ganzes, sowie dessen Ziele vorgestellt. Speziell zu der am Lehrstuhl von Prof. Loeckx geleisteten Arbeit wird der modellierte Ausschnitt Herzkatheter-Untersuchung des Fallbeispiels HDMS-A, sowie die Methodik in der Vorgehensweise geschildert. Die Beschreibung der SPECTRUM-Spezifikation ist in [Het93, SNM+93] zu finden.

- **Teil 2: dieser Bericht**
  In diesem Bericht geht es darum, eine schematische Übersetzung eines konkreten Entity/Relationship-Schema in eine OBSCURE-Spezifikation zu beschreiben. Dazu werden sowohl reine OBSCURE-Spezifikationen, als auch sogenannte "Schablonen" angegeben. Eine genauere Beschreibung des Konzepts einer Schablone findet man im Abschnitt **1.3 Abstecken des Rahmens**, sowie in [Het93]. Zusätzlich wird die Struktur der Spezifikation geschildert. Durch die Darstellung der Struktur, sowie der Angabe von OBSCURE-Spezifikationen und Schablonen, wird eine schematische Übersetzung von Entity/Relationship-Schemata motiviert.

- **Teil 3: [Ben93]**
  Der Bericht [Ben93] konzentriert sich auf die Modellierung des Ausschnittes Herzkatheter-Untersuchung aus der atomaren Ebene von HDMS-A. Dabei baut er sowohl auf den in [Hec93] dargestellten methodischen Grundlagen, als auch auf der in diesem Bericht vorgestellten schematischen Vorgehensweise auf. Weiterhin werden die Unterschiede der OBSCURE-Spezifikation zur SPECTRUM-Spezifikation des zu spezifierenden Ausschnittes von HDMS-A geschildert. Die in der Spezifikation des Ausschnittes importierten Sorten und Operationen resultieren aus der Anwendung der in dem vorliegenden Bericht angegebenen schematischen Übersetzung auf das, dem Ausschnitt zugrundeliegenden Entity/Relationship-Schema.

1.3 Abstecken des Rahmens

In diesem Bericht werden Schablonen angegeben, um ein konkretes Entity-Relationship Schema (E/R-Schema) mit der Spezifikationssprache OBS-CURE beschreiben zu können. Wir orientieren uns dabei an der Spezifikation von Entity/Relationship Modellen in SPECTRUM, wie in [Het93] beschrieben. Dabei übernehmen wir auch die dort beschriebenen Operationen. Zur Beschreibung der Übersetzung verwenden wir sowohl Schablonen, , als auch parametrisierte Spezifikationen.
Schablonen geben einerseits an, wie der Spezifizierer einen Modul (beziehungsweise Teil-Spezifikation) schreiben soll. Andererseits geben sie auch an, wie er die einzelnen Teile der Spezifikation zusammensetzen muß und sollte. Er "sollte", da in den Schablonen eine Strukturierung der Gesamtspezifikation vorgenommen wurde, die unsere intuitive Vorstellung eines E/R-Schemas widerspiegeln.

1.4 Begriffsbildung

Wir gebrauchen die Begriffe "Entity" und "Relationship" in Anlehnung an die Beschreibung der SPECTRUM Spezifikation eines E/R-Schemas (siehe hierzu auch [Het93] und [Hec93]). Diese entsprechen auch den in der Fachliteratur (Siehe [MP88]) über E/R-Modelle gebräuchlichen Begriffen. Um Mißverständnissen vorzubeugen geben wir dennoch an, was wir unter den einzelnen Begriffen verstehen. Das zugrundeliegende E/R-Modell enthält einerseits Entities, die aus sogenannten Attributen bestehen und andererseits binäre Relationships über den Schlüsseln der Entities. Die formale Definition baut auf bestimmten Voraussetzungen auf, die im folgenden aufgeführt werden. Zur Verdeutlichung geben wir ein paar Beispiele an, die sich auf die in Abbildung 1 auf Seite 4 gezeigte Entity eines E/R-Schemas beziehen.

1. Attributtypen sind als gegeben vorausgesetzt. Sie sind in der Menge **Attributtypes** zusammengefaßt. Ein Attributtyptyp aus der Beispiel Entity ist zum Beispiel **Datum**.

2. Attributnamen sind als gegeben vorausgesetzt. Sie sind in der Menge **Attributnames** zusammengefaßt. Dies ist zum Beispiel der Name **GebDatum**,

3. eine Menge von Entitynamen **Entitynames**, zum Beispiel **Patient**.

4. eine Menge von Relationshipnamen **Relationshipnames**.

5. für jeden Attributtyp aus der obigen Menge einen Domain. Für den Attributtyp **Datum** sind dies alle **Daten**.

Entity PATIENT

Attributnamen	Attributtypen	
PatId	PatId	(mandatory, primary key)
Name	Name	(mandatory)
Geschlecht	Geschlecht	(mandatory)
GebDatum	Datum	(mandatory)
GebOrt	Ort	(mandatory)
Kostentraeger	Kostentraeger	(mandatory)
Adresse	Adresse	
Hausarzt	Hausarzt	
Koerperdaten	Koerperdaten	
Station	Station	
Zimmer	Zimmer	

Abbildung1: Die Beispiel Entity PATIENT

1.4.1 Attribute

- Attributtypen sind als gegeben vorausgesetzt. Sie sind unter **Attributtypes** zusammengefaßt (Siehe oben).
- **Attributnamen** sind auch als gegeben vorausgesetzt. Sie sind unter **Attributnames** zusammengefaßt (Siehe oben).
- Um den "**Domain** eines Attributtyps" zu erhalten, nehmen wir an, daß es eine Funktion D_A gibt, die zu jedem Attributtyp seinen entsprechenden Domain liefert. Diese Funktion erhält den Namen D_A und hat folgende Funktionalität:

$$D_A(Attributtyp) = Domain\_Of\_Attributtyp$$

- Ein "**Attribut**" besteht aus einem **Attributnamen** und einem Element (Wert) aus dem Domain des Attributtyps des Attributs. Ein Attribut ist also ein Element aus dem Kreuzprodukt

$$Attribut \in Attributnames \times \bigcup_i D(Attributtyp_i)$$

- Unter einem "**benannten Attributtyp**" verstehen wir im Folgenden ein Paar, welches aus einem Attributnamen und einem Attributtyp besteht. Ein benannter Attributtyp ist also ein Element aus folgendem Kreuzprodukt

$$Attributnames \times Attributtypes$$

Zum Beispiel ist das Paar **(GebDatum, Datum)** aus der Entity PATIENT in Abbildung 1 auf Seite 4 ein sogenannter **benannter Attributtyp**.

1.4.2 Entities

- Entitynamen sind als gegeben vorausgesetzt. Sie sind in der Menge **Entitynames** zusammengefaßt.

- Ein **Entitytyp** für Entities, die aus n Attributen bestehen, wird folgendermaßen aufbauend auf den entsprechenden **benannten Attributtypen** definiert:

$$Entitytyp = (AName_1, ATyp_1) \times \ldots \times (AName_n, ATyp_n)$$

wobei

  - $AName_i \in AttributNames$ *und*

  - $ATyp_i \in Attributtypes$ *und*

  - $\forall 1 \leq i, j \leq n : i \neq j \Rightarrow AName_i \neq AName_j\}$

- Der **Domain** eines Entitytyps definiert sich auch über den Domains, der in der Definition des Entitytyps enthaltenen Attributtypen.

$$\begin{aligned} D_E(Entitytyp) = \{f : \{AName_1, \ldots, AName_n\} \\ \rightarrow \textstyle\bigcup_{i=1\ldots n} D_A(ATyp_i) \\ |f(AName_i) \in D_A(ATyp_i)\ \forall 1 \leq i \leq n\} \end{aligned}$$

wobei Entitytyp wie oben definiert.

- Eine **Entity** besteht aus einem Namen aus der Menge **Entitynames** und einem Element (Wert) aus dem Domain des Entitytyps. Somit ist eine Entity ein Element aus dem Kreuzprodukt

$$Entitynames \times \textstyle\bigcup_i D_E(Entitytyp_i)$$

- Ein **benannter Entitytyp** ist, analog zu einem benannten Attributtyp, ein Paar bestehend aus einem Entitynamen und einem Entitytyp, und somit ein Element aus dem Kreuzprodukt

$$Entitynames \times Entitytypes$$

1.4.3 Schlüsselattribute

Zur Definition eines Schlüsselattributs, eines konkreten Schlüssels und eines abstrakten Schlüssels verweisen wir auf [Hec93]. Eine formale Definition analog zu Attributen und Entities würde an dieser Stelle den Leser nur verwirren. Dennoch ist sie möglich und der interessierte Leser findet sie im Anhang A dieses Berichts. Es sei aber explizit darauf hingewiesen, daß er zunächst die informale, aber präzise Definition aus [Hec93] verstanden haben sollte, bevor er die formale Definition liest.

1.4.4 Relationships

- Die Relationshipnamen sind als gegeben vorausgesetzt. Sie sind in der Menge **Relationshipnames** zusammengefaßt.

- Ein **Relationshiptyp** ist definiert als ein Paar bestehend aus den beiden benannten Entitytypen, die ihm zugrunde liegen. Im späteren Verlauf der Spezifikation werden wir zwar Relationen zwischen zwei Entities über deren Schlüssel definieren, was aber an der Semantik nichts ändert.

$$Relationshiptyp = (EName_1, ETyp_1) \times (EName_2, ETyp_2)$$

  wobei $ENname_i \in Entitynames$ und ETyp_1, ETyp_2 wie oben definierte Entitytypen sind. Die Klasse aller Relationshiptypen bezeichnen wir mit **Relationshiptypes**.

- Der **Domain** eines Relationshiptyps definiert sich, analog zur Definition des Domains eines Entitytyps, über den Domains, der in seiner Definition enthaltenen Entitytypen. Die dazu eingeführte Funktion D_R hat folgende Semantik:

$$D_R(RTyp) = \mathcal{P}(D_E(ETyp_1) \times D_E(ETyp_2))$$

wobei

1. $\mathcal{P}$ der Potenzmengen-Operator ist.
2. $RTyp = ((ENname_1, ETyp_1) \times (ENname_2, ETyp_2))$
3. $ENname_i \in Entitynames$ und ETyp_1, ETyp_2 wie oben definierte Entitytypen.

- Eine **Relationship** definieren wir als Paar, bestehend aus einem Relationshipnamen und einem Element (Wert) aus dem Domain des Relationshiptyps.

$$relationship \in Relationnames \times \bigcup_{Relationshiptyp} D_R(Relationshiptyp)$$

- Ein **benannter Relationshiptyp** wird definiert als ein Paar, bestehend aus einem Relationshipnamen und einem Relationshiptyp. Ein benannter Relationshiptyp ist also ein Element aus dem Kreuzprodukt

$$Relationshipnames \times Relationshiptypes$$

1.4.5 Datenbank

- **Datenbanknamen**: wir führen hier keine Namen für Datenbanken ein, da wir im Folgenden immer nur von einer einzigen sprechen werden.

- Ein **Datenbanktyp** definiert sich durch die benannten Entitytypen und benannten Relationshiptypen, die in einer Datenbank dieses Typs enthalten sein sollen.

$$\begin{aligned}\text{datenbanktyp} = (&(Ename_1, Etyp_1), \ldots (ENname_n, Etyp_n),\\ &(RName_1, Rtyp_1), \ldots, (RName_m, RTyp_m))\end{aligned}$$

wobei

1. die Namen $ENname_i$ der Entitytypen $ETyp_i$ zusammen mit den Namen $RName_j$ der Relationshiptypen $RTyp_j$ paarweise disjunkt sein müssen.
2. jeder auftretende Relationshiptyp $RTyp_i$ sich aus zwei benannten Entitytypen $(ENname_j, ETyp_j)$ und $(ENname_k, ETyp_k)$ aus der Liste zusammensetzen muß.

- Der **Domain** eines Datenbanktyps definiert sich aufbauend auf den benannten Entity- und Relationshiptypen, die in seiner Definition vorkommen, ist aber etwas komplizierter, als die der Domains von Entitytypen und Relationshiptypen. Die Schwierigkeit ist die, daß im allgemeinen mehrere Entities eines benannten Entitytyps in einer Datenbank enthalten sind. Die Semantik der dafür einzuführenden Funktion D_D sieht folgendermaßen aus:

$$D_D(DTyp) = \{f : \{EName_1, \ldots, EName_n, RName_1, \ldots, RName_m\} \rightarrow ((\bigcup_{ETyp_i} \mathcal{P}(D_E(ETyp_i))) \bigcup (\bigcup_{RTyp_j} D_R(RTyp_j))) \}$$

  wobei

  1. DTyp ein Datenbanktyp ist,
  2. EName$_i$ und ETyp$_i$ respektive, die Namen und Typen der im Datenbanktyp DTyp enthaltenen benannten Entitytypen
  3. RName$_j$ und RTyp$_j$ respektive, die Namen und Typen der im Datenbanktyp DTyp enthaltenen benannten Relationshiptypen.

  Diese Funktion D_D liefert zu jedem Datenbanktyp eine Klasse von Funktionen, welche die folgende Eigenschaft haben:

  - Zu jedem Entitynamen liefert sie eine Ansammlung von Entities aus dem Domain des entsprechenden Entitytyps.
  - Zu jedem Relationshipnamen liefert sie eine Relationship aus dem Domain des entsprechenden Relationshiptyps.

1.5 Die Ebenen der Spezifikation

Wir geben die Struktur unserer Spezifikation in Ebenen an, die mittels des OBSCURE Konstruktors X_COMPOSE zusammengesetzt werden. Eine Ebene ist entweder in weitere Ebenen unterteilbar oder setzt sich aus voneinander unabhängigen Spezifikationen zusammen, die mit Hilfe des OBSCURE Konstruktors PLUS verbunden werden.

Die Spezifikation eines E/R-Schemas unterteilt sich grob in 3 Ebenen (vergleiche hierzu Abbildung 2 auf Seite 10):

1. **Die Attributebene:** Sie ist die unterste Ebene und soll, intuitiv gesehen, alles liefern, was zur eigentlichen Spezifikation des E/R-Schemas benötigt wird. Beispiele: Attribut-Sorten, Schlüssel-Sorten, elementare Schlüsselgenerierungs-Funktionen.

2. **Die Datenbankebene:** Sie bildet die mittlere Ebene der Spezifikation. In ihr werden die eigentlichen Bestandteile des E/R-Schemas spezifiziert. Dies sind hauptsächlich die Spezifikationen

   - der Entities des E/R-Schemas,
   - der einzelnen Entity-Ansammlungen des E/R-Schemas,
   - der Ansammlung aller Entity-Ansammlungen des E/R-Schemas,
   - der Relationships des E/R-Schemas,
   - der Ansammlung aller Relationships des E/R-Schemas und
   - der gesamten Datenbank des E/R-Schemas.

   Wir verwenden bewußt den Begriff "Ansammlung" und nicht Mengen, da wir im folgenden keine Mengen verwenden werden, sondern sogenannte Monolisten. Näheres zu dieser Thematik findet man in [Hec93].

3. **Die Komfort- und Schnittstellenebene:** Die Komfort- und Schnittstellenebene hat, wie ihr Name schon andeutet, zwei Funktionen: In ihr sollen Operationen spezifiziert werden, die in SPECTRUM nicht vorhanden sind, da sie dort nicht gebraucht werden. Typischerweise handelt es sich hierbei um Operationen, die zur Simulation des Existenzquantors von SPECTRUM verwendet werden. Zum Beispiel ist die Operation **is_in***Entityname* : *Entityname DB* -> **bool** eine solche Operation. Diese überprüft, ob eine bestimmte Entity in der Datenbank enthalten ist oder nicht.
   Auch soll diese Ebene eine Schnittstelle zur nächsten Ebene, der Ablaufebene, sein. Sie soll alle diejenigen Operationen und Sorten exportieren, die in der Ablaufebene benötigt werden und nur diese. Ein Grund ist zum Beispiel, daß es im allgemeinen nicht sinnvoll ist, in der Ablaufebene mittels low-level Operationen auf der Datenbank zu arbeiten: In der Ablaufebene soll nur mit den in Modul DB zur Verfügung gestellten Operationen auf die Datenbank lesend oder schreibend zugegriffen werden können. Dieser Aspekt ist vor allem wichtig im Hinblick

auf eine Verifikation der Spezifikation, insbesondere der Überprüfung der statischen Integritätsbedingungen der Gesamtspezifikation (Datenmodellebene und Ablaufebene). Das Vergessen der Sorten und Operationen kann zum Teil schematisch geschehen. Aus der schematischen Beschreibung mittels Schablonen, is schon ersichtlich, welche in jedem Fall vergessen werden können. Jedoch sind die in der Spezifikation eines konkreten E/R-Schemas zu vergessenden Operationen auch abhängig von der Spezifikation der Abläufe, bzw. von den dort benötigten Operationen. Diese müssen in der Spezifikation eines konkreten E/R-Schemas vom Spezifizierer angegeben werden.

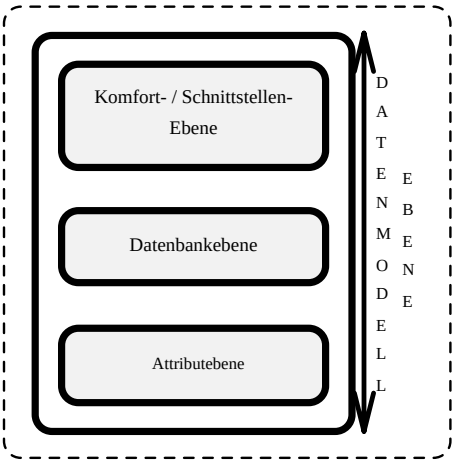


Abbildung2: Struktur des Datenmodells

1.5.1 Die Attributebene

Die Spezifikation der Attributebene unterteilt sich in 4 Ebenen (vergleiche hierzu Abbildung 3 auf Seite 11):

1. **Die Wertebereiche der Attribute:** Sie bilden die unterste Ebene der Attributebene. In ihr werden die Sorten spezifiziert, die zur Spezifikation der Domains der einzelnen Attribute der Entities verwendet werden.

2. **Die Umbenennungsebene:** Sie bildet die zweite Ebene der Attributebene. In ihr werden die Sorten spezifiziert, die die Domains der

Attributtypen darstellen. Dabei handelt es sich vielfach nur um eine Umbenennung der alten Sorten. Diese Ebene ist dennoch notwendig, da zum Beispiel zwei Sorten *Name* und *Ort* beide auf der Grundlage einer Sorte *String* aufbauen können.

3. **Die UNDEF-Einführungsebene:** Sie bildet die dritte Ebene der Attributebene. In dieser Ebene wird für alle Sorten ein neuer Träger eingeführt. Dazu werden alle Sorten S aus der vorherigen Ebene in eine neue Sorte *AttrS* eingebettet und als zusätzlichen Träger der Sorte *AttrS* der Träger *UNDEF* spezifiziert. Näheres hierzu findet man in [Hec93].

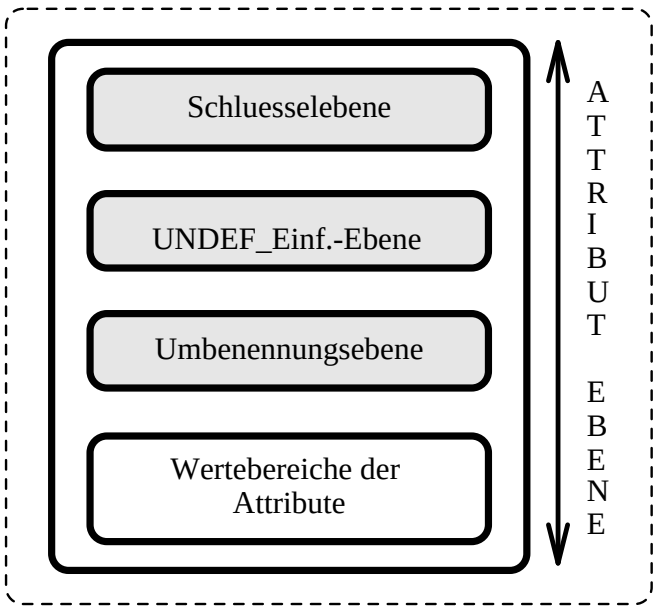


Abbildung3: Die Attributebene

4. **Die Schlüsselebene** (vergleiche hierzu Abbildung 4 auf Seite 12)
Sie bildet die oberste Ebene der Attributebene. Es werden in ihr sogenannte "next"-Operationen spezifiziert, die aus einer Menge von Schlüsselattributen ein neues Schlüsselattribut berechnen. Diese Art von Funktionen muß "nach außen" zur Verfügung gestellt werden, damit die Generierung neuer konkreter Schlüssel außerhalb der Datenmodellebene realisiert werden kann. Zu beachten ist, daß diese "next"-Operationen sehr spezifisch sein können: zum Beispiel kann prinzipiell nicht ausgeschlossen werden, daß eine "next"-Operation eines Schlüsselattributs, das in zwei benannten Entitytypen als solches

enthalten ist, unterschiedliche Semantiken haben muß. Deshalb ist eine "next"-Operation nicht nur spezifisch bezüglich eines Schlüsselattributs, sondern auch bezüglich des benannten Entitytyps, für die sie verwendet werden soll. Es sei noch angemerkt, daß die Entscheidung, "next"-Operationen einzuführen, schon eine Design-Entscheidung ist. Auch hierzu findet man eine ausführlichere Beschreibung in [Hec93].
Zudem werden die Sorten der konkreten und abstrakten Schlüssel der Entities des E/R-Schemas spezifiziert.
Diese Ebene unterteilt sich nicht nur in verschiedene Ebenen, sie ist vielmehr in zwei voneinander unabhängige Teil-Spezifikationen unterteilt. Dies resultiert daraus, daß sowohl die Spezifikationen der "next"-Operationen, der konkreten Schlüssel und der abstrakten Schlüssel nur die Sorten der Domains aus der vorangegangenen **UNDEF-Einführungsebene** brauchen, die in mindestens einem benannten Entitytyp zur Spezifikation eines konkreten Schlüssels verwendet werden. Alle anderen Sorten werden in dieser Ebene nur "durchgereicht".

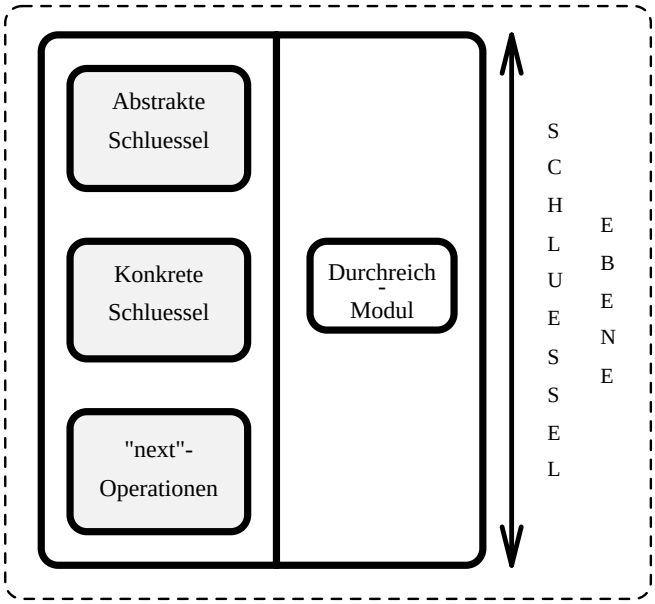


Abbildung4: Die Schlüsselebene

1.5.2 Die Datenbankebene

Die Datenbankebene ist in unserem Ansatz nicht unterteilt in verschiedene Ebenen. Sie ist vielmehr eine Art Graph mit sechs Knoten, einer Quelle und

einer Senke. (Siehe Abbildung 5 auf Seite 13)
Die Quelle wird durch die Spezifikationen der Entities des E/R-Schemas gebildet und die Senke durch die Spezifikation der gesamten Datenbank. Dazwischen liegen noch vier weitere Knoten:

1. **Entity-Ansammlungen:** In diesem Teil der Spezifikation werden für alle Entities Monolisten [5] dieser Entities spezifiziert.

2. **Ansammlung aller Entity-Ansammlungen:** Hier werden alle Entity-Ansammlungen zu einer großen Ansammlung zusammengefaßt, die gerade soviele Bestandteile hat, wie es benannte Entitytypen in dem spezifischen E/R-Schema gibt. Diese große Ansammlung spezifizieren wir aber nicht als Monoliste[6], sondern als ein n-Tupel, dessen Komponenten gerade die Entity-Ansammlungen sind, wobei n die Anzahl der im E/R-Schema vorkommenden benannten Entitytypen ist.

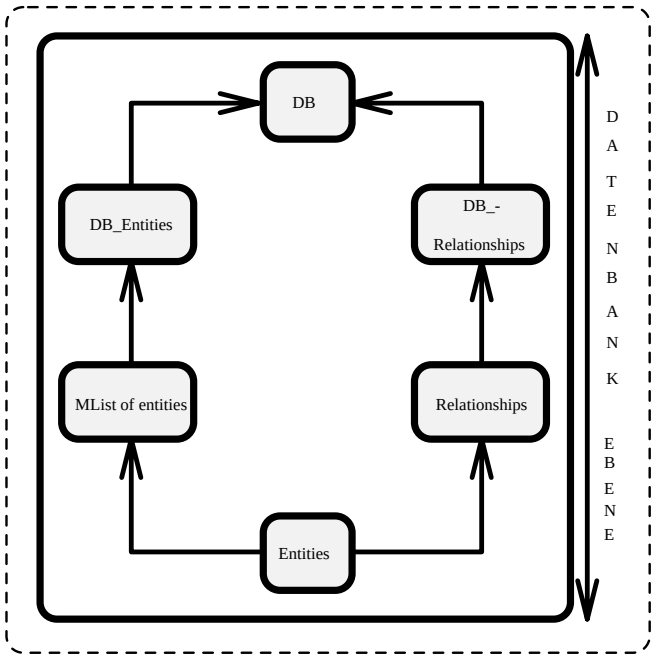


Abbildung5: Die Datenbankebene

[5]Monolisten sind Listen, in denen es keine doppelten Vorkommen gibt. Eine genauere Beschreibung, sowie deren Signatur findet man in [Hec93]
[6]Siehe Fußnote 5, Seite 13

3. **Binäre Relationships über abstrakten Schlüsseln:** Diese Teil-Spezifikation dient dazu, die Relationships über den Schlüsseln zu spezifizieren, mit Hilfe derer Zugehörigkeiten zweier Entities zueinander ausgedrückt werden können. Zusätzlich zur eigentlichen Spezifikation der Relationships, werden sogenannte "check"-Funktionen angegeben, die für die jeweilige Relationship überprüfen, ob es eine 1:1, 1:N, N:1 oder N:N Relationship ist. Die 'check"-Funktionen zur zwingenden bzw. ausschließenden Partizipation einer Entity, können auf dieser Ebene noch nicht spezifiziert werden: Deren Spezifikationen wird in der nächsten großen Ebene der Datenmodell-Spezifikation, der Komfort- und Schnittstellenebene, spezifiert. Diese "check"-Funktionen haben den Zweck, die statischen Integritätsbedingungen zu formulieren.

4. **Ansammlung aller Relationships:** Die oben spezifizierten Relationships werden zusammengefaßt zu einer Ansammlung. Deren Spezifikation erfolgt in diesem Abschnitt der Spezifikation. Wie bei der Spezifikation der Ansammlung aller Entity-Ansammlungen, spezifizieren wir diese nicht als Monoliste[7], sondern als m-Tupel über den Relationshiptypen, wobei m die Anzahl der im E/R-Schema vorkommenden benannten Relationshiptypen ist.

1.5.3 Struktur der Komfort- und Schnittstellenebene

Für diese Ebene gibt es keine besondere Struktur. Sie besteht aus einem großen Modul, in dem die entsprechenden Operationen spezifiziert werden, und an dessen Ende stehen das schematische und das spezifische FORGET zum Vergessen der Operationen.

[7] Siehe Fußnote 5, Seite 13

Teil I
Die Attributebene

2 Attribute

2.1 Vorbemerkungen:

In diesem Abschnitt beschreiben wir zunächst, wie wir Attribute spezifizieren.
Attribute bestehen aus einem Attributnamen und einem Attributtyp. Um den Attributnamen kümmern wir uns erst bei der Spezifikation von Entities. In diesem Abschnitt interessiert uns nur die Spezifikation des Domains eines Attributtyps.
Wir unterteilen die Spezifikation eines Wertebereichs in zwei Teilspezifikationen:

1. Die Domains der Attributtypen werden zum Teil mit Hilfe anderer Sorten spezifiziert. Zum Beispiel könnte man eine Sorte *Nat* zur Spezifikation des Domains eines Attributtyps *Alter* verwenden. Dies würde durch Umbenennung der Sorte *Nat* in die Sorte/Attributtyp *Alter* erfolgen. Diesen Umbenennungen ist der erste Teil der Spezifikation gedacht. Im allgemeinen wird dort eine Sorte S in eine neue Sorte *Attributtyp* umbenannt.

2. Im zweiten Teil spezifizieren wir, wie ein Attributtyp *ATyp* um einen sogenannten UNDEF Träger erweitert wird. Dabei wird die Sorte *Atyp* in eine neu kreierte Sorte *AttrAtyp* eingebettet, und diese erhält eine zusätzliche Konstante UNDEF_Atyp.

Dabei gehen wir davon aus, daß die oben erwähnte Sorte S spezifiziert ist (siehe Abschnitt 1.5.1 **Die Attributebene**).

2.2 Die Umbenennungsebene

2.2.1 Vorgehen

Wie einführend schon erwähnt, wollen wir in der Struktur der Spezifikation ausdrücken, daß die Sorte S zur Spezifikation von Domains von Attributtypen unterschiedlicher, benannter Attributtypen verwendet werden kann. Der Modul, in dem die Sorte S spezifiziert ist, wird in dem Modul, in dem

der Domain eines Attributtyps spezifiziert werden soll, eingebunden und die Sorte *S* mittels eines E_RENAME in den Attributtyp umbenannt.

2.2.2 Die Spezifikation des Attributtyps

Der Modul, in dem die Sorte *S* spezifiziert wurde, heiße *S* und der Attributtyp sei *ATyp*. Dann sieht die Spezifikation der Sorte *ATyp* folgendermaßen aus:

Der Modul *ATyp*
(INCLUDE *S*__X
E_RENAME SORTS *S* AS SORTS *ATyp*)

Im Anschluß an diese Spezifikation (zum Beispiel in einem EXTENSION-Teil oder durch hinzufügen von QUOTIENT oder SUBSET Konstruktoren), können noch zusätzliche Eigenschaften der neuen Sorte spezifiziert werden. Zum Beispiel könnte die Sorte *Alter* auf die Träger der Sorte *Nat* eingeschränkt werden, die die Zahlen von 0 bis 150 repräsentieren. Für das oben erwähnte Beispiel der Sorte *Nat*, die zur Spezifikation des Attributtyps *Alter* verwendet werden könnte, sähe die Spezifikation folgendermaßen aus:
Der Modul ALTER
(INCLUDE NAT E_RENAME SORTS Nat AS SORTS Alter)
Das Vorgehen kann man sich veranschaulichen anhand des in Abbildung 6 auf Seite 16 aufgezeigten Beispiels.

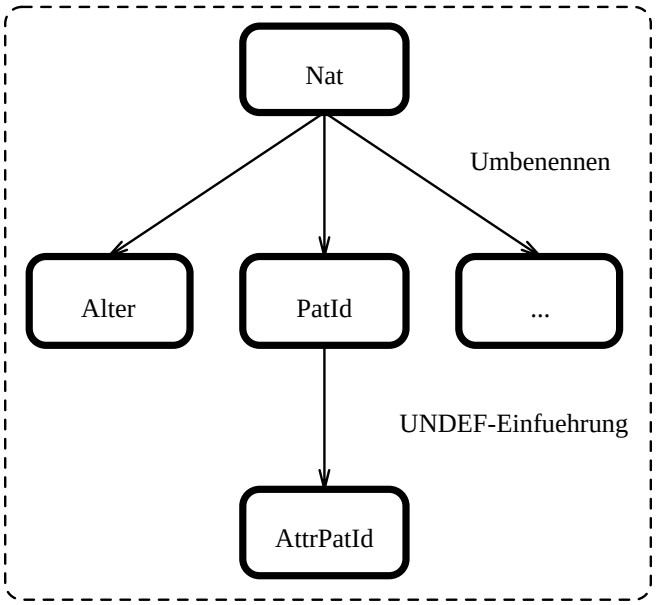


Abbildung6: Vorgehen

2.2.3 Der Modul UMBENENNUNGSEBENE

Die Gesamtspezifikation der Umbenennungsebene geschieht im Modul UMBENENNUNGSEBENE. Dieser setzt sich aus den einzelnen kleinen Moduln (wie zum Beispiel der oben erwähnte Modul ALTER). Die Voraussetzungen für die Schablone des Moduls UMBENENNUNGSEBENE baut sich folgendermaßen auf:

Der Modul UMBENENNUNGSEBENE
(INCLUDE $Attributtyp_1$)
PLUS
$\vdots$
PLUS
(INCLUDE $Attributtyp_n$)

wobei die Namen der Moduln sind, in denen die Domains der benannten Attributtypen des zu spezifizierenden E/R-Schema spezifiziert werden.

2.3 Die UNDEF-Einführungsebene

Sei gegeben ein Attributtyp *ATyp*, der in dem Modul *ATYP* spezifiziert wurde. Die Einführung eines UNDEF Trägers geschieht folgendermaßen:

1. Es wird eine Sorte Attr*Atyp* spezifiziert, welche im gewissen Sinne eine Obersorte zur Sorte *ATyp* darstellen soll. Die Zuordnung eines Trägers der Sorte *ATyp* zu einem Träger Attr*ATyp* geschieht mittels eines einstelligen Konstruktors

   attr : *ATyp* -> Attr*ATyp*;

2. zusätzlich wird eine Konstante

   UNDEF_*ATyp* : -> Attr*ATyp*;

   definiert, welche den einzuführenden undefinierten Träger darstellen soll.

3. eine Selektoroperation

   rtta : Attr*ATyp* -> *ATyp*,

welche aus einem Träger der Sorte Attr*ATyp* den entsprechenden Träger der Sorte *ATyp* zurückliefert; wird die Operation auf UNDEF_*ATyp* angewendet, liefert sie den entsprechenden Error-Träger der Sorte *ATyp*.

Die Spezifikation der Obersorte Attr*ATyp* soll einheitlich geschehen. Aus diesem Grund wird dafür eine parametrisierte Spezifikation angegeben: der Importparameter ist

- die Sorte *ATyp*

und die Exportparameter sind

- die Sorte Attr*ATyp* und
- der Träger UNDEF_*ATyp*

2.3.1 Die Spezifikation

```
## Der Modul MK_ATTR_SORT
(
  IMPORTS
   SORTS
    ATyp
   OPNS
    _ = _ : ATyp ATyp -> bool    CREATE
   SORTS
    AttrATyp
   OPNS
    attr : ATyp -> AttrATyp
    rtta : AttrATyp -> ATyp
    UNDEF_ATyp : -> AttrATyp
  SEMANTICS
   CONSTR
    attr : ATyp -> AttrATyp
    UNDEF_ATyp : -> AttrATyp
   VARS
    var_ATyp : ATyp
    var_AttrATyp : AttrATyp
```

```
    PROGRAMS
     rtta ( var_Attr ATyp )
     <- CASE var_Attr ATyp OF
            attr ( var_ATyp ) : var_ATyp
            ELSE Error( ATyp )
        ESAC;
  ENDCREATE
)
PARAMS
  (SORTS ATyp)
  [SORTS AttrATyp
   OPNS UNDEF_ATyp]
```

2.3.2 Beispiel

Die Sorte *Alter*, spezifiziert in dem Moduln ALTER[8], soll um einen zusätzlichen Träger erweitert werden. Der Modul, in dem die Sorte *AttrAlter* spezifiziert wird, heißt ATTR_ALTER:

```
## Der Modul ATTR_ALTER
(INCLUDE MK_ATTR_SORT
  (SORTS Alter)
  [SORTS AttrAlter, OPNS UNDEF_Alter])
```

2.3.3 Verwendung der parametrisierten Spezifikation

An die Verwendung dieser parametrisierten Spezifikation werden noch einige Anforderungen gestellt:

1. Der Name der exportierten Sorte *AttrATyp* soll sich auf die Sorte *ATyp* beziehen.

2. Gleiches gilt für den Namen der exportierten Konstante UNDEF_*ATyp*.

Der Spezifizierer muß sich dessen bewußt sein, daß der hier verwendete Name *ATyp* als Meta-Variable für die Beschreibung dient. Im konkreten Fall muß er hierfür den Namen der Sorte verwenden, welche erweitert werden soll.

[8] Dieser Modul stammt aus der Wertebereichebene

2.3.4 Der Modul UNDEF_EBENE

Die Spezifikation der UNDEF_Ebene geschieht im Modul UNDEF_EBENE. Dieser setzt sich aus den einzelnen "ATTR"-Moduln zusammen (wie zum Beispiel der oben aufgezeigte Modul ATTR_ALTER). Die Schablone für den Modul UNDEF_EBENE sieht folgendermaßen aus:

Der Modul UNDEF_EBENE
(INCLUDE $AttrAttributtyp_1$)
PLUS
⋮
PLUS
(INCLUDE $AttrAttributtyp_n$)

wobei die Namen der Moduln sind, in denen die Domains der benannten Attributtypen des zu spezifizierenden E/R-Schema spezifiziert werden.

3 Die Schlüssel

3.1 "next"-Operationen

3.1.1 Vorbemerkungen

Für die Sorten, die zur Spezifikation von Schlüsselattributen verwendet werden, müssen sogenannte "next"-Operationen zur Verfügung gestellt werden, damit es möglich ist "von Außen" neue Schlüssel zu generieren. Unter "von Außen" verstehen wir hier die Spezifikation, die die im Datenmodell spezifizierten Sorten und Operationen verwendet. Dies ist zum Beispiel eine Spezifikation einer atomaren Ebene, wie sie [Ben93] spezifiziert. Da ein Schlüssel sich nur aus ein oder mehreren Schlüsselattributen zusammensetzt, können neue Schlüssel mit Hilfe von neuen Schlüsselattributen erzeugt werden. Zudem, wie in der Einleitung schon erläutert, ist eine "next"-Operation nicht nur spezifisch bezüglich des Schlüsselattributs, sondern auch der Entity, in der das Schlüsselattribut als solches enthalten ist.

3.1.2 Die Schablone

Die Semantik der "next"-Operationen ist von dem jeweiligen Spezifizierer anzugeben. Dennoch haben alle Spezifikationen von "next"-Operationen vier Gemeinsamkeiten:

- Der Name der zu spezifizierenden "next"-Operation wird abhängig gemacht von dem Namen der Entity, für die sie bestimmt ist. Zum Beispiel: Die "next"-Operation des Schlüsselattributs *PatId* in der Entity PATIENT erhält den Namen "**nextPATIENT**". Es ist nicht notwendig den Namen des Schlüsselattributs mit in die Namensgebung zu integrieren, da diese Unterscheidung mittels der Sorten des Argument- und Zielbereichs schon erkennbar ist.

- Sie importieren alle einen Schlüsselattributtyp und die Operationen darüber, die es ermöglichen, die "next"-Operation anzugeben[9].

- Jede "next"-Operation hat die Stelligkeit
  next*Entityname* : *MListOfAttributtyp* -> *Attributtyp*

- Wegen der Stelligkeit der "next"-Operation, muß in der Spezifikation der "next"-Operation, zuerst die Sorte *MListOfSchlüsselattribut* spezifiziert werden.

Zudem kann die Namensgebung der Spezifikationen vereinheitlicht werden: Ein Modul, in dem die "next"-Operationen für den durch *AttrKey* spezifizierten Domain eines Schlüsselattributs der Entities $E_1, \ldots, E_N$ spezifiziert wird, erhält den Namen MK_NEXT_*AttrKey*. Die der Sorte *AttrKey* zugrundeliegenden Sorte habe den Namen *RttaAttrKey*.

```
## Der Modul MK_NEXT_AttrKey
(
(INCLUDE MK_MONO_LIST
      (SORTS AttrKey)
      [SORTS MListOfAttrKey])
EXTENDED BY
IMPORTS
 SORTS RttaAttrKey
 OPNS rtta : AttrKey -> RttaAttrKey
    – Die Operation rtta stammt aus der Spezifikation MK_ATTR_SORT.
    –
    – ... zusätzlich noch die Operationen, die über RttaAttrKey arbeiten –
```

[9] Durch diese Forderung werden schon an die Spezifikation der Sorte S, die dem Schlüsselattribut zugrundeliegt, Anforderung gestellt. So muß es zum Beispiel zu gegebenen Trägern $t_1, \ldots, t_n$ der Sorte S, einen Träger t geben, so daß gilt:
$\forall 1 \leq i \leq n : t \neq t_i$

CREATE
 OPNS
  ## nextE_1 : *MListOfAttrKey* -> *AttrKey* ist die "next"-Operation für
  ## das Schlüsselattribut des Typs *AttrKey* aus der Entity mit
  ## Namen E_1.
  nextE_1 : *MListOfAttrKey* -> *AttrKey*
  ⋮
  nextE_N : *MListOfAttrKey* -> *AttrKey*
 SEMANTICS
  ⋮
ENDCREATE
ENDEXTENDED
)
E_AXIOMS
  VARS mlistOfAttrKey : *MListOfAttrKey*
   ALL mlistOfAttrKey : *MListOfAttrKey*
    (not(is_in(nextE_1(mlistOfAttrKey), mlistOfAttrKey)) = true)

    ⋮

    (not(is_in(nextE_N(mlistOfAttrKey), mlistOfAttrKey)) = true);
ENDAXIOMS

Sinn des Export-Axiom: Mit dem Export-Axiom wird sichergestellt, daß die "next"-Operation zu einer gegebenen Monoliste von Schlüsselattributen immer ein neues, in der Menge nicht enthaltenes Schlüsselattribut liefert.

3.2 Der Modul NEXT_OPERATIONEN

Die aus der obigen Schablonen resultierenden Spezifikationen werden in dem Modul NEXT_OPERATIONEN (vergleiche mit Abbildung 5 auf Seite 13) mittels des OBSCURE Konstruktors PLUS zusammengesetzt. Für ein zu spezifizierendes E/R-Schema, in dem die benannten Attributtypen, die die Schlüsseleigenschaft in mindestens einem benannten Entitytyp haben, durch die Sorten $ANname_1$, ..., $ANname_7$ spezifiziert werden, sieht der Modul NEXT_OPERATIONEN folgendermaßen aus:

Der Modul NEXT_OPERATIONEN
(INCLUDE MK_NEXT_ATTRANAME1) PLUS
(INCLUDE MK_NEXT_ATTRANAME2) PLUS

(INCLUDE MK_NEXT_ATTRANAME3) PLUS
(INCLUDE MK_NEXT_ATTRANAME4) PLUS
(INCLUDE MK_NEXT_ATTRANAME5) PLUS
(INCLUDE MK_NEXT_ATTRANAME6) PLUS
(INCLUDE MK_NEXT_ATTRANAME7)

3.3 Konkrete und abstrakte Schlüssel

Jede Entity eines bestimmten benannten Entitytyps muß durch ihre Schlüsselattribute innerhalb der Datenbank identifizierbar sein. Die Schlüsselattribute einer Entity bilden den sogenannten "konkreten Schlüssel" dieser Entity. Eine genauere und motivierendere Beschreibung der konkreten Schlüssel findet man in [Hec93]. Bei einem konkreten Schlüssel einer Entity handelt es sich, aus einer abstrakten Sichtweise heraus, lediglich um eine Ansammlung dieser Schlüsselattribute. Dies würde zur Identifizierung von Entities ausreichen. Aus verschiedenen Gründen, wie zum Beispiel datenschutzrechtlichen Gründen, kann es für jeden konkreten Schlüssel eine eindeutige Kodierung geben, aus der die Komponenten des konkreten Schlüssels, nämlich Einträge in der Entity, weder ersichtlich, noch auf einfache Art und Weise berechenbar sind. Auch dieses Problem ist genauer in [Hec93] beschrieben.
Aus diesen Gründen müssen wir noch folgende Dinge modellieren: Zu jedem benannten Entitytyp eine Sorte, die die konkreten Schlüssel der Entities dieses benannten Entitytyps modelliert. Diese spezifizieren wir als Tupel, welche gebildet werden aus den Schlüsselattributen dieser Entities. Zum anderen muß eine Kodierungsoperation spezifiziert werden, die aus einem konkreten Schlüssel einen abstrakten Schlüssel berechnet. Die Bildsorte dieser Operation wird aus der Attributebene importiert. Die Semantik dieser Operation ist natürlich spezifisch bezüglich des jeweiligen benannten Entitytyps, sprich spezifisch bezüglich der benannten Schlüsselattributtypen eines benannten Entitytyps eines konkreten zu spezifizierenden E/R-Schemas. Dieser Teil wird deshalb auch bewußt in der Schablone offen gelassen. In der folgenden Schablone ist "EName" der Name des benannten Entitytyps, für den die konkreten Schlüssel spezifiziert werden und "ATyp$_i$" der Name des i-ten benannten Schlüsselattributtyps. Der aus der Schablone resultierende Modul importiert die benannten Attributtypen der Attribute, die gerade in den betreffenden Entities als Schlüsselattribute ausgezeichnet sind. Die benannten Attributtypen sind in der Attributebene spezifiziert worden. Weiterhin importiert die Spezifikation die Sorte der abstrakten Schlüssel. Hierfür kann irgendeine Sorte verwendet werden (zum Beispiel *Nat*), mit der die abstrakten Schlüssel modelliert werden können. Die Spezifikation dieser Sorte

liegt unterhalb der hier modellierten Datenmodellebene und stammt aus der Wertebereichebene der Attribute. Die Schablone zur Erstellung einer Spezifikation für die Sorte eines konkreten Schlüssels, sowie der Operationen konKey_*EName* und codeKonKey_*EName* für die Entities mit Namen "EName" sieht folgendermaßen aus:

```
## Der Modul MK_n_KEY_EName
((INCLUDE MK_n_TUPEL
 (SORTS ATyp1, ..., ATypN)
 [SORTS KonKey_EName
  OPNS set-n-th : ATypN KonKey_EName -> KonKey_EName,
         get-n-th : KonKey_EName -> ATypN,
                  ⋮
         setfst : ATyp1 KonKey_EName -> KonKey_EName,
         getfst : KonKey_EName -> ATyp1])
 E_RENAME OPNS _/_/.../_ AS OPNS konKey_EName
 EXTENDED BY
 IMPORTS
  SORTS
   AKey_EName
  OPNS
  ... Operationen auf der/den Sorte(n) AKey_EName (ATyp1, ... ATypN),
  ... um einen neuen abstrakten Schlüssel berechnen zu können.
  CREATE
    OPNS
      codeKonKey_EName : KonKey_EName -> AKey_EName
  SEMANTICS
  ... Hier muß der Spezifizierer die Semantik der Operation codeKonKey_EName
  ... angeben.
  ENDCREATE)
PARAMS
  (SORTS ATyp1, ..., ATypN,
           AKey_EName,
           KonKey_EName
   OPNS ... die Operationen auf der/den Sorte(n) AKey_EName und
          ... (ATyp1, ... ATypN))
  [OPNS konKey_EName, codeKonKey_EName]
```

3.4 Der Modul SCHLUESSEL

Für jeden benannten Entitytyp $EName_i$, $1 \leq i \leq N$, eines konkreten E/R-Schemas wurde die obige Schablone entsprechend mit der Anzahl der Schlüsselattribute und dem Namen des benannten Entitytyps instanziiert. Alle diese Spezifikationen werden in dem Modul SCHLUESSEL mittels des OBSCURE Konstruktors PLUS zusammengesetzt. Die Schablone sieht folgendermaßen aus (dabei seien j_i die Anzahl der Schlüsselattribute in dem benannten Entitytyp mit Namen $EName_i$):

Der Modul SCHLUESSEL
(INCLUDE MK_j_1_KEY_$EName_1$) PLUS
PLUS
⋮
PLUS
(INCLUDE MK_j_N_KEY_$EName_1$)

3.5 Der Durchreichmodul

Dieser drückt nur aus, daß wir die Sorten, die nicht zur Spezifikation eines konkreten Schlüssels verwendet werden, einfach "nach oben" durchreichen. Der Modul hat an dieser Stelle eine reine Strukturierungsfunktion.

3.6 Der Modul SCHLUESSELEBENE

Dieser Modul setzt die drei vorangegangenen Moduln NEXT_OPERATIONEN, SCHLUESSEL und DURCHREICHMODUL zusammen zur Spezifikation der Schlüsselebene:

```
## Der Modul SCHLUESSELEBENE
(
 (INCLUDE NEXT_OPERATIONEN)
      X_COMPOSE
 (INCLUDE SCHLUESSEL)
)
PLUS
(INCLUDE DURCHREICHMODUL)
```

Teil II
Die Datenbankebene

4 Entities

4.1 Vorbemerkungen

Seien eine zu spezifizierende Entity mit dem Namen *EName* gegeben, sowie deren N Attribute mit den Namen $AttrName_1$, ..., $AttrName_N$ und Attributtypen D_1, ..., D_N, deren Domains als Sorten $AttrD_1$, ... $AttrD_N$ spezifiziert wurden (Siehe S. 15). Der Domain eines Attributtyps ist eine Menge von Werten, die ein Attribut annehmen kann.

- Die zu spezifizierende Entity mit Namen *EName* wird als Sorte *EName* spezifiziert. Die Träger der Sorte *EName* stellen wir uns als N-Tupel über den Domain-Sorten $AttrD_1$, ..., $AttrD_N$ vor. Die Sorte *EName* wird also durch einen parametrisierten Aufruf des Moduls mit Namen MK_N-ATTRIBUTES-ENTITY, welcher wiederum eine N-Tupel Spezifikation parametrisiert aufruft: die Import-Parameter sind die Attributtypen $AttrD_1$, ..., $AttrD_N$ und die Export-Parameter die Sorte *EName* und die Operationen über den Trägern der Sorte *EName*.
  Anmerkung: N-Tupel Spezifikationen sind in der Standard-Modul-Datenbank der Spezifikationsumgebung von OBSCURE enthalten.

- Der Spezifizierer, der den Modul MK_N-ATTRIBUTES-ENTITY verwenden wird, muß darauf achten, daß sich die Namen der Operationen auf den Namen *EName* beziehen sollen.

- Der parametrisierte Modul MK_N-ATTRIBUTES-ENTITY kann zur Spezifikation aller Entities mit N Attributen verwendet werden.

In Anlehnung an die Spezifikation von Entities in SPECTRUM (Siehe [Het93]) wird zum Kreieren einer Entity eine Operation "create*EName*" spezifiziert: diese ist einfach ein "renaming" des n-Tupel Konstruktors aus der n-Tupel Spezifikation.
Eine Entity enthält im allgemeinen mehrere sogenannte "notwendige" (mandatory) Attribute. Das bedeutet einfach, daß der Wert dieser Attribute in einer Entity nie undefiniert sein darf. Das heißt konkret, daß der Wert des Attributs nicht der UNDEF Träger des entsprechenden Attributtyps

sein darf, da wir diesen zur Modellierung der Undefiniertheit speziell eingeführt haben. Da wir dies aber nicht von vorneherein festlegen wollen, sondern eher in Anlehnung an [Hoh93], die Formulierung bzw. Überprüfung der notwendigen Attribute als statische Integritätsbedingungen formulieren wollen, spezifizieren wir "check"-Funktionen für jedes Attribut einer Entity. Eine "check"-Funktion eines Attributs nimmt als Argument eine Entity und überprüft, ob der Wert dieses Attributs nicht der UNDEF Träger des Attributtyps ist. Diese "check"-Funktionen werden "nach Außen" zur Verfügung gestellt, d.h. sie werden von der Gesamtspezifikation der Datenmodellebene exportiert. Damit können die statischen Integritätsbedingungen angegeben werden. Ein Beispiel: die Entities des benannten Entityps mit Namen *Patient* beinhalten Attribute vom benannten Attributtyp mit Namen *PatId*. Die "check"-Funktion für dieses Attribut hat folgende Syntax und folgende Definition:

```
Patient-PatId-ex : Patient -> bool
Patient-PatId-ex(patient) <- not(PatId(patient) = UNDEF_PatId)
```

Anmerkung: Mit Hilfe eines Moduls MK_N-ATTRIBUTES-ENTITY können alle benannten Entitypen von Entities mit N Attributen spezifiziert werden.

4.2 Die Schablone

```
## Der Modul MK_N-ATTRIBUTES-ENTITY
(
(
 (INCLUDE MK_N-TUPEL
     (SORTS AttrD_1, ..., AttrD_N)
     [SORTS EName

     OPNS setattrName_1 **,
          attrName_1 **,
          :
          setattrName_N **,
          attrName_N **]

E_RENAME OPNS {_/_/.../_} AS createEName
)
EXTENDED BY
IMPORTS
```

OPNS
UNDEF_D$_1$: -> AttrD$_1$
⋮
UNDEF_D$_N$: -> AttrD$_N$
_ = _ : AttrD$_1$ AttrD$_1$ -> bool
⋮
_ = _ : AttrD$_N$ AttrD$_N$ -> bool
CREATE
OPNS
EName-AttrName$_1$-ex : *EName* -> bool
⋮
EName-AttrName$_N$-ex : *EName* -> bool
SEMANTICS
VARS
ename : *EName*
PROGRAMS
EName-AttrName$_1$-ex(ename)
<- not(attrname$_1$(ename) = UNDEF_D$_1$);
⋮
EName-AttrName$_N$-ex(ename)
<- not(attrname$_N$(ename) = UNDEF_D$_N$);
ENDCREATE
)
PARAMS
(SORTS *AttrD*$_1$, ..., *AttrD*$_N$
OPNS UNDEF_D$_1$ **,
⋮
UNDEF_D$_N$ **) [SORTS *EName*
OPNS *setattrName*$_1$ **,
attrName$_1$ **,
⋮
setattrName$_N$ **,
attrName$_N$ **,
EName-AttrName$_1$-ex **,
⋮
EName-AttrName$_N$-ex **]

4.3 Beispiel

Die Anwendung der Schablone, sowie der daraus resultierenden, parametrisierten Spezifikation, wollen wir am Beispiel des in Abbildung 1 auf Seite 4 gezeigten benannten Entitytyps mit Namen PATIENT zeigen: die aus der Schablone resultierende Spezifikation sieht folgendermaßen aus:

```
## Der Modul MK_ELF-ATTRIBUTES-ENTITY
(
(
 (INCLUDE MK_ELF-TUPEL

     (SORTS AttrD1, AttrD2, AttrD3, AttrD4,
            AttrD5, AttrD6, AttrD7, AttrD8,
            AttrD9, AttrD10, AttrD11)

     [SORTS EName

     OPNS
       setattrName1 **, attrName1 **, setattrName2 **, attrName2 **,
       setattrName3 **, attrName3 **, setattrName4 **, attrName4 **,
       setattrName5 **, attrName5 **, setattrName6 **, attrName6 **,
       setattrName7 **, attrName7 **, setattrName8 **, attrName8 **,
       setattrName9 **, attrName9 **, setattrName10 **, attrName10 **,
       setattrName11 **, attrName11 **]

E_RENAME OPNS {_/_/_/_/_/_/_/_/_/_/_} AS createEName
)
EXTENDED BY
IMPORTS
OPNS

 UNDEF_D1 : -> AttrD1
 UNDEF_D2 : -> AttrD2
 UNDEF_D3 : -> AttrD3
 UNDEF_D4 : -> AttrD4
 UNDEF_D5 : -> AttrD5
 UNDEF_D6 : -> AttrD6
 UNDEF_D7 : -> AttrD7
 UNDEF_D8 : -> AttrD8
 UNDEF_D9 : -> AttrD9
 UNDEF_D10 : -> AttrD10
 UNDEF_D11 : -> AttrD11
 _ = _ : AttrD1 AttrD1 -> bool
```

```
_ = _ : AttrD2 AttrD2 -> bool
_ = _ : AttrD3 AttrD3 -> bool
_ = _ : AttrD4 AttrD4 -> bool
_ = _ : AttrD5 AttrD5 -> bool
_ = _ : AttrD6 AttrD6 -> bool
_ = _ : AttrD7 AttrD7 -> bool
_ = _ : AttrD8 AttrD8 -> bool
_ = _ : AttrD9 AttrD9 -> bool
_ = _ : AttrD10 AttrD10 -> bool
_ = _ : AttrD11 AttrD11 -> bool


CREATE
  OPNS

    EName-AttrName1-ex : EName -> bool
    EName-AttrName2-ex : EName -> bool
    EName-AttrName3-ex : EName -> bool
    EName-AttrName4-ex : EName -> bool
    EName-AttrName5-ex : EName -> bool
    EName-AttrName6-ex : EName -> bool
    EName-AttrName7-ex : EName -> bool
    EName-AttrName8-ex : EName -> bool
    EName-AttrName9-ex : EName -> bool
    EName-AttrName10-ex : EName -> bool
    EName-AttrName11-ex : EName -> bool

SEMANTICS
  VARS
    ename : EName
  PROGRAMS
  EName-AttrName1-ex(ename)
       <- not(attrname1(ename) = UNDEF_D1);
  EName-AttrName2-ex(ename)
       <- not(attrname2(ename) = UNDEF_D2);
  EName-AttrName3-ex(ename)
       <- not(attrname3(ename) = UNDEF_D3);
  EName-AttrName4-ex(ename)
       <- not(attrname4(ename) = UNDEF_D4);
  EName-AttrName5-ex(ename)
       <- not(attrname5(ename) = UNDEF_D5);
```

```
   EName-AttrName6-ex(ename)
        <- not(attrname6(ename) = UNDEF_D6);
   EName-AttrName7-ex(ename)
        <- not(attrname7(ename) = UNDEF_D7);
   EName-AttrName8-ex(ename)
        <- not(attrname8(ename) = UNDEF_D8);
   EName-AttrName9-ex(ename)
        <- not(attrname9(ename) = UNDEF_D9);
   EName-AttrName10-ex(ename)
        <- not(attrname10(ename) = UNDEF_D10);
   EName-AttrName11-ex(ename)
        <- not(attrname11(ename) = UNDEF_D11);
ENDCREATE
)
PARAMS

     (SORTS AttrD1, AttrD2, AttrD3, AttrD4,
            AttrD5, AttrD6, AttrD7, AttrD8,
            AttrD9, AttrD10, AttrD11
      OPNS UNDEF_D1 : -> AttrD1
           UNDEF_D2 : -> AttrD2
           UNDEF_D3 : -> AttrD3
           UNDEF_D4 : -> AttrD4
           UNDEF_D5 : -> AttrD5
           UNDEF_D6 : -> AttrD6
           UNDEF_D7 : -> AttrD7
           UNDEF_D8 : -> AttrD8
           UNDEF_D9 : -> AttrD9
           UNDEF_D10 : -> AttrD10
           UNDEF_D11 : -> AttrD11

   [SORTS EName

   OPNS
     setattrName1 **, attrName1 **, setattrName2 **, attrName2 **,
     setattrName3 **, attrName3 **, setattrName4 **, attrName4 **,
     setattrName5 **, attrName5 **, setattrName6 **, attrName6 **,
     setattrName7 **, attrName7 **, setattrName8 **, attrName8 **,
     setattrName9 **, attrName9 **, setattrName10 **, attrName10 **,
     setattrName11 **, attrName11 **,
     EName-AttrName1-ex **,
     EName-AttrName2-ex **,
```

EName-AttrName3-ex **,
EName-AttrName4-ex **,
EName-AttrName5-ex **,
EName-AttrName6-ex **,
EName-AttrName7-ex **,
EName-AttrName8-ex **,
EName-AttrName9-ex **,
EName-AttrName10-ex **,
EName-AttrName11-ex**]

Zur Spezifikation der oben erwähnten Beispiel-Entity PATIENT, wird der Modul MK_ELF_-ATTRIBUTE-ENTITY parametrisiert über dem Namen "Patient" der Entity, den Attributtypen AttrPatId, AttrName, AttrGeschlecht, ..., sowie den Namen der Operationen. Für deren Namensgebung werden die Namen der Attribute verwendet. Daraus entsteht folgende Spezifikation:

Der Modul PATIENT

```
(INCLUDE MK_ELF-ATTRIBUTE-ENTITY
  (SORTS AttrPatId, AttrName, AttrGeschlecht, AttrDatum, AttrOrt,
         AttrKostenträger, AttrAdresse, AttrHausarzt,
         AttrKörperdaten, AttrStation, AttrZimmer
   OPNS UNDEF_PatId : -> AttrPatId,
        UNDEF_Name : -> AttrName,
        UNDEF_Geschlecht : -> AttrGeschlecht,
        UNDEF_Datum : -> AttrDatum,
        UNDEF_Ort : -> AttrOrt,
        UNDEF_Kostenträger : -> AttrKostenträger,
        UNDEF_Adresse : -> AttrAdresse,
        UNDEF_Hausarzt : -> AttrHausarzt,
        UNDEF_Körperdaten : -> AttrKörperdaten,
        UNDEF_Station : -> AttrStation,
        UNDEF_Zimmer : -> AttrZimmer)
  [SORTS Patient,
   OPNS setPatId **, patId **,
        setName **, name,
        setGeschlecht **, geschlecht **,
        setDatum **, datum **,
        setOrt **, ort **,
        setKostenträger **, kostenträger **,
        setAdresse **, adresse **,
        setHausarzt **, hausarzt **,
```

setKörperdaten **, körperdaten **,
setStation **, station **,
setZimmer **, zimmer **,
Patient-PatId-ex **,
Patient-Name-ex **,
Patient-Geschlecht-ex **,
Patient-Datum-ex **,
Patient-Ort-ex **,
Patient-Kostenträger-ex **,
Patient-Adresse-ex **,
Patient-Hausarzt-ex **,
Patient-Körperdaten-ex **,
Patient-Station-ex **,
Patient-Zimmer-ex **]

4.4 Entities und abstrakte Schlüssel

Zu jedem Träger der Sorte *EName* muß es möglich sein einen abstrakten Schlüssel zu berechnen. Diese Berechnung baut im allgemeinen auf einem Tupel von Attributen dieses Trägers auf: diese Attribute bilden den konkreten Schlüssel einer Entity. Deshalb werden zwei Operationen importiert:

1. Die Operation KonKey_*EName* : $AttrD_{i1}$, ..., $AttrD_{il}$ -> *KonKey_EName*. Diese Berechnet aus den l Vertretern der Schlüsselattribute einer Entity E den konkreten Schlüssel dieser Entity (Siehe Abschnitt **3.3 Konkrete und abstrakte Schlüssel**),

2. Die Kodierungsoperation code*KonKey_EName* : *KonKey_EName* -> *AKey_EName*, die den konkreten Schlüssel auf einen abstrakten Schlüssel abbildet.

Die Spezifikation der Operation *KeyEName* erfolgt in einem Modul MK_l-KEY-OPERATION, wobei l die Anzahl der Schlüsselattribute ist. Dieser Modul ist parametrisiert über den folgenden Sorten:

- *EName*
- *AttrDi*1, ..., $AttrD_{il}$, $1 \leq l \leq N$: dies sind die Attributypen der Schlüsselattribute.
- *KonKey_EName* : die Sorte, die die konkreten Schlüssel modelliert.
- *AKey_EName* : die Sorte, die die abstrakten Schlüssel modelliert.

und folgenden Operationen

- KonKey_*EName* : $AttrD_{i_1}$, ..., $AttrD_{i_l}$ -> *KonKey_EName*
- code*KonKey_EName* : *KonKey_EName* -> *AKey_EName*
- $AttrD_{i_1}$: *EName* -> $AttrD_{i_1}$
- $\vdots$
- $AttrD_{i_l}$: *EName* -> $AttrD_{i_l}$

4.4.1 Die Schablone

Der Modul MK_*l*-KEY-OPERATION
(
IMPORTS
  SORTS
    EName
    $AttrD_{i_1}$, ..., $AttrD_{i_l}$, $1 \leq l \leq N$
    KonKey_EName,*AKey_EName*
  OPNS
    konKey_*EName* : $AttrD_{i_1}$, ..., $AttrD_{i_l}$ -> *KonKey_EName*
    code*KonKey_EName* : *KonKey_EName* -> *AKey_EName*
    $attrname_{i_1}$: *EName* -> $AttrD_{i_1}$
        $\vdots$
    $attrname_{i_l}$: *EName* -> $AttrD_{i_l}$
CREATE
  OPNS
    key*EName* : *EName* -> *AKey_EName*
  SEMANTICS
    VARS
      ename : *EName*
    PROGRAMS

      key*EName*(ename) <-
       codeKonKey_EName(konKey_EName(attrname$_{i_1}$(ename),
                                                  $\vdots$
                                      attrname$_{i_l}$(ename)));

ENDCREATE
)
PARAMS
(SORTS
EName
$AttrD_{i_1}, \ldots, AttrD_{i_l}$,
KonKey_EName,*AKey_EName*
OPNS
konKey_*EName* : $AttrD_{i_1}, \ldots, AttrD_{i_l}$ -> *KonKey_EName*
code*KonKey_EName* : *KonKey_EName* -> *AKey_EName*
$attrname_{i_1}$: *EName* -> $AttrD_{i_1}$
⋮
$attrname_{i_l}$: *EName* -> $AttrD_{i_l}$)
[OPNS key*EName*]

4.4.2 Beispiel

Um die obige Schablone zu motivieren, wenden wir sie auf eine Entity AUFENTHALT an, deren Struktur in Abbildung 7 auf Seite 35 dargestellt ist. Dazu nehmen wir an, daß es die Sorten *Aufenthalt*, *AttrNat*, *AttrPa-*

Entity AUFENTHALT

Attributname	Attributtyp	
AufhFolgeNr	Nat	(mandatory, primary key)
PatId	PatId	(mandatory, primary key)
AufnahmeDatum	DateTime	(mandatory)

Abbildung7: Die Beispiel Entity AUFENTHALT

tId und *AttrDateTime* gibt, sowie die Sorten *KonKey_Aufenthalt* und *AKey_Aufenthalt*, die respektive, die konkreten und die abstrakten Schlüssel modellieren. Der, durch die Instanziierung der Schablone entstehende Modul sieht folgendermaßen aus:

```
## Der Modul MK_ZWEI-KEY-OPERATION

(IMPORTS
 SORTS
    EName,
    ATyp1, ATyp2,
    KonKey_EName, AKey_EName
 OPNS
    konKey_EName : ATyp1 ATyp2 -> KonKey_EName,
    codeKonKey_EName : KonKey_EName -> AKey_EName,
    attrname1 : EName -> ATyp1,
    attrname2 : EName -> ATyp2
 CREATE
 OPNS
    keyEName : EName -> AKey_EName
 SEMANTICS
  VARS
    ename : EName
  PROGRAMS
    keyEName(ename)
       <- codeKonKey_EName(konKey_EName(attrname1(ename)
                                        attrname2(ename)));
 ENDCREATE)
PARAMS
 (SORTS
    EName,
    ATyp1, ATyp2,
    KonKey_EName, AKey_EName
  OPNS
     konKey_EName : ATyp1 ATyp2 -> KonKey_EName,
     codeKonKey_EName : KonKey_EName -> AKey_EName,
     attrname1 : EName -> ATyp1,
     attrame2 : EName -> ATyp2)
 [OPNS keyEName : EName -> AKey_EName]
```

Um die Operation *keyAufenthalt : Aufenthalt -> AKey_Aufenthalt* zu spezifizieren, wird obige Spezifikation mit den entsprechenden Parametern wie folgt aufgerufen:

```
## Der Modul KEY_AUFENTHALT
INCLUDE MK_ZWEI-KEY-OPERATION
          (SORTS Aufenthalt, AttrNat, AttrPatId, KonKey_Aufenthalt,
```

```
            AKey_Aufenthalt
      OPNS konKey_Aufenthalt,
           codeKonKey_Aufenthalt,
           AufhFolgeNr : Aufenthalt -> AttrNat,
           PatId : Aufenthalt -> AttrPatId)
    [OPNS keyAufenthalt : Aufenthalt -> AKey_Aufenthalt]
```

4.4.3 Bemerkung

Um entsprechende Schlüsseloperationen für alle Arten von Entities mit N Attributen zu spezifizieren, benötigen wir N parametrisierte Spezifikationen MK_*l*-KEY-OPERATION_L.

4.5 Spezifikation einer Entity

Die Spezifikation einer Entity unterteilt sich gemäß dem vorher Erwähnten in zwei Teile: zum einen in die Spezifikation der Sorte *EName* und zum anderen in die Spezifikation der Schlüsseloperationen für diesen benannten Entitytyp. Die Schablone dazu sieht für einen gegebenen benannten Entitytyp mit Namen *EName* und benannten Attributtypen mit Namen $AttrName_1$, ..., $AttrName_N$ und l Schlüsselattributen folgendermaßen aus:

```
## Der Modul EName
((INCLUDE MK_N-ATTRIBUTES-ENTITY
    (SORTS AttrName_1, ..., AttrName_N
     OPNS UNDEF_Name_1, ..., UNDEF_Name_N)
    [SORTS EName
     OPNS setName_1 **, Name_1 **,
                    ⋮
          setName_N **, Name_N **
          EName-Name_1-ex **,
                ⋮
          EName-Name_N-ex **])
)
X_COMPOSE
(INCLUDE MK_l-KEY-OPERATION
   (SORTS EName, AttrName_i1, ..., AttrName_iN)
          Konkey_EName, AKey_EName
    OPNS konKey_EName : AttrName_i1, ..., AttrName_il -> Konkey_EName
         codeKonKey_EName : Konkey_EName -> AKey_EName
         attrname_i1 : EName -> AttrName_i1
```

$\vdots$
attr$name_{i_l}$: *EName* -> Attr$Name_{i_l}$)
[OPNS key*Entityname*])

4.6 Der Modul ENTITIES

Der Modul ENTITIES (Siehe Abbildung 5 auf Seite 13) setzt die obigen Spezifikatoinen zusammen mittels des OBSCURE Konstruktors PLUS. Somit entsteht folgende Schablone für den Modul ENTITIES, wobei die Namen der im E/R-Schema enthaltenen benannten Entitytypen die folgenden sind: $EName_1, \ldots, EName_N$.

Der Modul ENTITIES
(INCLUDE ENAME$_1$) PLUS ...PLUS (INCLUDE ENAME$_N$)

5 Ansammlungen von Entities

Der folgende Abschnitt beschäftigt sich mit der Spezifikation einer Ansammlung von Entities. Zu deren Spezifikation verwenden wir keine Mengen, da dies im allgemeinen nicht konstruktiv nicht. Eine ausführliche Beschreibung dieser Problematik findet man in [Hec93]. Aus diesem Grund verwenden wir sogenannte **Monolisten**, die auch in [Hec93] eingeführt werden.
Die Spezifikation von "Ansammlungen von Entities" unterteilt sich in vier Teile:

1. Zum einen muß erst einmal eine Sorte *MListOfE* spezifiziert werden, wobei *E* der Name einer Entity ist. Dies geschieht mittels des im OBSCURE-Systems zur Verfügung stehenden Standard-Modul MK_MONOLIST.

2. Darüber hinaus müssen die in der SPECTRUM-Spezifikation verwendeten Operationen zur Verfügung gestellt werden. Dies sind die Operationen del*E*: *E MListOfE* -> *MListOfE*, put*E*: *E MListOfE* -> *MListOfE*, get*E*: *Akey_E MListOfE* -> *E* und update*E*: *E MListOfE* -> *MListOfE*. Einige davon sind Operationen, die aus parametrisiert aufgerufenen MK_MONOLIST Spezifikation stammen, und umbenannt werden. Dies sind die Operationen del*E*: *E MListOfE* -> *MListOfE* und put*E*: *E MListOfE* -> *MListOfE*.

3. Die nicht aus der Spezifikation MK_MONOLIST stammenden Operationen müssen spezifiziert werden. Dies sind die Operationen get*E*: *Akey_E MListOfE* -> *E* und update*E*: *E MListOfE* -> *MListOfE*. Dabei ist die Sorte *Akey_E* die Sorte der abstrakten Schlüssel der Entities mit Namen *E*.

4. Desweiteren müssen wir zusätzliche, nicht in der SPECTRUM-Spezifikation spezifizierte Operationen spezifizieren. Dies ist die oben erwähnte Operation is_in*E*: *E MListOfE* -> *bool*, die *true* liefern soll, falls in der Monoliste eine Entity mit demselben abstrakten Schlüssel, wie dem der als Argument übergebenen Entity; sonst *false*.
   Um die statischen Integritätsbedingungen formulieren zu können, spezifieren wir eine "check"-Funktion check*MListOfE*: *MListOfE* -> *bool*. Diese soll true liefern, falls es in der Argument-Monoliste keine zwei Entities mit demselben abstrakten Schlüssel gibt. Die Namen der "check"-Funktionen beziehen sich jeweils auf den Namen der Entity, für die sie spezifiziert werden. Für die Entity PATIENT in Abbildung 1 auf Seite 4 hieße die "check"-Funktion *checkPatient* und hätte die Stelligkeit *checkPatient* : MListOfPatient -> bool.

Wir brauchen keine Schablone anzugeben, sondern können eine parametrisierte Spezifikation angeben: Diese ist parametrisiert über die Sorte der Entities, also deren Namen (siehe dazu Abschnitt 4.1), der Sorte der abstrakten Schlüssel dieser Entities und der Operation, die zu einer Entity den abstrakten Schlüssel liefert. Die Export-Parameter sind dann die Namen der spezifizierten Operationen, und zwar nicht nur der explizit im Spezifikationstext spezifizierten, sondern auch der, die aus der parametrisierten MK_MONOLIST Spezifikation stammen.

5.1 Die Spezifikation

```
## Der Modul MK_COLLECTION_OF_ENTITIES
( (
(INCLUDE MK_MONOLIST
     (SORTS E)
     [SORTS MListOfE])


E_RENAME OPNS del,
                    append
              AS OPNS delE,
                      putE

EXTENDED BY
IMPORTS
   SORTS AKey_E
   OPNS
   KeyE : E -> Akey_E,
   _ = _ : Akey_E Akey_E -> bool,
   _ = _ : MListOfE MListOfE -> bool
CREATE
 OPNS
   is_inE : E MListOfE -> bool
   getE : AKey_E MListOfE -> E
   updateE : E MListOfE -> MListOfE
   checkE : MListOfE -> bool

SEMANTICS
VARS
mlistOfE, MListOfE1 : MListOfE
e, e1 : E
aKeyE : AKey_E
```

PROGRAMS

```
is_in E(e, mlistOf E)
     <- IF not(mlistOf E = empty_MList)
           THEN
           LET e1 : head(mlistOf E) IN
                 IF key E(e1) = key E(e)
                    THEN true
                    ELSE is_in E(e, del E(e1, mlistOf E))
                 FI;
           TEL;
           ELSE false;
        FI;

get E(aKey_E, mlistOf E)
   <- IF not(mlistOf E = empty_MList)
         THEN
         LET e : head(mlistOf E) IN
               IF key E(e)=aKey_E
                  THEN e
                  ELSE get E(aKey_E, del E(e, mlistOf E))
               FI;
         TEL;
         ELSE ERROR(E);
      FI;

update E(e, mlistOf E)
       <- IF not(mlistOf E = empty_MList)
             THEN
             LET e1 : head(mlistOf E) IN
                   IF key E(e1) = key(e)
                      THEN append(e, del E(e1, mlistOf E))
                      ELSE append(e1, update E(e, del E(e1, mlistOf E)
                   FI;
             TEL;
             ELSE ERROR(E);
          FI;
```

```
checkE(mlistOfE)
      <- IF not(mlistOfE = empty_MList)
            THEN
            LET e1 : head(mlistOfE) IN
                  LET mlistOfE1 : delE(e1, mlistOfE) IN
                        not(is_inE(e1, mlistOfE1))
                        and
                        checkE(mlistOfE1)
                  TEL;
            TEL;
            ELSE true;
         FI;


ENDCREATE
ENDEXTENDED
)
PARAMS
  (SORTS E, AKey_E
  OPNS keyE)
  [SORTS mlistOfE
  OPNS
       is_inE : MListOfE E -> bool
       getE : MListOfE AKey_E -> E
       updateE : MListOfE E -> MListOfE ]
```

5.2 Beispiel

Um die Anwendung der obigen Spezifikation zu demonstrieren, wenden wir die parametrisierte Spezifikation auf eine Sorte *Patient* an. Die Entities mit Namen **Patient** haben abstrakte Schlüssel, die in der Sorte *AKey_Patient* spezifiziert sind. Die Operation, die zu einem gegebenen Träger der Sorte *Patient* den abstrakten Schlüssel liefert, heißt key*Patient*. Die Anwendung der obigen Spezifikation soll uns nun eine Monoliste über der Sorte *Patient* liefern. Die Anwendung sieht folgendermaßen aus:

```
## Der Modul PATIENT_MONOLISTEN
         (INCLUDE MK_COLLECTION_OF_ENTITIES
                  (SORTS Patient,
                         AKey_Patient
                   OPNS keyPatient)
```

```
[SORTS MListOfPatient
 OPNS delPatient **,
      putPatient **,
      is_inPatient **,
      getPatient **,
      updatePatient **,
      checkPatient ** ])
```

Durch Anwendung der parametrisierten Spezifikation mit obigen Im- und Export-Parametern, erhält man die gewünschte Sorte, sowie die entsprechenden Operationen.

5.3 Der Modul ENTITY_MONOLISTEN

Die einzelnen Spezifikationen der Monolisten über den Entities eines zu spezifizierenden E/R-Schema mit N benannten Entitytypen werden in dem Modul ENTITY_MONOLISTEN mittels des OBSCURE Konstruktors PLUS zusammengesetzt (vergleiche hierzu Abbildung 5 auf Seite 13). In der folgenden Schablone sind $EName_1$, ..., $EName_N$ die Namen der benannten Entitytypen.

```
## Der Modul ENTITY_MONOLISTEN
(INCLUDE ENAME1_MONOLISTEN)
         PLUS
           ...
         PLUS
(INCLUDE ENAMEN_MONOLISTEN)
```

6 Ansammlung aller Entity-Ansammlungen

6.1 Die Schablone

Die im vorherigen Abschnitt beschriebene parametrisierte Spezifikation von Entity-Ansammlungen wird bei der Spezifikation eines konkreten E/R-Schemas mehrfach verwendet und es entstehen mehrere Sorten von Entity-Ansammlungen, und zwar für jeden benannten Entitytyp des E/R-Schemas eine. Jetzt müssen noch alle diese Entity-Ansammlungen in einer "großen" Ansammlung zusammengefaßt werden. Dies geschieht mit Hilfe einer parametrisierten M-Tupel Spezifikation. Diese erhält als Import-Parameter die Sorten der Entity-Ansammlungen und liefert als Export-Parameter eine Sorte M_MListsOfEntities. Diese Spezifikation kann natürlich nicht universell, daß heißt bei jedem E/R-Schema anwendbar sein, da die Anzahl der benannten Entitytypen variieren kann. Deshalb geben wir an dieser Stelle nur eine OBSCURE-Schablone an, die als "syntaktischen Parameter" das M aus dem Namen der (M-)Tupel Spezifikation hat. Zusätzlich wird noch eine Konstante der Sorte DB_Entities spezifiziert, die die Ansammlung leerer Entity-Ansammlungen modelliert. Diese wird später zur Spezifikation der leeren Datenbank verwendet werden.

```
## Der Modul DB_ENTITIES
(INCLUDE MK_M_TUPEL
                (SORTS MListOfEntityname_1,
                                ⋮
                       MListOfEntityname_M)
                [SORTS DB_Entities,
                 OPNS set_MListOfEntityname_M,
                      get_MListOfEntityname_M,
                      ⋮
                      set_MListOfEntityname_1,
                      get_MListOfEntityname_1,
                      {_/.../_}])
EXTENDED BY
IMPORTS
 OPNS
   empty_MListOfEntityname_1 : -> MListOfEntityname_1,
                               ⋮
   empty_MListOfEntityname_M : -> MListOfEntityname_M
CREATE
```

```
  OPNS empty_DB_Entities : -> DB_Entities,
  SEMANTICS
  PROGRAMS
    empty_DB_Entities <- { empty_MListOfEntityname_1 /
                                    ⋮
                           empty_MListOfEntityname_M }
ENDCREATE
ENDEXTENDED
```

6.2 Ein Beispiel

Wenden wir die obige Schablone im Falle eines E/R-Schemas an, dessen benannte Entitytypen die folgenden Namen haben: Patient, Arzt, HK_UW, HK_Daten und HK_Befund. Aufgrund der Spezifikation der Entity-Ansammlungen, wurden bereits folgende Sorten spezifiziert: MListOfPatient, MListOfArzt, MListOfHK_UW, MListOfHK_Daten und MListOfHK_Befund. Da das betrachte E/R-Schema 5 benannte Entitytypen beinhaltet, ergibt sich die Spezifikation in diesem Fall durch Instantiierung von M durch FUENF und der entsprechenden Entitynamen.

```
## Der Modul DB_ENTITIES
(INCLUDE MK_FUENF_TUPEL
                (SORTS MListOfPatient,
                       MListOfArzt,
                       MListOfHK_UW,
                       MListOfHK_Daten,
                       MListOfHK_Befund)
                [SORTS DB_Entities,
                 OPNS set_HK_Befund,
                      get_HK_Befund,
                      set_HK_Daten,
                      get_HK_Daten,
                      set_HK_UW,
                      get_HK_UW,
                      set_Arzt,
                      get_Arzt,
                      set_Patient,
                      get_Patient,
                      {-/-/-/-/-}])
EXTENDED BY
```

```
IMPORTS
 OPNS
   empty_MListOfPatient : -> MListOfPatient,
   empty_MListOfArzt : -> MListOfArzt,
   empty_MListOfHK_UW : -> MListOfHK_UW,
   empty_MListOfHK_Daten : -> MListOfHK_Daten,
   empty_MListOfHK_Befund : -> MListOfHK_Befund
CREATE
  OPNS empty_DB_Entities : -> DB_Entities,
  SEMANTICS
  PROGRAMS
    empty_DB_Entities <- { empty_MListOfPatient /
                           empty_MListOfArzt /
                           empty_MListOfHK_UW /
                           empty_MListOfHK_Daten /
                           empty_MListOfHK_Befund };
ENDCREATE
ENDEXTENDED
```

7 Binäre Relationships

7.1 Die Spezifikation

Gemäß Abbildung 5 auf Seite 13 spezifizieren wir binäre Relationships. Eine binäre Relationship dient dazu, zwei Entities zweier benannter Entitytypen miteinander in Beziehung zu setzen und dadurch einen bestimmten Sachverhalt festzuhalten. Dazu ein Beispiel: angenommen, es seien zwei Entities zweier benannter Entitytypen mit respektiven Namen PATIENT und ARZT gegeben: dann könnte es sinnvoll sein, festzuhalten, daß der Arzt a den Patienten p behandelt. Hierzu wird eine binäre Relationship *behandelt* eingeführt, aus der für jeden Patient p sichtbar wird, welcher Arzt ihn behandelt.
Zur Spezifikation einer solchen binären Relationship, verwenden wir eine parametrisierte Spezifikation MK_BINARY_RELATION. Da Entities über ihre abstrakten Schlüssel identifiziert werden, verwenden wir diese als Grundlage für die Spezifikation der binären Relationships.
Über die eigentliche Spezifikation einer binären Relationship hinaus, müssen noch 3 verschiedene Arten von Operationen spezifiziert werden: Erstens wollen wir die gleichen Operationen, wie die SPECTRUM Spezifikation (siehe [Het93]) zur Verfügung stellen: Das heißt, Operationen wie *estRelationship* und *relRelationship* müssen extra spezifiziert werden. Zweitens müssen wir zusätzliche Operationen spezifizieren: Dies sind hauptsächlich Operationen, mit Hilfe derer geprüft werden kann, ob eine Entity in einer Relationship in Beziehung zu einer anderen Entity steht, de facto also, ob ein abstrakter Schlüssel in einer binären Relationship enthalten ist. Drittens werden, wie auch schon bei der Spezifikation von Entity-Ansammlungen, "check"-Funktionen für Relationships angegeben. Die "check"-Funktionen dienen dazu, den Typ einer Relationship überprüfen zu können, wobei eine Relationship vier verschiedene Typen haben kann: 1-1, 1-N, N-1 und N-N. Die Unterscheidung zwischen 1-N und N-1 Relationships ist hierbei zwar nicht zwingend, wird aber dennoch aus Komfortgründen gemacht.
Aus den obigen Anmerkungen entsteht folgende, parametrisierte Spezifikation: "RName" und "RTyp" sind dabei jeweils der Name und der Typ des zu spezifzierenden benannten Relationshiptyps.

```
 ## Der Modul MK_RELATIONSHIP
(
(INCLUDE MK_BINARY_RELATION
                (SORTS AKey1, AKey2)
                [SORTS RTyp
                 OPNS empty-RTyp])
```

```
EXTENDED BY
CREATE
OPNS

## 1. Die SPECTRUM Operationen :
estRName,
relRName : RTyp Akey1 AKey2 -> RTyp

## 2. Die Existenz-Anfragen:
is_inRName : AKey1 RTyp -> bool
is_inRName : Akey2 RTyp -> bool
is_inRName : Akey1 Akey2 RTyp -> bool

## 3. Die "check"-Funktionen:
RName_1_1, RName_1_N,
RName_N_1, RName_N_N : RTyp -> bool

SEMANTICS
VARS
akey1 : AKey1,
akey2 : Akey2,
akey1×akey2 : Akey1×Akey2
relationship : RTyp
PROGRAMS

## 1. Die SPECTRUM Operationen :
estRName(relationship, akey1, akey2) <- ins(relationship, {akey1/akey2});
relRName(relationship, akey1, akey2) <- del(relationship, {akey1/akey2});

## 2. Die Existenz-Anfragen:
is_in(akey1, relationship) <- is_in(akey1, domain(relationship));
is_in(akey2, relationship) <- is_in(akey2, codomain(relationship));
is_in(akey1, akey2, relationship) <- is_in(relationship, {akey1/akey2});

## 3. Die "check"-Funktionen:
RName_N_N(relationship) <- true;

relation-Name_1_N(relationship)
   <- IF (relationship=empty-RTyp)
         THEN true
         ELSE LET akey1×akey2 : choose(relationship) IN
               (#(reachable_from(get_fst(akey1×akey2), relationship))=succ(null))
```

```
                and
                RName_1_N(del(relationship, akey1×akey2))
                TEL;
        FI;

relation-Name_N_1(relationship)
   <- IF (relationship=empty-RTyp)
          THEN true
          ELSE LET akey1×akey2 : choose(relationship) IN
                (#(reachable(get_snd(akey1×akey2), relationship))=succ(null))
                and
                RName_N_1(del(relationship, akey1×akey2))
                TEL;
        FI;

relation-Name_1_1(relationship)
   <- RName_N_1(relationship) and RName_1_N(relationship)


ENDCREATE
ENDEXTENDED)
PARAMS

 (SORTS Akey1, AKey2)
 [SORTS RTyp
  OPNS estRName **,
          relRName **,
          RName_1_1 **,
          RName_1_N **,
          RName_N_1 **,
          RName_N_N **]
```

7.2 Anmerkungen

Obige Schablone ist leider in dem Sinn nicht ganz allgemeingültig, daß sie zur Spezifikation jeder Relationship eines E/R-Schema verwendet werden kann. Das Problem ist, daß die beiden importierten Sorten verschieden sein müssen, da sonst die durch Instanziierung entstehende Spezifikation kein korrekter OBSCURE Modul ist. Für diesen Spezialfall muß der Spezifizierer die aus der obigen Schablone resultierende Spezifikation anpassen, d.h. die doppelten Importe von Sorten und Operationen entfernen. Ebenso werden in einer solchen Spezifikation zwei Operationen **is_in** der gleichen Stelligkeit

spezifiziert. Auch da muß der Spezifizierer neue Namen für diese beiden Operationen einführen. Aus Gründen der Übersichtlichkeit, wurde in der Schablone auf diese Problematik keine Rücksicht genommen.

7.3 Beispiel

Zur Verdeutlichung wenden wir die Spezifikation MK_RELATIONSHIP auf das obige Beispiel an: Gegeben seien die zwei Sorten *AKeyPatient* und *AKeyArzt*. Die Spezifikation der binären relationship *behandelt* erfolgt folgendermaßen:

```
## Der Modul BEHANDELT
(MK_RELATIONSHIP (SORTS AKeyPatient, AKeyArzt)
                 [SORTS Behandelt,
                  OPNS estbehandelt **,
                       relbehandelt **,
                       behandelt_1_1 **,
                       behandelt_1_N **,
                       behandelt_N_1 **,
                       behandelt_N_N **]
```

7.4 Der Modul RELATIONSHIPS

Die Spezifikationen der einzelnen Relationships des E/R-Schemas werden im Modul RELATIONSHIPS mittels des OBSCURE Konstruktors PLUS miteinander kombiniert. Die Schablone für diesen Modul hat folgendes Aussehen, wobei $RName_1, \ldots, RName_k$ die Namen der benannten Relationshiptypen des E/R-Schemas sind.

```
## Der Modul RELATIONSHIPS
(INCLUDE RNAME1)
   PLUS
     ⋮
PLUS
(INCLUDE RNAMEk)
```

8 Ansammlung aller Relationships

In einem konkreten E/R-Schema gibt es im allgemeinen mehrere benannte Relationshiptypen. Jeder einzelne benannte Relationshiptyp wird durch eine Sorte spezifiziert, deren Spezifikation durch Anwendung der parametrisierten Spezifikation MK_RELATIONSHIP des vorherigen Abschnitts entstanden ist. Alle Relationships des konkreten E/R-Schemas müssen nun zusammengefaßt werden. Dies soll mittels einer, über den Sorten der benannten Relationshiptypen parametrisierten Spezifikation geschehen. Analog zur Spezifikation der Ansammlung aller Entity-Ansammlungen (siehe Seite 44), ist die Anzahl der in konkreten E/R-Schemata enthaltenen benannten Relationshiptypen nicht festgelegt. Daher geben wir hier eine Schablone an und lassen die Anzahl N der benannten Relationshiptypen offen. Die durch die Schablone repräsentierten parametrisierten Spezifikationen erhalten als import Parameter die Sorten der benannten Relationshiptypen und liefern als export Parameter die Sorte, die die Ansammlung aller Relationships spezifizieren soll.

8.1 Die Schablone

In der folgenden Schablone sind "RName$_i$" und "RTyp$_i$" jeweils die Namen und Typen der spezifizierten benannten Relationshiptypen. Zusätzlich wird noch eine Konstante spezifiziert, die die Ansammlung leerer Relationhips modelliert. Diese wird später zur Spezifikation der leeren Datenbank benötigt.

Der Modul DB_RELATIONSHIPS
(INCLUDE MK_N_TUPEL
(SORTS RTyp$_1$,
⋮
RTyp$_N$)
[SORTS DB_Relationships
OPNS set_RName$_N$ **,
get_RName$_N$ **,
⋮
set_RName$_1$ **,
get_RName$_1$ **,
{_/ . . . /_}])
EXTENDED BY
IMPORTS

```
 OPNS
   empty-RTyp_1 : RTyp_1,
                  ⋮
   empty-RTyp_N : RTyp_N
CREATE
  OPNS empty_DB_Relationships : -> DB_Relationships
  SEMANTICS
  PROGRAMS
  empty_DB_Relationships <- { empty-RTyp_1 /
                                          ⋮
                                empty-RTyp_N };
ENDCREATE
ENDEXTENDED
```

Obige Schablone wird im Anwendungs-Falle mit dem Namen der entsprechenden Tupel-Spezifikation, dem Namen der benannten Relationshiptypen instanziiert, sowie der Namen der exportierten Operationen. Die Instantiierung verläuft analog zur Instanziieerung der Schablone für Spezifikationen von Ansammlungen von Entity-Ansammlungen (siehe Seite 44). Aus diesem Grund wird an dieser Stelle auf ein Beispiel verzichtet. In dieser Schablone tauchen die gleichen Probleme wie bei der von Relationships auf (siehe Abschnitt 7.2 auf Seite 49). Auch für diese Schablone gelten die gleichen Bemerkungen wie für die der Relationships.

9 Die Datenbank

9.1 Die Grundspezifikation

In den vorherigen beiden Abschnitten über **Ansammlungen von Entity-Ansammlungen** auf Seite 44 und **Ansammlung aller Relationships** auf Seite 51, wurde ein Großteil der Spezifikation einer Datenbank vorweggenommen, so daß jetzt nur noch die erhaltene "Ansammlung von Entity-Ansammlungen" und "Ansammlung aller Relationships" zusammengefaßt werden müssen. Dies geschieht in einer Spezifikation, die die Standard Spezifikation MK_PAIR mit den importierten Sorten aufruft, welche gerade zum einen die Sorten der Ansammlung von Entity-Ansammlungen, und zum anderen die Sorte der Ansammlung aller Relationships sind.

```
## Der Modul RAW_DB
(INCLUDE MK_PAIR
                (SORTS DB_Entities,
                       DB_Relationships)
                [SORTS Datenbank
                 OPNS set_DB_Relationships **,
                      get_DB_Relationships **,
                      set_DB_Entities **,
                      get_DB_Entities **,
                      {-/-} **])
EXTENDED BY
IMPORTS
 OPNS
   empty_DB_Entities : -> DB_Entities,
   empty_DB_Relationships : -> DB_Relationships
CREATE
 OPNS empty_DB : -> DB
 SEMANTICS
 VARS
 PROGRAMS
 empty_DB <- { empty_DB_Entities
               empty_DB_Relationships };
ENDCREATE
ENDEXTENDED
```

9.2 Ein Beispiel

Als Grundlage für das Beispiel wählen wir das E/R-Schema des Ablaufs HK aus [Ben93], das aus 5 benannten Entitytypen mit Namen Arzt, Patient, HK_UW, HK_Daten und HK_Befund besteht, sowie den 6 benannten Relationshiptypen mit Namen *gehört_zu*, *untersuchung*, *befundung*, *ermittelt*, *ordnet_an* und *erstellt*. Das E/R-Schema sieht folgendermaßen aus:

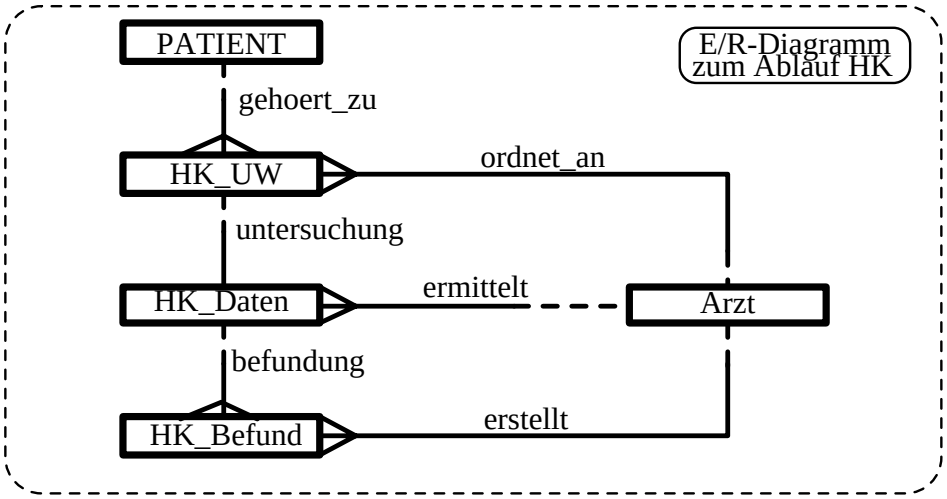


Abbildung8: Ein Beispiel E/R-Schema

In den vorherigen Abschnitten wurde beschrieben, wie die Sorten MListOfArzt, MListOfPatient, MListOfHK_UW, MListOfHK_Daten und MListOfHK_Befund sukzessive spezifiziert wurden. Darauf aufbauend wurde die Sorte DB_Entities spezifiziert. Auf der anderen Seite wurden die Relationships als Sorten mit entsprechenden Namen spezifiziert und darüber die Sorte DB_Relationships. Diese beiden Sorten, DB_Entities und DB_Relationships, werden in der Spezifikation DB, der Spezifikation MK_PAIR als Import-Parameter übergeben, welche dann die Sorte DB liefert.

```
## Der Modul RAW_DB
(INCLUDE MK_PAIR
                (SORTS DB_Entities,
                       DB_Relationships)
                [SORTS Datenbank
                 OPNS set_DB_Relationships **,
                      get_DB_Relationships **,
                      set_DB_Entities **,
                      get_DB_Entities **,
```

```
                        {-/-} **])
EXTENDED BY
IMPORTS
 OPNS
   empty_DB_Entities : -> DB_Entities,
   empty_DB_Relationships : -> DB_Relationships
CREATE
 OPNS empty_DB : -> Datenbank
 SEMANTICS
 PROGRAMS
 empty_DB <- { empty_DB_Entities /
               empty_DB_Relationships };
ENDCREATE
ENDEXTENDED
```

9.3 Die Gesamtspezifikation

Wie schon in der Einführung in Abschnitt 3 auf Seite 9 erwähnt, erfüllt die Komfortebene drei Funktionen: Dieser Teil der Spezifikation der Datenbank hat folgende Aufgabe: in ihr sollen bestimmte Operationen, die bis zu diesem Zeitpunkt der Spezifikation des E/R-Schemas auf Teilen der Datenbank operieren, auf Datenbankebene "hochgezogen" werden. Dies ist zum Beispiel der Fall bei einer Operation get*Patient* : Akey_Patient MListPatient -> Patient. Diese sollte eigentlich folgende Stelligkeit haben: get*Patient* : Akey_Patient Datenbank -> Patient. Aus diesem Grund spezifizieren wir die neue Operation get*Patient* mit Hilfe der alten.
Insgesamt müssen folgende Operationen eines konkreten E/R-Schemas "hochgezogen" werden:

1. Für alle benannten Entitytypen (EName_i, ETyp_i) des konkreten E/R-Schemas müssen die Operationen

   putEName_i : EName_i MListEName_i -> MListEName_i,
   delEName_i : EName_i MListEName_i-> MListEName_i,
   getEName_i : AKey_EName_i MListEName_i -> EName_i und
   updateEName_i : EName_i MListEName_i -> MListEName_i

   auf Datenbankebene "hochgezogen" werden.

2. Für alle benannten Relationshiptypen (RName_j, RTyp_j) des konkreten E/R-Schemas müssen die Operationen

estRName$_j$: RName$_j$ Akey_EName$_{j1}$ AKey_Name$_{j2}$ -> RName$_j$,
relRName$_j$: RName$_j$ Akey_EName$_{j1}$ AKey_Name$_{j2}$ -> RName$_j$,
RName$_j$_1_1 : RName$_j$ -> bool,
RName$_j$_1_N : RName$_j$ -> bool,
RName$_j$_N_1 : RName$_j$ -> bool und
RName$_j$_N_N : RName$_j$ -> bool

auf Datenbankebene "hochgezogen" werden.

Zusätzlich zu diesen "hochgezogenen" Operationen, müssen wieder "check"-Funktionen spezifiziert werden, die die zwingende Partizipation oder die ausschließende Partizipation von Entities an Relationships überprüfen, wobei unter zwingender Partizipation und ausschließender Partizipation folgendes gemeint ist:

- Unter einer Relationship die **zwingende Partizipation** von Entities eines bestimmten benannten Entitytyps BETyp verlangt, verstehen wir, daß jede Entity dieses benannten Entitytyps in dieser Relationship in Beziehung mit mindestens einer Entity eines anderen benannten Entitytyps steht, für den der Relatonship definiert ist.

- Unter zwei Relationships mit sich **ausschließender Partizipation** verstehen wir zwei Relationships , für die gilt, daß eine Entity nicht sowohl in der ersten, als auch in der zweiten in Beziehung mit anderen Entities steht.

Die Semantik dieser Operationen kann und wird in diesem Teil der Spezifikation nur angedeutet in Form einer Schablone. Die Namen dieser Operationen werden allerdings festgelegt. Die Spezifikation dieser Operationen für ein konkretes E/R-Schema wird im gleichen Modul erfolgen, wie die Spezifikationen der oben "hochzieh" Operationen .
Insgesamt entsteht aus obigen Ausführungen die folgende Schablone. Zur Vereinfachung verwenden wir hier eine mathematische Schreibweise mit ALL-Quantoren $\forall$, die aber keine OBSCURE-Syntax sind. Sie sind vielmehr eine abkürzende Schreibweise und sind intuitiv wie folgt zu verstehen: der Ausdruck

IMPORTS SORTS $\forall 1 \leq i \leq N$: EName$_i$

ist eine Abkürzung für

IMPORTS SORTS EName_1, EName_2, ..., EName_{N-1}, EName_N

Zur Schablone der Spezifikation DB:

Der Modul DB
INCLUDE RAW_DB
EXTENDED BY

((IMPORTS
SORTS
0. Die Datenbank:
Datenbank,
1. Die Liste der Sorten aller M benannten Entitytypen
des konkreten E/R-Schemas:
$\forall 1 \leq i \leq M$: EName_i,
2. Die Sorten der abstrakten Schlüssel:
$\forall 1 \leq i \leq M$: AKey_EName_i,
3. Die Liste der Sorten aller Monolisten über den benannten
Entitypen:
$\forall 1 \leq i \leq M$: MListOfEName_i,
4. Die Sorte, die die Ansammlung aller Entity-Ansammlungen
spezifiziert:
M_MListsOfEntities,
5. Die Liste der Sorten aller N benannten Relationshiptypen
des konkreten E/R-Schemas:
$\forall 1 \leq j \leq N$: RName_j,
6. Die Sorte, die die Ansammlung aller benannten
Relationshiptypen spezifiziert:
N_Relationships
OPNS
7. Die putEName_i Operationen:
$\forall 1 \leq i \leq M$: putEName_i : EName_i MlistOfEName_i -> MListOfEName_i
8. Die delEName_i Operationen:
$\forall 1 \leq i \leq M$: delEName_i : EName_i MlistOfEName_i -> MListOfEName_i
9. Die getEName_i Operationen:
$\forall 1 \leq i \leq M$: getEName_i : AKey_EName_i MlistOfEName_i -> EName_i
10. Die updateEName_i Operationen:
$\forall 1 \leq i \leq M$: updateEName_i : EName_i MlistOfEName_i -> MlistOfEName_i

11. Die estRName$_j$ Operationen:
$\forall 1 \leq i \leq N$:
relRName$_j$: RName$_j$ Akey_EName$_{j1}$ AKey_Name$_{j2}$ -> RName$_j$,
12. Die relRName$_j$ Operationen:
$\forall 1 \leq i \leq N$:
relRName$_j$: RName$_j$ Akey_EName$_{j1}$ AKey_Name$_{j2}$ -> RName$_j$,
13. Die RName$_j$_1_1 Operationen:
$\forall 1 \leq i \leq N$: RName$_j$_1_1 : RName$_j$ -> bool,
14. Die RName$_j$_1_N Operationen:
$\forall 1 \leq i \leq N$: RName$_j$_1_N : RName$_j$ -> bool,
15. Die RName$_j$_N_1 Operationen:
$\forall 1 \leq i \leq N$: RName$_j$_N_1 : RName$_j$ -> bool,
16. Die RName$_j$_N_N Operationen:
$\forall 1 \leq i \leq N$: RName$_j$_N_N : RName$_j$ -> bool
17. Die Zugriffsoperationen auf die Datenbank:
get-M_MListsOfEntities : Datenbank -> M_MListsOfEntities,
set-M_MListsOfEntities : M_MListsOfEntities Datenbank
-> M_MListsOfEntities,
get-N_Relationships : Datenbank -> N_Relationships,
set-N_Relationships : N_Relationships Datenbank -> N_Relationships
CREATE
OPNS
Die putEName$_i$ Operationen:
$\forall 1 \leq i \leq M$: putEName$_i$: EName$_i$ Datenbank -> Datenbank
Die delEName$_i$ Operationen:
$\forall 1 \leq i \leq M$: delEName$_i$: EName$_i$ Datenbank -> Datenbank
Die getEName$_i$ Operationen:
$\forall 1 \leq i \leq M$: getEName$_i$: AKey_EName$_i$ Datenbank -> EName$_i$
Die updateEName$_i$ Operationen:
$\forall 1 \leq i \leq M$: updateEName$_i$: EName$_i$ Datenbank -> Datenbank
Die estRName$_j$ Operationen:
$\forall 1 \leq i \leq N$:
relRName$_j$: Datenbank Akey_EName$_{j1}$ AKey_Name$_{j2}$ ->Datenbank,
Die relRName$_j$ Operationen:
$\forall 1 \leq i \leq N$:
relRName$_j$: Datenbank Akey_EName$_{j1}$ AKey_Name$_{j2}$ ->Datenbank,
Die RName$_j$_1_1 Operationen:
$\forall 1 \leq i \leq N$: RName$_j$_1_1 : Datenbank -> bool,
Die RName$_j$_1_N Operationen:
$\forall 1 \leq i \leq N$: RName$_j$_1_N : Datenbank -> bool,

Die RName$_j$_N_1 Operationen:
$\forall 1 \leq i \leq N$: RName$_j$_N_1 : Datenbank -> bool,
Die RName$_j$_N_N Operationen:
$\forall 1 \leq i \leq N$: RName$_j$_N_N : Datenbank -> bool
Für die Relationships RName$_k$ für die eine zwingende Partizipation
für die Entities eines bestimmten benannten Entitytyps mit
abstraktem Schlüssel der Sorte AKey_EName$_{j1}$ überprüft werden soll,
werden folgende Operationen spezifiziert:
Für alle obigen Relationships RName$_k$ und obigen Entities EName$_l$:
RName$_k$-EName$_l$-zw : Datenbank -> bool,
Für alle Paare von Relationships, für die eine ausschließende
Partizipation überprüft werden soll, werden folgende Operationen
spezifiziert:
Für alle obigen Relationships RName$_{k1}$ und RName$_{k2}$:
RName$_{k1}$-RName$_{k2}$-aus : Datenbank -> bool

```
SEMANTICS
VARS
db : Datenbank,
m_mlistsOfEntities : M_MListsOfEntities,
n_relationships : N_Relationships,
∀1 ≤ i ≤ M : mlistOfEName_i : MListOfEName_i,
∀1 ≤ i ≤ M : ename_i : EName_i,
∀1 ≤ i ≤ M : akey_ename_i : AKey_EName_i,
∀1 ≤ j ≤ N : rname_j : RName_j,
PROGRAMS
  ## Die putEName_i Operationen:
  ∀1 ≤ i ≤ M :
  putEName_i(ename_i, db)
      <- LET m_mlistsOfEntities : get-M_MlistsOfEntities(db)) IN
           LET mlistOfEName_i : getMListOfEName_i(m_mlistsOfEntities) IN
             set-M_MlistsOfEntities(setMListOfEName_i
                                                (putEName_i(ename_i,
                                                            mlistOfEName_i),
                                                 m_mlistsOfEntities),
                                    db)
           TEL;
         TEL;
  ## Die delEName_i Operationen:
  ∀1 ≤ i ≤ M :
  delEName_i(ename_i, db)
```

```
   <- LET m_mlistsOfEntities : get-M_MlistsOfEntities(db)) IN
        LET mlistOfEName_i : getMListOfEName_i(m_mlistsOfEntities) IN
          set-M_MlistsOfEntities(setMListOfEName_i
                                   (delEName_i(ename_i,
                                               mlistOfEName_i),
                                    m_mlistsOfEntities),
                                 db)
        TEL;
      TEL;
## Die getEName_i Operationen:
∀1 ≤ i ≤ M :
getEName_i(akey_ename_i, db)
   <- getEName_i(akey_ename_i, getMListOfEName_i
                                 (get-M_MlistsOfEntities(db)));
## Die updateEName_i Operationen:
∀1 ≤ i ≤ M :
updateEName_i(ename_i, db)
      <- LET m_mlistsOfEntities : get-M_MlistsOfEntities(db)) IN
           LET mlistOfEName_i : getMListOfEName_i(m_mlistsOfEntities) IN
             set-M_MlistsOfEntities(setMListOfEName_i
                                      (updateEName_i(ename_i,
                                                     mlistOfEName_i),
                                       m_mlistsOfEntities),
                                    db)
           TEL;
         TEL;
## Die estRName_j Operationen:
∀1 ≤ j ≤ N :
estRName_j(db, akey_ename_j1, akey_ename_j2)
   <- LET rname_j : estRName_j(getRName_j(get-N_Relationships(db)),
                               akey_ename_j1,
                               akey_ename_j2) IN
      set-N_Relationships(setRName_j(rname_j, get-N_Relationships(db)),
                          db)
      TEL;
## Die relRName_j Operationen:
∀1 ≤ i ≤ N :
relRName_j(db, akey_ename_j1, akey_ename_j2)
   <- LET rname_j : relRName_j(getRName_j(get-N_Relationships(db)),
                               akey_ename_j1,
```

akey_ename$_{j2}$) IN
set-N_Relationships(setRName$_j$(rname$_j$, get-N_Relationships(db)),
db)
TEL;
Die RName$_j$_1_1 Operationen:
$\forall 1 \leq i \leq N$:
RName$_j$_1_1(db) <- RName$_j$_1_1(getRName$_j$(get-N_Relationships(db)));
Die RName$_j$_1_N Operationen:
$\forall 1 \leq i \leq N$:
RName$_j$_1_N(db) <- RName$_j$_1_N(getRName$_j$(get-N_Relationships(db)));
Die RName$_j$_N_1 Operationen:
$\forall 1 \leq i \leq N$:
RName$_j$_N_1(db) <- RName$_j$_N_1(getRName$_j$(get-N_Relationships(db)));
Die RName$_j$_N_N Operationen:
$\forall 1 \leq i \leq N$:
RName$_j$_N_N(db) <- RName$_j$_N_N(getRName$_j$(get-N_Relationships(db)));
An dieser Stelle müssen noch die Definitionen der Operationen
RName$_k$-EName$_l$-zw und RName$_{k1}$-RName$_{k2}$-aus
angegeben werden.
ENDCREATE)

ENDEXTENDED

9.4 Der Modul DATENBANKEBENE

Die Spezifikation der gesamten Datenbankebene (siehe 5 auf Seite 13) erfolgt im Modul DATENBANKEBENE. In dieser werden die einzelnen Moduln ENTITIES, ENTITY_MONOLISTEN, DB_ENTITIES, RELATIONSHIPS, DB_RELATIONSHIPS und DB gemäs der Abbildung 5 mittels der OBSCURE Konstruktoren X_COMPOSE und PLUS zusammengesetzt. Die Spezifikation der ganzen Datenbankebene sieht folgendermaßen aus:
Der Modul DATENBANKEBENE

(INCLUDE ENTITIES)
X_COMPOSE
(((INCLUDE ENTITY_MONOLISTEN)
X_COMPOSE
(INCLUDE DB_ENTITIES))
PLUS

```
((INCLUDE RELATIONSHIPS)
        X_COMPOSE
 (INCLUDE DB_RELATIONSHIPS)))
        X_COMPOSE
        (INCLUDE DB)
```

Teil III
Die Komfort- und Schnittstellenebene

10 Die Komfort- und Schnittstellenebene

Dieser Modul hat zwei Funktionen: Die eine Funktion besteht in Spezifikation von Operationen, die in der SPECTRUM-Spezifikation nicht vorkommen, da sie dort nicht gebraucht werden. Typischerweise handelt es sich hierbei um Operationen, die zur Simulation des Existenzquantors von SPECTRUM verwendet werden müssen. Die andere Funktion dieser Spezifikation besteht darin, Operationen mittels des OBSCUREKonstruktors FORGET zu "vergessen" (Stichwort: Hiding), die nicht außerhalb der Datenmodellebene vorhanden sein dürfen. Hierfür wird sowohl eine schematische Form angegeben, als auch eine spezifische. Das schematische FORGET vergißt Sorten und Operationen, die in keinem Fall – d.h. in keiner konkreten Spezifikation eines E/R-Schemas – außerhalb der Datenmodellebene sichtbar sein dürfen. Das spezifische FORGET soll dem Spezifizierer die Möglichkeit geben, über das schematische FORGET hinaus noch Operationen zu vergessen, die er in seiner Spezifikation der Ablaufebene (siehe hierzu [Hec93]) nicht braucht.
Insgesamt ergibt sich aus den obigen Bemerkungen folgender Modul:
Der Modul KOMFORT_UND_SCHNITTSTELLE

((IMPORTS
  SORTS
    ## 0. Die Datenbank:
    DB,
    ## 1. Die Liste der Sorten aller M benannten Entitytypen
    ##   des konkreten E/R-Schemas:
    $\forall 1 \leq i \leq M$: EName$_i$,
    ## 2. Die Sorten der abstrakten Schlüssel:
    $\forall 1 \leq i \leq M$: AKey_EName$_i$,
    ## 3. Die Liste der Sorten aller Mono-Listen über den benannten
    ##   Entitypen:
    $\forall 1 \leq i \leq M$: MListOfEName$_i$,
    ## 4. Die Sorte, die die Ansammlung aller Entity-Ansammlungen
    ##   spezifiziert:
    M_MListsOfEntities,

5. Die Liste der Sorten aller N benannten Relationshiptypen
des konkreten E/R-Schemas:
$\forall 1 \leq j \leq N$: RName$_j$,
6. Die Sorte, die die Ansammlung aller benannten
Relationshiptypen spezifiziert:
N_Relationships

OPNS

7. Die is_inEName$_i$ Operationen:
$\forall 1 \leq i \leq M$: is_inEName$_i$: EName$_i$ MlistOfEName$_i$ -> bool
8. Die is_inRName$_j$ Operationen:
$\forall 1 \leq i \leq N$: is_inRName$_j$: Akey_EName$_{j1}$ RName$_j$ -> bool,
$\forall 1 \leq i \leq N$: is_inRName$_j$: Akey_EName$_{j2}$ RName$_j$ -> bool,
$\forall 1 \leq i \leq N$:
is_inRName$_j$: Akey_EName$_{j1}$ Akey_EName$_{j2}$ RName$_j$ -> bool,
9. Die Zugriffsoperationen auf die Datenbank:
get-M_MListsOfEntities : DB -> M_MListsOfEntities,
set-M_MListsOfEntities : M_MListsOfEntities DB -> M_MListsOfEntities,
get-N_Relationships : DB -> N_Relationships,
set-N_Relationships : N_Relationships DB -> N_Relationships

CREATE

OPNS
Die is_inEName$_i$ Operationen:
$\forall 1 \leq i \leq M$: is_inEName$_i$: EName$_i$ DB -> bool
Die is_inRName$_j$ Operationen:
$\forall 1 \leq i \leq N$: is_inRName$_j$: Akey_EName$_{j1}$ DB -> bool,
$\forall 1 \leq i \leq N$: is_inRName$_j$: Akey_EName$_{j2}$ DB -> bool,
$\forall 1 \leq i \leq N$: is_inRName$_j$:
Akey_EName$_{j1}$ Akey_EName$_{j2}$ DB -> bool,

SEMANTICS

VARS

db : DB,
m_mlistsOfEntities : M_MListsOfEntities,
n_relationships : N_Relationships,
$\forall 1 \leq i \leq M$: mlistOfEName$_i$: MListOfEName$_i$,
$\forall 1 \leq i \leq M$: ename$_i$: EName$_i$,
$\forall 1 \leq i \leq M$: akey_ename$_i$: AKey_EName$_i$,
$\forall 1 \leq j \leq N$: rname$_j$: RName$_j$,

PROGRAMS

Die is_inEName$_i$ Operationen:
$\forall 1 \leq i \leq M$:

```
    is_inEName_i(ename_i, db)
      <- is_in(ename_i, getMListOfName_jget-M_MlistsOfEntities(db));
    ## Die is_inRName_j Operationen:
    ∀1 ≤ i ≤ N :
    is_inRName_j(akey_ename_j1, db)
      <- is_inRName_j(akey_ename_j1,
                      getRName_j(get-N_Relationships(db)));
    ∀1 ≤ i ≤ N :
    is_inRName_j(akey_ename_j2, db)
      <- is_inRName_j(akey_ename_j2,
                      getRName_j(get-N_Relationships(db)));
    ∀1 ≤ i ≤ N :
    is_inRName_j(akey_name_j1, akey_ename_j2, db)
      <- is_inRName_j(akey_ename_j1,
                      akey_ename_j2,
                      getRName_j(get-N_Relationships(db)));
  ENDCREATE)
  FORGET
  SORTS
  ## Die Sorten der Entity-Ansammlungen:
  ∀1 ≤ i ≤ M : M_MListOfEName_i,
  ## Die Sorten der Relationships:
  ∀1 ≤ j ≤ N : RName_j,
  ## Die Ansammlung aller Entity-Ansammlungen:
  M_MListsOfEntities,
  ## Die Ansammlung aller Relationships:
  N_Relationships
)
## Und zuletzt noch das spezielle FORGET:
FORGET
SORTS ...
OPNS ...
```

Teil IV
Anhang

11 Schlüsselattribute & konkrete Schlüssel

Wir wollen an dieser Stelle noch die formale Definition eines Schlüsselattributs und darauf aufbauend, die formale Definition eines konkreten Schlüssels angeben. Dafür verfahren wir, analog zur Definition eines Attributs und einer Entity, in folgenden Schritten:

1. Definition eines Schlüsselattributtyps
2. Definition des Domains eines Schlüsselattributtyps
3. Definition eines Schlüsselattributs
4. Definition des Typs eines konkreten Schlüssels
5. Definition des Domains eines Typs eines konkreten Schlüssels
6. Definition eines konkreten Schlüssels

11.1 Schlüsselattributtyp

Um ein Attribut als Schlüsselattribut auszeichnen zu können, brauchen wir eine Entity, in der dieses Attribut als Schlüsselattribut auftritt. Somit benötigen wir zur Definition eines Schlüsselattributtyps sowohl einen benannten Attributtyp, als auch einen benannten Entitytyp:

$$\textit{Schlüssel-ATyp} = ((AName, ATyp), (EName, ETyp))$$

wobei

1. $ETyp = ((AName_1, ATyp_1), \ldots, (AName_n, ATyp_n))$
2. und $\exists\ 1 \leq i \leq n \mid (AName_i, ATyp_i) = (AName, ATyp)$

11.2 Domain eines Schlüsselattributtyps

Der Domain eines Schlüsselattributtyps wird gegeben über eine Funktion D_{SA}, deren Semantik folgendermaßen definiert ist:

D_{SA}(Schlüssel-ATyp) = { (a, f) *mit* 1. a $\in$ D_A(ATyp)
2. f $\in$ D_E(ETyp)
3. f(AName) = a }

wobei *Schlüssel-ATyp* wie oben definiert.

11.3 Schlüsselattribut

Ein Schlüsselattribut ist dann ein Element aus dem Kreuzprodukt:

$$Attributnames \times \bigcup_{Schlüssel-ATyp} D_{SA}(Schlüssel\text{-}ATyp)$$

11.4 Typ eines konkreten Schlüssels

Die Definitionen des Typs eines konkreten Schlüssels, des Domains eines konkreten Schlüssels, sowie eines konkreten Schlüssels funktioniert analog zu den Definitionen von Entitytypen, Domains von Entitytypen und Entities. Im Unterschied zu Entities werden allerdings **Schlüsselattribute** anstelle von **Attributen** verwendet. Somit definiert sich der Typ eines konkreten Schlüssels wie folgt:
Typ-eines-konkreten-Schlüssels = $((Schlüssel\text{-}ATyp_1, \ldots, Schlüssel\text{-}ATyp_m))$
wobei
$\forall 1 \leq i \leq m$ *gilt* $Schlüssel\text{-}ATyp_i = ((AName_i, ATyp_i), (EName, ETyp))$

11.5 Domains eines Typs eines konkreten Schlüssels

Der Domain eines Typs eines konkreten Schlüssels definieren wir wie folgt mit Hilfe einer Funktion D_{KS}:
D_{KS}(Typ-eines-konkreten-Schlüssels) =
{ f: {$AName_1, \ldots, AName_n$}
$\rightarrow \bigcup_{Schlüssel-ATyp_i} D_{SA}(Schlüssel\text{-}ATyp_i)$
mit $f(AName_i) \in D_{SA}(Schlüssel\text{-}ATyp_i)$ }
wobei *Schlüssel-ATyp*$_i$ analog zu oben definiert ist.

11.6 Konkreter Schlüssel

Ein konkreter Schlüssel eines Typs eines konkreten Schlüssels ist dann ein Element aus der Menge

$$D_{KS}(Typ\text{-}eines\text{-}konkreten\text{-}Schlüssels).$$

Index

Literatur

[Aut93] S. Autexier. HDMS-A und OBSCURE in KORSO – Die funktionale Essenz von HDMS-A aus Sicht der algorithmischen Spezifikationsmethode. Teil 2: Schablonen zur Übersetzung eines E/R-Schemas in eine OBSCURE Spezifikation. Technical Report A/05/93, Universität des Saarlandes, Saarbrücken, December 1993.

[Ben93] C. Benzmüller. HDMS-A und OBSCURE in KORSO – Die funktionale Essenz von HDMS-A aus Sicht der algorithmischen Spezifikationsmethode. Teil 3: Die Spezifikation der atomaren Funktionen. Technical Report A/06/93, Universität des Saarlandes, Saarbrücken, December 1993.

[CHL94a] F. Cornelius, H. Hußmann, and M. Löwe. The KORSO Case Study for Software Engineering with Formal Methods: A Medical Information System. Technical Report 94-5, Technische Universität Berlin, February 1994.

[CHL94b] F. Cornelius, H. Hußmann, and M. Löwe. The KORSO Case Study for Software Engineering with Formal Methods: A Medical Information System. In Broy, M. and Jähnichen, S., editor, KORSO–*Abschlußband (noch offen)*. Springer LNCS, 1994. to appear, also published in an extended version as technical report no. 94-5, Technische Universität Berlin, February 1994.

[CKL93] F. Cornelius, M. Klar, and M. Löwe. Ein Fallbeispiel für KORSO: Ist–Analyse HDMS-A. Technical Report 93-28, Technische Universität Berlin, 1993.

[Con93] S. Conrad. Einbindung eines bestehenden Datenbanksystems in einen formalen Software–Entwicklungsprozeß — ein Beitrag zur HDMS-A–Fallstudie. In H.-D. Ehrich, editor, *Beiträge zu* KORSO–*und* TROLL *light–Fallstudien*, pages 1–14. Technische Universität Braunschweig, Informatik–Bericht 93–11, 1993.

[Dam93] W. Damm. KORSO–Applikationen — HDMS-A Teilprojekt Universität Oldenburg. Short description of ongoing work, February 1993.

[Fuc94] T. Fuchß. Translating E/R-diagrams into Consistent Database Specifications. Technical Report 2/94, Universität Karlsruhe, January 1994.

[GH93] M. Grosse and H. Hufschmidt. SOLL–Spezifikation aus Sicht der Sicherheit. Technical Report A/07/93, Universität des Saarlandes, Saarbrücken, December 1993.

[Hec93] R.A. Heckler. HDMS-A und OBSCURE in KORSO – Die funktionale Essenz von HDMS-A aus Sicht der algorithmischen Spezifikationsmethode. Teil 1: Einführung und Anmerkungen. Technical Report A/04/93, Universität des Saarlandes, Saarbrücken, December 1993.

[Het93] R. Hettler. Zur Übersetzung von E/R-Schemata nach SPECTRUM. Technical Report TUM-I9333, Technische Universität München, 1993.

[Hoh93] Uwe Hohenstein. *Formale Semantik eines erweiterten Entity–Relationship–Modells.* Teubner, 1993. ISBN 3-8154-2052-0.

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

[MP88] McMenamin and Palmer. *Strukturierte Systemanalyse.* London: Prentice Hall / München, Wien: Hanser, 1988. Übersetzung von Peter Hruschka.

[MZ94] M. Mehlich and W. Zhang. Specifying Interactive Components for Configuratiing Graphical User Interfaces. Technical Report 9401, Ludwig-Maximilians-Universität München, 1994.

[Nic93] F. Nickl. Ablaufspezifikation durch Datenflußmodellierung und stromverarbeitende Funktionen. Technical Report TUM-I9334, Technische Universität München, 1993.

[Ren94] K. Renzel. Formale Beschreibung von Sicherheitsaspekten für das Fallbeispiel HDMS-A. Technical Report 9402, Ludwig-Maximilians-Universität Munich, January 1994.

[Sch94] M. Schulte. Spezifikation und Verifikation von kommunizierenden Objekten in einem verteilten System. Master's thesis, University of Oldenburg, Computer Science Department, March 1994. (in German).

[SH94] M. Strecker and R. A. Heckler. Modifizierungsrahmen, Zwei–Ebenen–Konzept und Eintopf–Konzept — Erster Bericht der Rahmen–Gruppe — vormals "Erweiterbarkeitsgruppe". Technical Report at the Universities of Ulm and Saarbrücken, to appear, 1994.

[Shi94] H. Shi. Benutzerschnittstelle und -Interaktion für die HK-Untersuchung. Technical Report at the Universität Bremen, to appear, February 1994.

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

[Ste93] K. Stenzel. A Verified Access Control Model. Technical Report 26/93, Fakultät für Informatik, Universität Karlsruhe, Germany, December 1993.