HDMS–A und OBSCURE in KORSO

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

Teil 1: Einführung und Anmerkungen

Ramses A. Heckler

Technischer Bericht **A/04/93**

... vom Dezember '93

Ramses A. Heckler (Autor)

ramses@cs.uni–sb.de

unter Mitarbeit von

Serge Autexier und Christoph Benzmüller

und unter Beratung durch

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

HDMS

♥ Ā ♥

KORSO

Ramses A. Heckler

ramses@cs.uni–sb.de

HDMS–A **und** OBSCURE **in** KORSO

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

Teil 1: Einführung und Anmerkungen

Inhaltsverzeichnis

1 Einleitung

1.1 Das allgemeine Vorwort

1.1.1 Zu KorSo und Hdms

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 ([Huß93]) 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: [Hec93a, Aut93, Ben93, Fuc94, SH94].
Any of the cited reports can be obtained directly from the authors. Details about that can be found in [CHL94a, CHL94b].

1.1.2 Zu "Hdms–A und Obscure in KorSo"

Das Dokument "Hdms-A und Obscure in Korso – Die Funktionale Essenz von Hdms-A aus Sicht der algorithmischen Spezifikationsmethode" besteht aus drei einzelnen Berichten:

[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

- TEIL 1: Einführung und Anmerkungen,
- TEIL 2: Schablonen zur Übersetzung eines E/R–Schemas in eine OBSCURE–Spezifikation,
- TEIL 3: Die Spezifikation der atomaren Funktionen.

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 untereinander als auch mit anderen Veröffentlichungen der HDMS-A–Reihe inhaltlich eng verwoben.
Die Teile (2) ([Aut93]) und (3) ([Ben93]) sind relativ unabhängig voneinander, bleiben jedoch ohne [Hec93a] (Teil 1) unverständlich. Alle drei Teile setzen zumindest eine oberflächliche Lektüre der Arbeiten [Het93] und [SNM+93] der KORSO–Partner von der TU und der LMU München voraus.

Teil 2: E/R–Schemata und OBSCURE Teil 2 des Dokuments beschäftigt sich in Analogie zu [Het93] mit der Übersetzung von Entity/Relationship–Schemata in die Spezifikationssprache OBSCURE. Dies soll der Spezifikation der *Datenbank* für das zu entwerfende Patientenverwaltungs–System HDMS-A dienen. Sie erfolgt in zwei Schritten: zuerst wird die Datenbank mittels eines E/R–Schemas beschrieben und anschließend das E/R–Schema in eine OBSCURE–Spezifikation übersetzt.
Es sei ausdrücklich darauf hingewiesen, daß wir hier aussschließlich ein Übersetzungs–Schema bzw. –Verfahren beschreiben: dieses Verfahren wird im konkreten Fall der HDMS-A–Datenbank *nicht* angewendet. Ferner bestünde eine Anwendung des Verfahrens in einer zwar *schematischen*, nicht aber *automatischen* Übersetzung. Eine solche ist zwar denkbar – es ging uns jedoch weder um den Bau noch um die Beschreibung eines dazu benötigten Werkzeugs (eines Compilers).

Teil 3: Spezifikation von Arbeitsabläufen in OBSCURE Teil 3 dokumentiert in Analogie zu [SNM+93] eine – auf der (gedachten) Spezifikation der Datenbank aufbauende – OBSCURE–Spezifikation. Sie beantwortet die Frage, welche grundlegenden Operationen das Patientenverwaltungs–System den Benutzern zur Verfügung stellen muß – und zwar natürlich im Hinblick darauf, daß die Benutzer des Systems alle für den Betrieb des Krankenhauses[5] benötigten Aktionen im Zusammenhang mit der Verwaltung

[5] das Deutsche Herzzentrum Berlin (DHZB)

von Patientendaten durchführen können. Die grundlegenden Operationen werden auch "*elementare Transaktionen*" genannt.

Zu diesem Zwecke wurden in der "IST–Analyse" typische Arbeitsabläufe im DHZB identifiziert und beschrieben (vgl. [CKL93]). Die IST–Analyse (bzw. deren Resultat, die IST–Spezifikation) wurde mit äußerster Sorgfalt von KORSO–Partnern aus Berlin erarbeitet und dokumentiert: sie bildet nicht nur die Grundlage für alle anderen Arbeiten im HDMS-A–Rahmen, sondern gewährt darüberhinaus auch interessante Einblicke in den Betrieb von Krankenhäusern.
In den SOLL–Spezifikationen ist ein Arbeitsablauf (kurz einfach "Ablauf") nichts anderes als die Aneinanderreihung der oben erwähnten elementaren Transaktionen. Eine der Haupt–Aufgaben bestand gerade darin, aus den Abläufen der IST–Spezifikation das wesentliche "herauszufiltern" und daraus die Abläufe der SOLL–Spezifikation zu kreieren. Dies haben vor allem die Münchner KORSO–Partner in den Arbeiten [SNM+93] und [Nic93] getan.

Die OBSCURE–Spezifikation von Teil 3 ist entstanden in direkter Anlehnung an [SNM+93] und in Beschränkung auf die elementaren Transaktionen von genau einem der fünf dort in SPECTRUM spezifizierten Abläufe – nämlich dem Ablauf "Herzkatheter–Untersuchung" ("HK–Ablauf"). Warnung: spezifiziert wurden lediglich die elementaren Transaktionen dieses Ablaufs – *nicht* jedoch der Ablauf selbst im Sinne der Aneinanderreihung der elementaren Transaktionen wie etwa in [Nic93].

Teil 1: Einleitung und Anmerkungen Teil 1 enthält genau das, was der Titel vermuten läßt: eine ausführliche Einleitung zu den Spezifikationen von Datenbank und HK–Ablauf des HDMS-A–Systems in den beiden anderen Teilen, sowie Anmerkungen zu speziellen Problemen, deren Darstellung am Ort ihres Auftretens jeweils zu umfangreich gewesen wäre. Hier wird beispielsweise eingegangen auf Fragen wie:

- Was hat es mit den "abstrakten" Schlüsseln unseres Datenbank–Modells auf sich?
- Was verstehen wir unter E/R–Schemata und OBSCURE–Schablonen?
- Was behandeln wir im Sinne "statischer Integritätsbedingungen"?

Insbesondere zieht der letzte Abschnitt in Teil 1 ein Resümee der geleisteten Arbeit.

1.2 Das spezielle Vorwort

1.2.1 Inhalt und Gliederung

Worum geht es bei der "Funktionalen Essenz von HDMS–A aus Sicht der algorithmischen Spezifikationsmethode"?

1. Zunächst einmal um das (erwünschte) HDMS–A–System. Diesem liegt die Idee zugrunde, die bisher in unserem Krankenhaus[6] verwendeten *Patientenakten* durch elektronische Versionen zu ersetzen. Dabei denken wir uns alle aktuell im Krankenhaus vorhandenen Patientenakten zusammengefaßt in "der" großen *konventionellen Patientenakte* (kurz koPA). Analog werden die neuen, elektronischen Patientenakten zusammengefaßt zu "der" *elektronischen Patientenakte* (kurz ePA).

   Die koPA enthält in der Realität sowohl Papierdokumente, als auch Röntgenbilder oder gar –filme, als auch sonstige Dokumente verschiedenster Arten. Für unsere Zwecke genügt jedoch die Vorstellung von der koPA als einem großen Aktenordner voller Papierdokumente. Diese koPA soll nun ersetzt werden durch die ePA.

2. Vom HDMS–A–System interessiert uns "nur" dessen "*Funktionale Essenz*". Auf diesen Begriff werden wir im folgenden noch näher eingehen; er beruht auf einem Werk von McMenamin/Palmer ([MP88]) und wurde durch [Huß93] zur Behandlung des Fallbeispiels HDMS–A im KORSO–Rahmen eingeführt.

   Die Beschreibung der Funktionalen Essenz eines Systems entspricht der Beantwortung der Frage, was dieses System *unbedingt* leisten muß, damit es seiner Aufgabe gerecht werden kann. Dabei ignoriert man insbesondere alle Probleme, die mit der Gestaltung von Benutzungsoberflächen zusammenhängen – aber auch (und erst recht) programmiersprachen– oder maschinenspezifische Details. Darüberhinaus soll die Funktionale Essenz mehr *Inhalt* sein als *Form*: spezielle Daten*strukturen* und Designentscheidungen sollen vermieden werden – und wo sie unumgänglich sind, sollen sie lediglich der Beschreibung des *Was* dienen, nicht aber verbindlich sein für die Gestaltung (das *Wie*) des endgültigen Systems.

   Die Beschreibung der Funktionalen Essenz stellt das Ergebnis dessen dar, was im allgemeinen Vorwort mit "requirements analysis" bezeichnet wird.

[6] Wir sprechen im folgenden der Anschauung halber *nicht* vom DHZB, sondern von "unserem Krankenhaus"!

3. Die Funktionale Essenz von HDMS–A soll mit Hilfe einer formalen Spezifikationssprache beschrieben werden. In unserem Falle handelt es sich um die Sprache OBSCURE, die gemäß der Idee der algorithmischen Spezifikationsmethode entworfen wurde. Zur algorithmischen Spezifikationsmethode (zum "algorithmischen Spezifizieren") siehe [Loe87] und auch Abschnitt 6.1; zur Sprache OBSCURE siehe [LL93]. Die Spezifikation der Funktionalen Essenz entspricht der "requirements specification" aus dem allgemeinen Vorwort.

Die OBSCURE–Spezifikation der Funktionalen Essenz (bzw. eines Ausschnitts davon) wird in den Teilen 2 und 3 ([Aut93, Ben93]) des vorliegenden Berichts dokumentiert. Sie ist in enger Anlehnung an die Arbeiten [Het93] und [SNM+93] zur Spezifikation der Funktionalen Essenz in der Sprache SPECTRUM [BFG+93, Reg93] entstanden. Genauer sieht die Unterteilung unseres Berichts so aus:

1. Dieses Dokument ist Teil 1: "Einführung und Anmerkungen".

2. In [Aut93] findet sich Teil 2: "Schablonen zur Übersetzung eines E/R–Schemas in eine OBSCURE–Spezifikation" (Spezifikation des Datenmodells); dieser Teil entspricht [Het93].

3. In [Ben93] findet sich Teil 3: "Die Spezifikation der atomaren Funktionen" (Top–Level–Funktionen der HK–Untersuchung); dieser Teil entspricht einem Ausschnitt aus [SNM+93].

Gegenstand des vorliegenden ersten Teils unseres Berichts ist

1. Die Einführung in die Welt der Funktionalen Essenz des HDMS–A–Systems: damit beschäftigen sich neben der Einleitung die Abschnitte 2 und 4.

2. Die Einführung in die OBSCURE–Spezifikation sowie in das zu ihrer Erstellung gewählte Vorgehen: damit beschäftigen sich neben der Einleitung vor allem die Abschnitte 2 und 3 – aber auch die Abschnitte 4 und 5.

3. Schließlich Überlegungen zum Sinn der gesamten Arbeit im Abschnitt 6 ("Reflexion"); hier soll insbesondere auch die Angemessenheit der Sprache OBSCURE bzw. allgemeiner der algorithmischen Spezifikationsmethode untersucht werden.

Insbesondere wird in Abschnitt 2.4 sehr genau beschrieben, *wie* wir *welchen* Ausschnitt aus der Funktionalen Essenz des HDMS–A–Systems behandelt haben.

1.2.2 Der Begriff der Funktionalen Essenz

Was kann man mit der koPA machen? Oder besser: was können die in unserem Krankenhaus Beschäftigten mit der koPA machen?

- Sie können darin "herumblättern".
- Sie können darin lesen, schreiben und neue Dokumente einfügen.
- Sie können bzw. könnten sie aber auch einem "Sekretär" geben und diesen beispielsweise alle weiblichen Patienten heraussuchen lassen, die jünger als 23 Jahre sind.

Hier besteht ein fundamentaler Unterschied zur ePA:

- Mit der ePA kann man absolut *nichts* "direkt" tun; mit Aktionen aller Art muß stets "der Computer" beauftragt werden. Dies gilt nicht nur für das Heraussuchen bestimmter Patienten (oder ähnliche *Anfragen*), sondern eben auch für Aktionen
  - wie >> blättere auf die nächste Seite <<
  - oder >> trage hier den Namen "Bettina von Arnim" ein <<

  Natürlich ist dabei "der Computer" in Wirklichkeit die der Funktionalen Essenz entsprechende Software für das HDMS–A–System.

Noch genauer ist es ein ganz bestimmter Teil dieser Software, den wir im folgenden als den "*elektronischen Sekretär*" bezeichnen werden. Dazu ist anzumerken:

- Das gesamte HDMS–A–System beinhaltet also insbesondere zwei wichtige, fiktive (konzeptionelle) Objekte, die man sorgfältig auseinander halten sollte: die ePA und den elektronischen Sekretär.
- Benutzungsoberflächen versuchen üblicherweise, die Existenz des elektronischen Sekretärs bzw. vor allem die Abhängigkeit des Benutzers von diesem Sekretär zu verschleiern. Diese Tatsache ist für uns aus zwei Gründen völlig uninteressant: zum einen ändert die Verschleierung nichts an Existenz und Abhängigkeit. Zum andern wollen wir Benutzungsoberflächen ohnehin ignorieren: interessant ist, *was* man überhaupt zu dem elektronischen Sekretär sagen kann (welche Dienste er anbietet), nicht *wie* man das dann im einzelnen zu tun hat (per Mausklick oder per Kommandozeile).

Weiter oben wurde gesagt, daß die Beschreibung der Funktionalen Essenz eines Systems der Beantwortung der Frage entspreche, was dieses System *unbedingt* leisten muß, damit es seiner Aufgabe gerecht werden kann. Genauer formuliert, geht es um die Beantwortung zweier Fragen:

1. Was ist der Inhalt der ePA – bzw.: welche Informationen müssen in der ePA unbedingt abgelegt werden können, damit unser Krankenhaus seinen Aufgaben auch dann noch gerecht werden kann, wenn die koPA vollständig durch die ePA ersetzt wird?

2. Welche Dienste muß – zum selben Zwecke – der elektronische Sekretär unbedingt anbieten?

Für die ePA und die Dienste des elektronischen Sekretärs liefert [MP88] zwei allgemein verwendbare Begriffe: die ePA ist der "*essentielle Speicher*" unseres Systems; die Dienste des elektronischen Sekretärs sind die "*essentiellen Aktivitäten*".

1.2.3 Die ePA als essentieller Speicher

Versteht man die ePA im Sinne eines essentiellen Speichers, dann interessiert uns ausschließlich ihr *Inhalt*: *was* soll sie beinhalten können? Uninteressant ist dagegen ihre *Form*: *wie* speichert sie ihre Inhalte? Eigentlich würden wir dann gerne bei der Beschreibung der ePA auf die Angabe ihrer Struktur völlig verzichten: sie soll kein "beschrifteter Baum" sein, keine "Liste" und keine "relationale Datenbank". Sie soll übrigens *auch* keine Ansammlung von getypten Mengen und Relationen sein, wie das unsere Spezifikation durch die Verwendung eines E/R–Schemas vorgaukelt.
Aber: es gibt keinen Inhalt ohne Form. Bestenfalls könnten wir "den Inhalt (eines Speichers) schlechthin" definieren als die Kongruenzklasse aller gleichwertigen Speicher mit demselben Inhalt. Das ist aber unerheblich; denn auch dann kann der Inhalt (bzw. die möglichen Inhalt*e*) eines Speichers nur mit Hilfe mindestens einer konkreten Form beschrieben werden. Der Inhalt eines Buches kann auch mit Hilfe einer Pergamentrolle vermittelt werden oder mit Hilfe eines Films oder eines dreidimensionalen Objekts – in jedem Falle aber braucht man mindestens eine dieser Formen.
Die Form der ePA ist die eines E/R–Schemas (vgl. 3.1); dafür gibt es folgende Gründe:

- Die Struktur der ePA wird dadurch relativ einfach; im wesentlichen gibt es die Teilstrukturen "Menge von Entities", "Entity" und "Relation(ship)". Aufwendigere Strukturen wären überflüssig! Denn, wie gesehen, interessiert uns nicht die *Struktur* der ePA, sondern ihr *Inhalt*. Insbesondere ist diese Struktur *nicht* verbindlich für das endgültige Softwareprodukt.

- Die Verwendung von E/R–Schemata hat sich in der (industriellen) Praxis bewährt. Besonders interessant daran ist, daß sie einen leichteren und "feinstufigeren" Übergang ermöglichen von der Phase einer mehr informalen Systemanalyse zu der Phase des formalen Spezifizierens der Anforderungen an ein System. Siehe dazu auch [Huß93, NW93, Hus93].

Die *Beschreibung* eines essentiellen Speichers nennen wir "*Datenmodell*" und daher den entsprechenden OBSCURE–Modul DATENMODELL. Dieser Modul bildet zusammen mit dem Modul ATOMAR zur Spezifikation der Dienste des elektronischen Sekretärs die gesamte OBSCURE–Spezifikation zur Funktionalen Essenz von HDMS–A. Man beachte, daß diese beiden Moduln in völlig unterschiedlicher Art und Weise erstellt wurden (bzw. erstellt werden sollen): der Modul ATOMAR "direkt von Hand", der Modul DATENMODELL aber "indirekt" mit Hilfe der Übersetzung eines E/R–Schemas nach OBSCURE (vgl. Abschnitt 3).
Randbemerkung: beim formalen Spezifizieren wird die Struktur von Objekten (Trägern) einer bestimmten Sorte (z.B. der Sorte *Db* unserer ePA) gegeben durch die darauf zulässigen Operationen. Aufgrund der von DATENMODELL zur Verfügung gestellten Operationen *verhalten* sich die Träger der Sorte *Db* tatsächlich so wie Mengen von Entities und Relationships. Betrachtet man jedoch ausschließlich die vom Modul ATOMAR kreierten Operationen (die essentiellen Aktivitäten, auch "elementare Transaktionen" bzw. kurz eTa's genannt), dann verhalten sich die Träger dergleichen Sorte *Db* noch mehr wie "black boxes", von deren inneren Strukturen man nichts weiß.

1.2.4 Die Dienste des elektronischen Sekretärs als essentielle Aktivitäten

Die Frage nach den Diensten des elektronischen Sekretärs lautete im Sinne der Funktionalen Essenz:

- Welche Dienste muß der elektronische Sekretär unbedingt anbieten, damit unser Krankenhaus seinen Aufgaben auch dann noch gerecht werden kann, wenn die koPA vollständig durch die ePA ersetzt wird?

Diese Frage ist weniger harmlos, als es vielleicht auf den ersten Blick erscheinen mag: ihre Beantwortung läßt die ePA in den Hintergrund rücken, dafür den elektronischen Sekretär mit tyrannischen Vollmachten in den Vordergrund ... und völlig außerhalb spielen die Beschäftigten unseres Krankenhauses dann eine Rolle, die im Prinzip auch von "elektronischen Agenten" (von Prozessen) übernommen werden könnte. Zur Verdeutlichung dieser Tatsache wollen wir uns zunächst das Beispiel dreier zusammenhängender essentieller Aktivitäten betrachten. Essentielle Aktivitäten heißen in unserem Rahmen *elementare Transaktionen* (kurz eTa's). Uns interessieren die drei eTa's

1. $\textit{überweisen} : Db \times PatId \times ArztId \rightarrow Db \times PatId$
2. $init\_HKP : Db \times PatId \times ArztId \rightarrow Db \times HkId$
3. $HK\_Untersuchung : Db \times HK\_Daten \rightarrow Db \times HkId$

sowie eine sogenannte *Umweltfunktion*, nämlich

4. $med\_hku : HkId \rightarrow HK\_Daten$

Die formale Spezifikation dieser Funktionen kann man finden in [SNM+93] in den Abschnitten "Arzt–Ablauf" und "Herzkatheter–Untersuchung". Was bedeuten sie?

1. Die eTa *überweisen* entspricht einer Aufforderung eines Arztes A_1 an unseren elektronischen Sekretär: dieser soll in der ePA (Argument *Db*!) eine neue Herzkatheter–Überweisung (kurz "HK–Überweisung") anlegen. Eine solche Überweisung enthält ihren Erstellungszeitpunkt, einen Kommentar des Arztes A_1 sowie ihre eigene Identifizierungsnummer (in unserem Krankenhaus sind alle Formulare ordentlich durchnumeriert). Darüberhinaus muß der Sekretär in der ePA natürlich vermerken, auf welchen Patienten sich die HK–Überweisung bezieht und von welchem Arzt sie erstellt wird. Um diesen Vermerk vornehmen zu können, erhält er vom Arzt A_1 dessen Identifizierungsnummer (Argument *ArztId*!) sowie auch die des Patienten (Argument *PatId*!). Die Funktion *überweisen* liefert zum einen die solchermaßen modifizierte Datenbank (Zielsorte *Db*!) und zum anderen die Identifizierungsnummer des Patienten (Zielsorte *PatId*). Letzteres bedeutet, daß der elektronische Sekretär diese Nummer weiterreicht an ...

2. ... die eTa *init_HKP* – genauer: fordert ein weiterer Arzt A_2 den Sekretär zur essentiellen Aktivität *init_HKP* auf, so braucht A_2 die Identifizierungsnummer des Patienten nicht mehr anzugeben: der Sekretär kennt sie ja schon. Er übergibt ihm lediglich seine eigene Identifizierungsnummer (Argument *ArztId*!) und der Sekretär modifiziert erneut

die ePA (Argument *Db*!). Dieses Mal legt er ein neues Dokument mit sogenannten "HK–Daten" an; ein Teil dieser Daten ergibt sich erst durch die eigentliche HK–Untersuchung des Patienten und ist beim Anlegen des neuen Dokuments noch nicht bekannt: das Dokument bleibt an den entsprechenden Stellen vorläufig leer. Die Identifizierungsnummer des HK–Daten–Dokuments (Zielsorte *HkId*!) übergibt der Sekretär ...

3. ... der "Umweltfunktion" *med_hku* – diese Funktion ist *keine* eTa: sie modelliert *nicht* einen Dienst des elektronischen Sekretärs, sondern einen (medizinischen) Vorgang *außerhalb* des HDMS–A–Systems, nämlich die HK–Untersuchung. Von dieser interessiert uns lediglich, daß sie bzgl. eines ganz bestimmten HK–Daten–Dokuments erfolgt (Argument *HkId*!) und daß sie letztendlich die darin noch fehlenden Daten liefert (Zielsorte *HK_Daten*!).

4. Schließlich übergibt irgendein Beschäftigter unseres Krankenhauses die nunmehr kompletten HK–Daten dem elektronischen Sekretär und fordert ihn zur essentiellen Aktivität *HK_Untersuchung* auf, durch die das HK–Daten–Dokument in der ePA entsprechend ausgefüllt (aktualisiert) wird.

Nun wird das weiter oben zum Verhältnis zwischen elektronischem Sekretär, ePA und Mensch gesagte klarer:

1. Die ePA ist in den Hintergrund getreten: die Ärzte A_1 und A_2 "greifen" *nicht* nach der ePA, sie "blättern" *nicht* darin "herum", sie fertigen *nicht* ein "Blatt" mit HK–Daten an und "heften" dieses dann in der ePA an der richtigen Stelle "ab". Vielmehr müssen sie aus einer begrenzten Anzahl "essentieller Aktivitäten" die eine oder andere auswählen und werden dabei durchaus nicht unbedingt von der Intuition geleitet, *mit* bzw. *in* einer ePA zu arbeiten. Im besten Falle verstehen sie diese als *Datenbank* – mit einer für Laien nicht durchschaubaren Struktur – die (hoffentlich!) nicht nur Dienste der oben beschriebenen Art anbietet, sondern auch umfangreiche Anfragemöglichkeiten.

2. Der elektronische Sekretär steht im Vordergrund: er ist es, der Art und Anzahl der angebotenen Dienste bestimmt und – mehr noch – der die Tätigkeiten der Menschen in unserem Krankenhaus vorzugeben beginnt. Man betrachte dazu noch einmal den oben bschriebenen Übergang von der eTa *überweisen* zur eTa *init_HKP*: im lauffähigen

HDMS–A–System müßte nicht *eine* HK–Überweisung angelegt und weitergereicht werden, sondern eine "Flut" derselben (daher die Verwendung von "Strömen" in [Nic93]!). Folglich übergibt der elektronische Sekretär die Identifizierungsnummer der durch *überweisen* angelegten HK–Überweisung nicht *sofort* an die eTa *init_HKP*. Stattdessen benutzt er sozusagen einen Merkzettel mit den Nummern aller angelegten aber noch nicht (an *init_HKP*) weitergereichten HK–Überweisungen. Aufgrund dieses Merkzettels kann und wird er nun den bzw. die betreffenden Ärzte (in unserem Beispiel A_2) solange tyrannisieren, bis diese die anstehenden HK–Überweisungen (mittels *init_HKP*) abgearbeitet haben. Art, "Taktung" und Reihenfolge der menschlichen Arbeit beginnen sich nach dem elektronischen Sekretär zu richten.

3. Der Mensch schließlich ist vom Standpunkt des HDMS–A–Systems überflüssig bzw. ersetzbar. Die oben beschriebenen essentiellen Tätigkeiten wurden nicht deshalb in der angegebenen Art identifiziert und spezifiziert, weil sie den *menschlichen* Beschäftigten in unserem Krankenhaus dienen sollen, sondern weil sie reibungslose Arbeitsabläufe ermöglichen sollen. Die eTa *init_HKP* könnte – im Zuge automatisierter Arbeitsabläufe – statt von einem Arzt ebenso gut von irgendwelchen Rechnerprozessen ("elektronischen Agenten") ausgelöst werden.

1.2.5 Fazit

Wir hoffen, in der Einleitung klar gemacht zu haben:

- Die Idee des HDMS–A–Systems als eines Softwareprodukts zur Ersetzung der konventionellen Patientenakte koPA durch die elektronische Patientenakte ePA.

- Den Unterschied zwischen dieser ePA und dem elektronischen Sekretär sowie auch die *Bedeutung* dieser beiden fiktiven Objekte.

- Die Idee von der Funkionalen Essenz des HDMS–A–Systems als Beantwortung der Fragen nach

  - der Art des essentiellen Speichers – also der ePA,
  - den essentiellen Aktivitäten – also den Diensten des elektronischen Sekretärs (auch "elementare Transaktionen" genannt).

- Den Unterschied in der Behandlung der Beantwortung dieser beiden Fragen:

  - Der OBSCURE–Modul DATENMODELL zur Spezifikation der ePA wird indirekt erstellt mit Hilfe der Übersetzung eines E/R–Schemas nach OBSCURE.
  - Der OBSCURE–Modul ATOMAR zur Spezifikation der eTa's wird "direkt von Hand" erstellt – er baut auf DATENMODELL auf.

- Schließlich die Auswirkungen der Idee der Funktionalen Essenz auf das endgültige Softwareprodukt: im Vordergrund steht der elektronische Sekretär, (nur noch) im Hintergrund die ePA und außerhalb der (ersetzbare!) Mensch.

2 Struktur und Modularisierung

(Struktur der Funktionalen Essenz und Modularisierung der OBSCURE–Spezifikation)

2.1 Die Grobstruktur

Gemäß [Huß93] (S.10 ff.) umfaßt die "Funktionale Essenz" eines Systems (vgl. [MP88]) stets sogenannte "Fachgebiete" sowie ein "Datenmodell". Die Funktionale Essenz des HDMS–A–Systems hat dementsprechend zwei wesentliche Bestandteile:

- Eine *Datenmodellebene*; diese umfaßt alle Aspekte, die mit der Speicherung und Verwaltung der vom HDMS–A–System benötigten Informationen zusammenhängen. Solche Informationen sind beispielsweise patientenbezogene Daten, Beziehungen zwischen solchen Daten oder auch Anordnungen von Ärzten bzgl. bestimmter Untersuchungen. Diese Ebene entspricht der Frage nach dem Aussehen der ePA aus der Einleitung!

- Eine *Ablaufebene*; diese beschäftigt sich mit den fünf für HDMS–A ausgewählten Abläufen (Aufnahme, Arzt– und Behandlungs–Ablauf, Labor–Ablauf, Herzkatheter–Untersuchung und Entlassung: vgl. [CKL93]). Die Abläufe von HDMS–A entsprechen den "Fachgebieten" aus [Huß93].

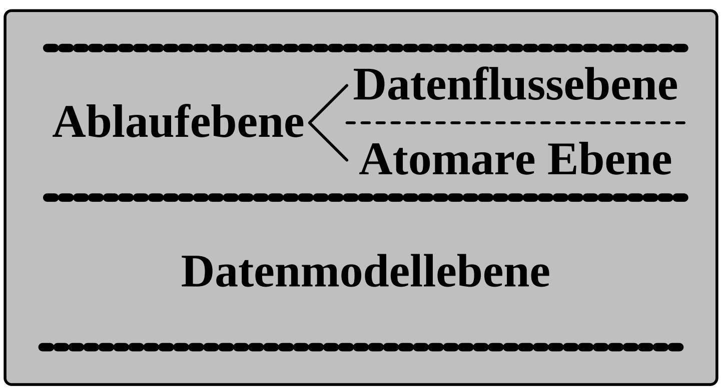
Die Ablaufebene hat selbst wiederum zwei wesentliche Bestandteile:

- Eine *Atomare Ebene*; diese umfaßt alle Aktionen, die in den fünf oben genannten HDMS–A–Abläufen von Relevanz sind. Die Aktionen seien im folgenden "atomare Funktionen" genannt (vgl. dazu 4.1). Die Atomare Ebene entspricht dem oberen Rechteck im grau unterlegten Anteil der Abbildung 8 in [Huß93]. Eine atomare Funktion ist entweder eine *Umweltfunktion* oder eine *elementare Transaktion* (eTa) (vgl. Einleitung). Diese Ebene entspricht der Frage nach den Diensten des elektronischen Sekretärs aus der Einleitung!

- Einer *Datenflußebene*; diese entspricht dem rechten, weißen Anteil in jener Abbildung 8. Die Datenflußebene baut auf der Atomaren Ebene auf und behandelt die Verknüpfung der atomaren Funktionen zu (den fünf) komplexen Abläufen. Eine einführende Beschreibung der Datenflußebene (und ihrer Spezifikation) findet sich in [Huß93]; eine ausführliche Darstellung liefert [Nic93]. Die hier zu dokumentierende OBSCURE–Spezifikation berücksichtigt die Datenflußebene *nicht*.

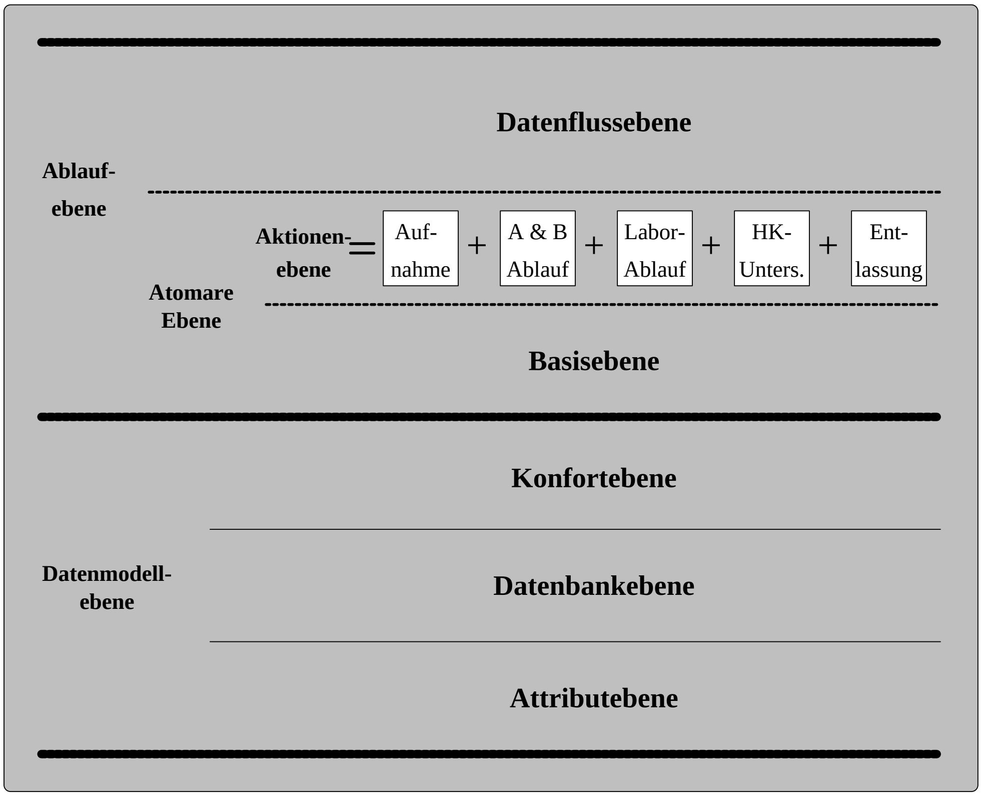
  Was mit "Verknüpfung" zu komplexen Abläufen gemeint ist, wird in der Einleitung anschaulich beschrieben an Hand des Bespiels der atomaren Funktionen *überweisen*, *init_HKP*, *HK_Untersuchung* und *med_hku*.

Man denke sich also die Funktionale Essenz des HDMS–A–Systems folgendermaßen strukturiert:



2.2 Verfeinerung und Beschränkung

Bereits im Hinblick auf die OBSCURE–Spezifikation verfeinern wir obige Struktur:



Dazu einige Erläuterungen:

- Die Atomare Ebene zerfällt in die Basisebene und die Aktionenebene; erstere umfaßt Aspekte, die zwar HDMS–A-spezifisch sind, nicht aber spezifisch für einen bestimmten Ablauf. Letzere umfaßt gerade die bereits erwähnten atomaren Funktionen der fünf Abläufe. Details finden sich in [Ben93].

- Die Bestandteile der Datenmodellebene beschäftigen sich mit ...

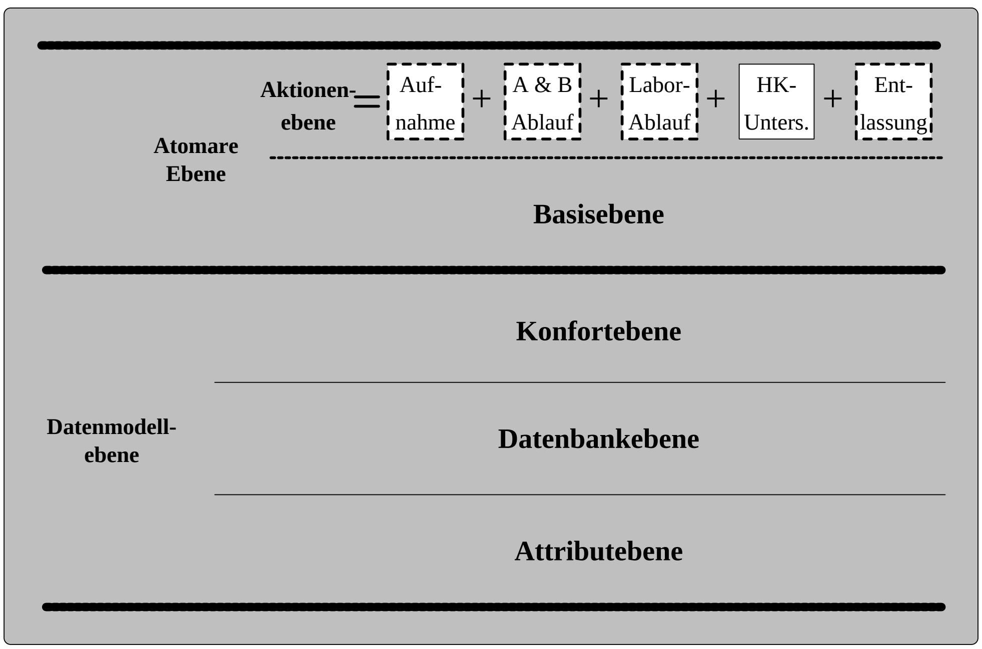
  1. ... Domains wie z.B. dem der "Namen" (Sorte *Name*), dem der "Geburtsdaten" (Sorte *GebOrt*) oder dem der "Entlassungsdiagnosen" (Sorte *EntlassungsDiagnose*); die Werte sogenannter "Attribute" werden aus diesen Domains kommen, weswegen letztere zur "*Attributebene*" gehören. Der Begriff des Attributs ist ein Konzept von E/R–Modellen (zu letzteren siehe 3.1).
  2. ... dem gesamten Speicher des HDMS–A–Systems, der sogenannten "Datenbank" – und zwar natürlich in der "*Datenbankebene*". Wir hätten die Datenbank auch "ePA" nennen können – in der

Einleitung wurde jedoch dargelegt, daß sie derart “in den Hintergrund” tritt, daß sie nicht als eine *Patientenakte* empfunden wird, sondern eher als eine “black box” mit unbekanntem und auch für den Laien unverstehbarem Innenleben.

3. … besonders “komfortablen” Zugriffsmöglichkeiten auf eben jene Datenbank; diese gehören zur “*Komfortebene*”.

Details zur Datenmodellebene und ihren Unterebenen finden sich in [Aut93].

Die Obscure–Spezifikation beschränkt sich in zweifacher Hinsicht auf einen Ausschnitt dieser Gesamtstruktur: zum einen behandelt sie nur die atomaren Funktionen von *einem* der fünf Abläufe, nämlich dem der HK–Untersuchung. Zum andern ignoriert sie die Datenflußebene, so daß auch von der Ablaufebene im folgenden nicht mehr explizit die Rede sein muß:



Die gestrichelten Boxen symbolisieren die von der Obscure–Spezifikation *ignorierten* atomaren Funktionen der Abläufe Aufnahme, Arzt– und Behandlungs–Ablauf, Labor–Ablauf und Entlassung. Dieses Bild spiegelt bereits die (grobe) Modularisierung der Obscure–Spezifikation wider!

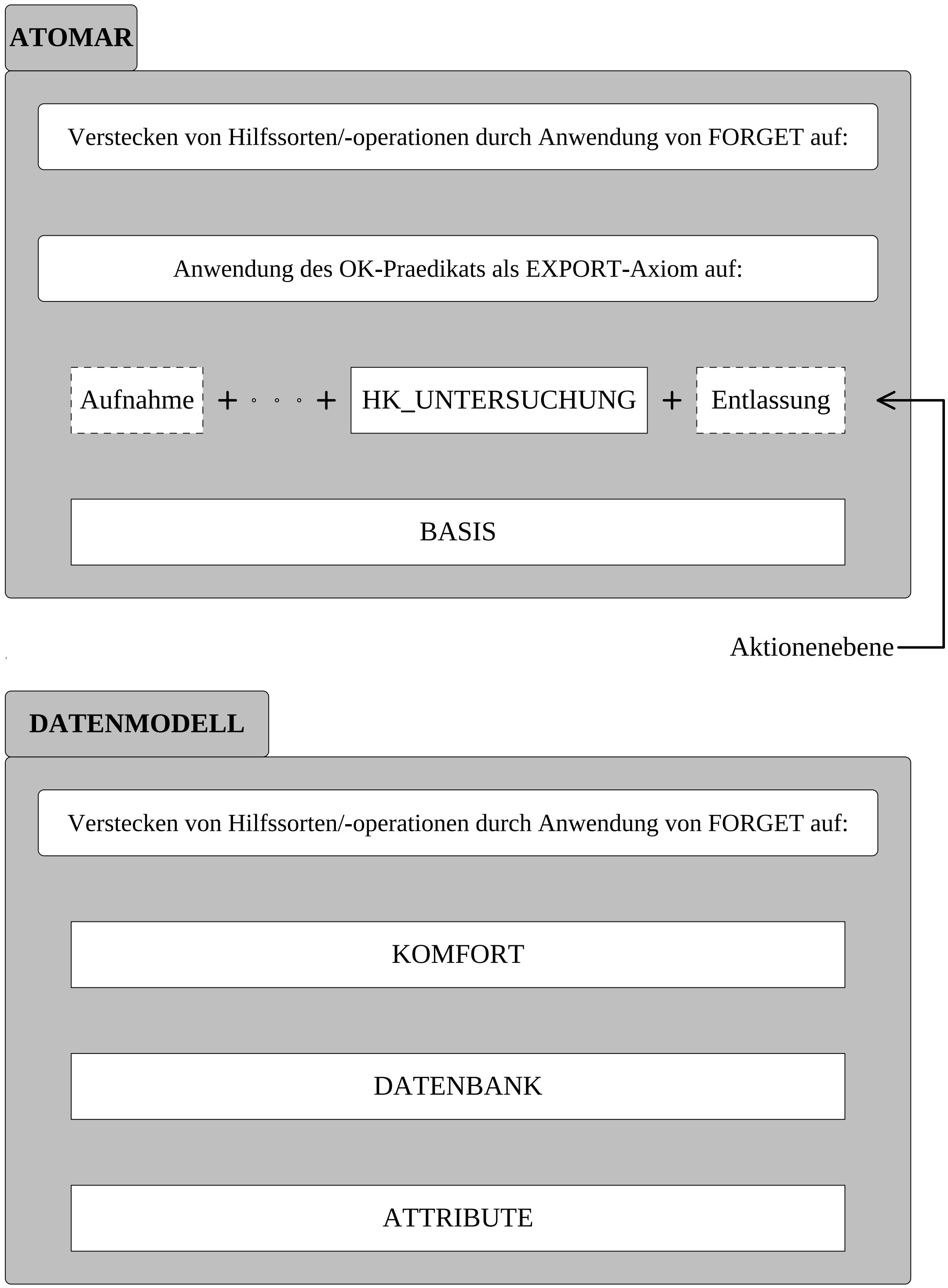
2.3 Die Modularisierung der OBSCURE–Spezifikation

2.3.1 Die Modularisierung im Bild

Die oben gegebene Struktur der Funktionalen Essenz muß lediglich um zwei Aspekte erweitert werden (ein wichtiges Export–Axiom und zwei FORGET–Konstruktionen), um die grundlegende Modularisierung der OBSCURE–Spezifikation zu erhalten. Diese ist für das Verständnis der Spezifikation extrem wichtig – nicht zuletzt auch im Sinne einer Orientierungshilfe für die Dokumente [Hec93a, Aut93, Ben93] und den darin enthaltenen OBSCURE–Quelltext. Es ist dagegen nicht von Nöten, bereits an dieser Stelle Details der einzelnen Ebenen zu kennen (etwa den Sinn des Export–Axioms). Die beiden neuen Aspekte sind:

- Je eine FORGET–Konstruktion, die Hilfssorten/–operationen der Datenmodellebene bzw. der Atomaren Ebene vor der "Außenwelt" versteckt.
- Ein EXPORT–Axiom, das eine wichtige (und zu beweisende!) Zusicherung über die von der Atomaren Ebene gelieferten elementaren Transaktionen macht. Diese Zusicherung hängt mit dem sogenannten "OK–Prädikat" zur Formulierung "statischer Integritätsbedingungen" zusammen (vgl. auch 4.4).

In der folgenden Visualisierung der Spezifikations–Modularisierung sind alle in Großbuchstaben geschriebenen Namen bereits Namen von OBSCURE-Moduln:


ATOMAR
Verstecken von Hilfssorten/-operationen durch Anwendung von FORGET auf:
Anwendung des OK-Praedikats als EXPORT-Axiom auf:
Aufnahme + ◦ ◦ ◦ + HK_UNTERSUCHUNG + Entlassung
BASIS
Aktionenebene
DATENMODELL
Verstecken von Hilfssorten/-operationen durch Anwendung von FORGET auf:
KOMFORT
DATENBANK
ATTRIBUTE

2.3.2 Die Modularisierung im Quelltext

Um dem Leser einen ersten Eindruck von der OBSCURE–Spezifikation zu vermitteln, werden nun die Moduln ATOMAR und DATENMODELL in korrekter OBSCURE–Syntax angegeben. Allerdings müßten noch die von spitzen Klammern umgebenen Stellen (< ... >) durch korrekten Quelltext ersetzt werden (siehe [Aut93, Ben93]). Die Modularisierung entspricht genau der obigen Visualisierung! Doppelteppiche ("##") leiten Kommentare ein.

1. Etwa zum Zwecke des Rapid Prototyping könnte man die Moduln ATOMAR und DATENMODELL zu einem Modul TEST zusammensetzen:

```
## Module TEST is

( (INCLUDE ATOMAR)
    X_COMPOSE
  (INCLUDE DATENMODELL)
)
```

2. Der Modul ATOMAR ist definiert durch:

```
## Module ATOMAR is

(
  ( (INCLUDE HK_UNTERSUCHUNG)
          ## (Die anderen Abläufe werden ignoriert!)
      X_COMPOSE
    (INCLUDE BASIS)
  )

  <Anwendung des OK–Prädikats als EXPORT–Axiom>

  <FORGET–Konstruktion>
)
```

3. Der Modul DATENMODELL ist definiert durch:

```
## Module DATENMODELL is

(
 ( (INCLUDE KOMFORT)
     X_COMPOSE
   (INCLUDE DATENBANK)
     X_COMPOSE
   (INCLUDE ATTRIBUTE)
 )

 <Spezifische FORGET–Konstruktion>
)
```

2.3.3 Wichtige Schnittstellen

Wie bereits erwähnt, wurden die Datenmodellebene und die Atomare Ebene methodisch auf völlig unterschiedliche Art und Weise behandelt: erstere mit Hilfe der Übersetzung von E/R–Schemata in OBSCURE–Schablonen (vgl. 3.2), letztere dagegen durch "direkte" Erstellung der erwünschten OBSCURE–Spezifikation. Dementsprechend bilden die Moduln DATENMODELL und ATOMAR je eine "eigenständige", große Einheit, deren Schnittstellen möglichst genau definiert werden sollten:

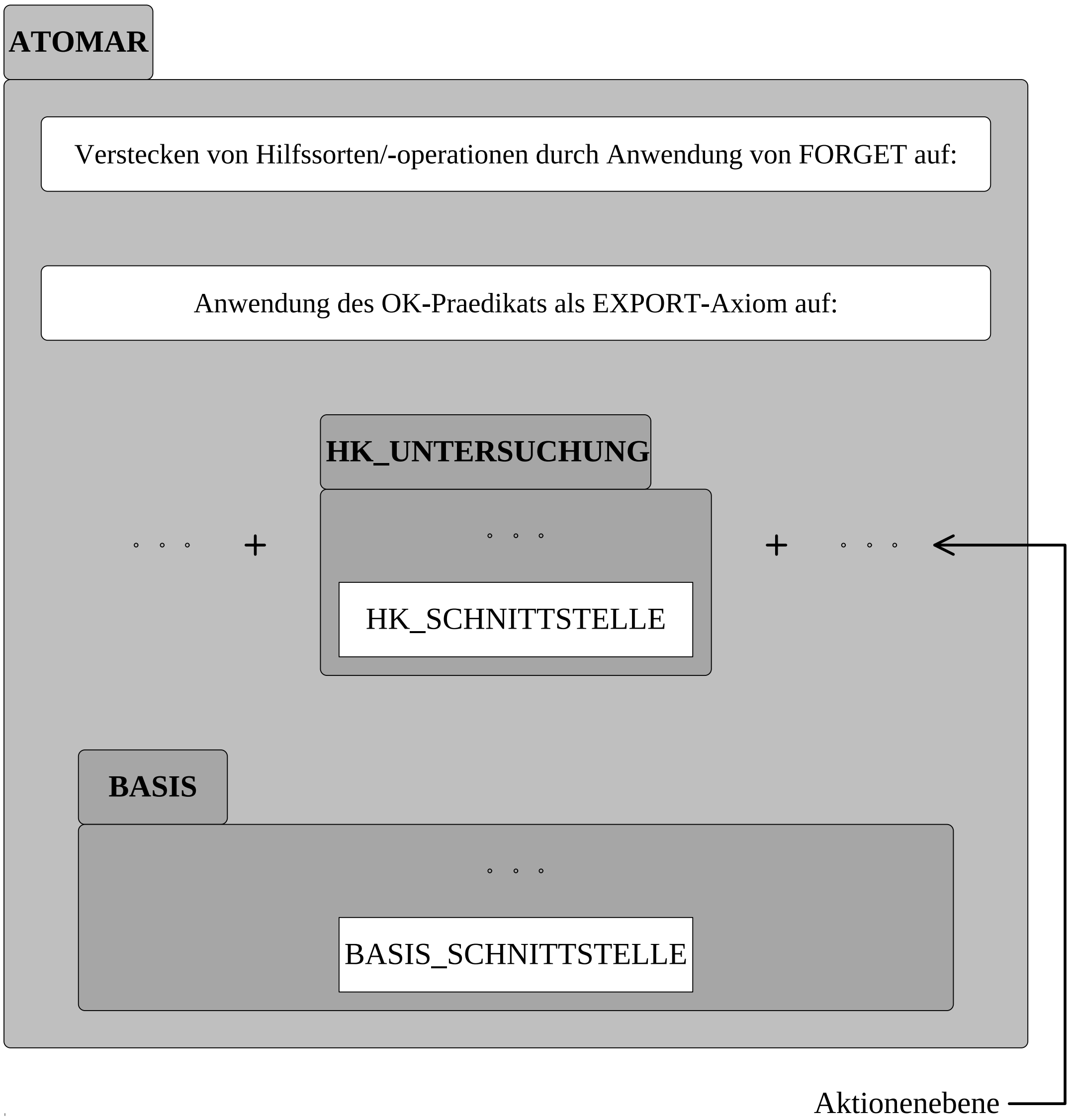
- Die Schnittstelle der Datenmodellebene besteht aus den von ihr der "Außenwelt" zur Verfügung gestellten Sorten und Operationen. Es wird mittels einer FORGET–Konstruktion (vgl. oben!) dafür gesorgt, daß keine "überflüssigen" Sorten/Operationen sichtbar bleiben.

- Die Schnittstelle der Atomaren Ebene besteht aus einer Import– und einer Export–Schnittstelle. Analog zur Datenmodellebene wird durch eine FORGET–Konstruktion dafür gesorgt, daß die Export–Schnittstelle nicht "zu groß" ist, daß also Hilfssorten und –operationen nach außen versteckt werden. Die Import–Schnittstelle wird definiert durch die beiden Moduln BASIS_SCHNITTSTELLE und HK_SCHNITTSTELLE (i.a. ergänzt um je einen Schnittstellenmodul für jeden der vier anderen Abläufe). Diese Moduln importieren jeweils genau[7] diejenigen Sorten und Operationen aus der Datenmodellebene,

[7] Das stimmt nicht ganz: der Modul HK_SCHNITTSTELLE importiert zusätzlich auch Operationen aus der Basisebene BASIS.

die an den entsprechenden Stellen gebraucht werden (also in der Basisebene bzw. im Modul HK_UNTERSUCHUNG).

Die Visualisierung der Modularisierung von ATOMAR sieht unter Berücksichtigung der Schnittstellenmoduln so aus:



2.4 Gesamtkonzept und Ausschnitt

An dieser Stelle soll die *tatsächlich* geleistete Arbeit eingeordnet werden in die *Gesamtheit* der zur Spezifikation der Funktionalen Essenz von HDMS–A anfallenden Arbeiten.

1. Die *gesamte* Arbeit besteht in der Spezifikation der Funktionalen Essenz in demjenigen Umfang, der von der ersten Abbildung in 2.2 repräsentiert wird.

2. Dagegen *fehlt* in der vorliegenden Arbeit (wie sie in [Hec93a, Aut93, Ben93] beschrieben wird):

   - Die Datenflußebene – also alles, was mit der Verknüpfung atomarer Funktionen zu komplexen Abläufen zu tun hat;
   - die atomaren Funktionen der Abläufe Aufnahme, Arzt– und Behandlungs–Ablauf, Labor–Ablauf und Entlassung – spezifiziert wurden lediglich die der HK–Untersuchung;
   - die Datenmodellebene als OBSCURE–Spezifikation – die gesamte Datenmodellebene wird nur in Form sogenannter OBSCURE–*Schablonen* beschrieben (vgl. 3.2). Die Instanziierung dieser Schablonen zu einem korrekten OBSCURE–Moduln DATENMODELL für den speziellen Fall der HDMS–A–Fallstudie wurde bisher *nicht* vorgenommen!

Der tatsächliche Umfang unserer Arbeiten geht also hervor aus der in Abschnitt 2.3 gegebenen Visualisierung der Moduln ATOMAR und DATENMODELL, wobei

- man sich vor Augen halten muß, daß das Datenmodell nur in Form universell wiederverwendbarer OBSCURE–Schablonen beschrieben wird, nicht aber als korrekter OBSCURE–Modul vorliegt.

2.5 Modularisierungsprinzipien

Für die Modularisierung "im großen" war vor allem die angestrebte Trennung zwischen "schematischem" und "spezifischem" ausschlaggebend – diese Trennung spiegelt sich insbesondere direkt wider in der Bildung der beiden "großen" Moduln DATENMODELL und ATOMAR. Siehe dazu auch 3.3.
Daneben wurde die Wahl der Modularisierung von folgenden Überlegungen beeinflußt:

1. Zum einen vom sogenannten "Schichtenmodell". Dieses geht davon aus, daß bereits mehrere "kleinere" Moduln existieren, die zu einem größeren Modul zusammengesetzt werden sollen. Der Zusammenbau erfolgt in naheliegender Art und Weise: alle Moduln, die keinerlei Sorten und Operationen importieren, bilden die erste – die "unterste" – "Schicht"

des Gesamtmoduls. Von den verbleibenden Moduln bilden nun all diejenigen, die ausschließlich Sorten und Operationen aus der untersten Schicht importieren, gerade die zweite Schicht. Innerhalb einer jeden Schicht werden die Moduln mit dem Sprachkonstrukt "PLUS" zusammengesetzt (meist visualisiert durch '+'); die Schichten selbst wiederum werden mit dem Sprachkonstrukt "X_COMPOSE" verknüpft (meist visualisiert durch 'o'). Das Verfahren wird natürlich solange fortgesetzt, bis alle Moduln in irgendeiner Schicht untergebracht sind. Man erhält Modularisierungen der Form

$$
\begin{array}{lclll}
(SP_{n,1} & + \; \ldots \; + & SP_{n,m(n)} &) & \#\#\ n\text{–te Schicht (mit } m(n) \text{ Moduln)} \\
 & \text{o} & & & \\
(SP_{n-1,1} & + \; \ldots \; + & SP_{n-1,m(n-1)} &) & \#\#\ (n-1)\text{–te Schicht} \\
 & \text{o} & & & \#\# \qquad \text{(mit } m(n-1) \text{ Moduln)} \\
 & \ldots & & & \\
 & \text{o} & & & \\
(SP_{1,1} & + \; \ldots \; + & SP_{1,m(1)} &) & \#\#\ 1\text{–te Schicht (mit } m(1) \text{ Moduln)}
\end{array}
$$

2. Zum anderen von der Idee, voneinander unabhängige Bestandteile einer Spezifikation zu identifizeren und dann so zusammen zu setzen, daß diese Unabhängigkeit zum Ausdruck kommt. Letzteres erreicht man durch die Verwendung des Sprachkonstruktes "PLUS": in der Spezifikation $SP \ := \ SP_1$ PLUS SP_2 importiert weder SP_1 Sorten/Operationen aus SP_2 noch umgekehrt. Offensichtlich widerspricht diese Idee der Philosophie des Schichtenmodells. Es könnte nämlich durchaus sein, daß z.B. innerhalb der Spezifikation

$$
\begin{array}{c}
(SP_{2,1} \ + SP_{2,2}) \\
\text{o} \\
(SP_{1,1} \ + SP_{1,2})
\end{array}
$$

keinerlei Importe "über Kreuz" erfolgen; gemäß der Unabhängigkeits–Idee müßte die Struktur dieser Spezifikation also so aussehen:

$$
\boxed{\begin{array}{c} SP_{2,1} \\ \text{o} \\ SP_{1,1} \end{array}} \; + \; \boxed{\begin{array}{c} SP_{2,2} \\ \text{o} \\ SP_{1,2} \end{array}}
$$

3. Schließlich von dem Gedanken, Schichtenmodell und Unabhängigkeits–Idee in geeigneter Art zu kombinieren: da das Schichtenmodell keinerlei Aussagen darüber macht, wie die kleineren Moduln (die $SP_{i,j}$) – und

deren Struktur – aussehen sollen, ist man sowohl bei der Wahl dieser Moduln frei (man kann kleinere oder größere Einheiten wählen) als auch bei der Gestaltung ihrer Struktur: man kann auf sie wiederum das Schichtenmodell anwenden oder aber die Unabhängigkeits–Idee oder aber sonstige Prinzipien. Offensichtlich verbleibt dem Spezifizierer sehr viel Freiheit beim Modularisieren.

Art und Ausmaß der Modularisierung "im kleinen" ist sicher willkürlich; beispielsweise muß man die Attributebene weder in genau vier Bestandteile zerlegen noch gerade in *diejenigen*, die von [Aut93] gegeben werden.

3 Methodik

3.1 E/R–Modelle und OBSCURE–Schablonen

Die beiden Moduln DATENMODELL und ATOMAR werden unterschiedlich behandelt. Während letzterer "direkt" auf der Basis der informalen Anforderungen in OBSCURE formuliert wird, erfolgt die Spezifikation des Datenmodells "auf dem Umweg" über ein E/R–Schema. Dieses Vorgehen entspricht dem der Münchner KORSO–Partner: siehe dazu die Berichte [Huß93], [Het93] und [SNM+93]. In [Hus93] wird es auch als "Translation Modelling" bezeichnet. Die Idee ist, die Datenbank des HDMS–A–Systems zuerst mit Hilfe eines E/R–Schemas zu modellieren und dieses dann in eine (formale) Spezifikationssprache zu übersetzen.
Bevor wir uns das näher ansehen, sollen die dafür benötigten Begriffe erklärt werden:

1. Ein *E/R–Modell* ist eine Beschreibungssprache. Weiter oben wurde gesagt, daß die Funktionale Essenz eines Systems zerfalle in ein Datenmodell und in Fachgebiete. Genauer hätte man sagen müssen: die Funktionale Essenz zerfällt in einen "essentiellen Speicher" und in "essentielle Aktivitäten" (vgl. Einleitung und [MP88]). Ihre *Beschreibung* besteht demzufolge aus einer Beschreibung des essentiellen Speichers – diese Beschreibung wird "Datenmodell" genannt – und einer Beschreibung der essentiellen Aktivitäten aus den einzelnen Fachgebieten.

   Mit Hilfe von E/R–Modellen (z.B. dem von uns gewählten) kann man essentielle Speicher beschreiben – ein Analogon zu diesem Begriff ist der der Programmiersprache (*nicht* der des Programms!): mit Hilfe von Programmiersprachen (z.B. PASCAL) kann man Algorithmen beschreiben.

2. Ein *E/R–Schema* ist eine in einer solchen Sprache gegebene Beschreibung – mithin ein Datenmodell (für einen essentiellen Speicher) und das Analogon zu einem PASCAL–Programm.

   Ein E/R–Schema für das HDMS–A–System (also ein Datenmodell für dessen essentiellen Speicher) findet sich in [SNM+93].

3. Typische Konzepte in E/R–Modellen sind das der "Entity" und das der "Relationship" – daher auch der Name "<u>E</u>ntity/<u>R</u>elationship–Modell " . Ein besonderes Merkmal des von uns bzw. von [Het93] gewählten E/R–Modells ist die Tatsache, daß nur *binäre* Relationships verwendet werden können.

   Man kann ein E/R–Modell auch sehen als die Gesamtheit aller von ihm zugelassenen E/R–Schemata (inklusive deren Semantik) – ebenso wie man etwa PASCAL auffassen kann als die Gesamtheit aller (syntaktisch korrekten) PASCAL–Programme (inklusive einer Semantik).

4. Eine OBSCURE–*Schablone* ist die Beschreibung mehrerer (meist unendlich vieler) OBSCURE–Spezifikationen; es handelt sich dabei um OBSCURE–Quelltext, der mit Ausdrücken einer (informalen!) Meta–Sprache durchsetzt ist. Typische Beispiele sind:

   - Eine Schablonen–Zeilenfolge der Form

     $$\begin{array}{lcl} \mathrm{next}E_1 & : & \mathit{SetOfKeyAttr} \rightarrow \mathit{KeyAttr} \\ & \vdots & \\ \mathrm{next}E_N & : & \mathit{SetOfKeyAttr} \rightarrow \mathit{KeyAttr} \end{array}$$

     Sie beschreibt unendlich viele OBSCURE–Zeilenfolgen; sie ist nämlich sowohl parametrisiert über der Anzahl der OBSCURE–Zeilen (Parameter N) als auch über die Namen von Entities (die Parameter E_i). Beschrieben wird also z.B. die Folge

     $$\begin{array}{lcl} \mathrm{nextPatient} & : & \mathit{SetOfKeyAttr} \rightarrow \mathit{KeyAttr} \\ \mathrm{nextArzt} & : & \mathit{SetOfKeyAttr} \rightarrow \mathit{KeyAttr} \end{array}$$

     … für N:=2, E_1:="Patient" und E_2:="Arzt".

   - Eine Zeile der Form

     $$\mathrm{SORTS}\ \forall 1 \leq i \leq N : \mathit{SetOfE_i}$$

     Sie ist eine Abkürzung für

     $$\mathrm{SORTS}\ \mathit{SetOfE_i}, \ldots, \mathit{SetOfE_N}$$

und beschreibt daher (analog zu oben) z.B. folgende Obscure–Zeile:

SORTS *SetOfPatient*, *SetOfArzt*

- Eine Schablone kann aus einem Obscure–Modul auch dadurch entstehen, daß die Definition einer Operation (unter dem Schlüsselwort PROGRAMS) einfach weggelassen wird – diese Operation wird dann nur noch durch die – ggfs. vorhandenen – EXPORT–Axiome definiert. Schablonen dieser Art ersetzen also die in Obscure selbst fehlende Möglichkeit losen Spezifizierens. Dieses Verfahren kommt zur Anwendung bei der Definition der sogenannten *next*–Funktionen: siehe Abschnitt 3.3.

5. Es sei noch einmal hervorgehoben, daß der Begriff der Obscure–Schablone *nicht* mathematisch rigoros definiert wird. Er wird hier ebenso informal eingeführt und benutzt wie die "SPECTRUM–Schablonen" in [Het93] (für die dort allerdings kein eigener Name eingeführt wird).

3.2 Modellieren durch Übersetzen

(Translation Modelling)

In [Het93] wird eine Funktion beschrieben, deren Argumentbereich gerade alle E/R–Schemata des dort gewählten E/R–Modells umfaßt – deren Wertebereich alle Spectrum–Spezifikationen umfaßt – und deren Graph einer (semantikerhaltenden) Übersetzung von E/R–Schemata in Spectrum–Spezifikationen entspricht.
Die Beschreibung dieser Funktion liefert insbesondere eine Beschreibung ihres Bildbereichs (der natürlich eine *echte* Teilmenge des Wertebereichs ist). Dieser Bildbereich (mit unendlich vielen Spectrum–Spezifikationen) wird beschrieben durch eine endliche Anzahl von Spectrum–Schablonen.
Ganz analog zu dieser Übersetzungsfunktion vom "Hdms–A–E/R–Modell" nach SPECTRUM wird in [Aut93] eine Funktion von diesem Modell nach Obscure beschrieben:

- neben *er2spec* : Hdms–A–E/R–Modell → Spectrum gibt es also auch
- *er2obs* : Hdms–A–E/R–Modell → Obscure.

Das zugrunde gelegte E/R–Modell ("Hdms–A–E/R–Modell") findet sich übrigens in [Het93] bzw. [Aut93]: es wird implizit durch die Beschreibung der jeweiligen Übersetzungs–Funktion gegeben.

Die Übersetzungs–Funktionen sind zur Verwendung im Sinne des "Translation–Modelling"–Ansatzes aus [Hus93] gedacht. Im Falle von HDMS–A bedeutet dies:

- Man gebe zunächst das Datenmodell für das HDMS–A–System in Gestalt eines E/R–Schemas. Dies geschieht in [SNM+93].
- Man wende auf dieses E/R–Schema die entsprechende Übersetzungs–Funktion an, wodurch man die Spezifikation DATENMODELL des Datenmodells in der Sprache SPECTRUM bzw. in der Sprache OBSCURE erhält. Explizit wurde das weder von den Münchner KORSO–Partnern gemacht noch von uns. Es gibt jedoch eine Arbeit der Karlsruher Partner in diese Richtung (vgl. [Fuc94]).
- Auf der Grundlage der Spezifikation DATENMODELL können nun beliebig komplexe Spezifikationen erstellt werden. In SPECTRUM wurden sowohl die atomaren Funktionen der Abläufe spezifiziert (siehe [Huß93]) als auch Aspekte der Datenflußebene (siehe [Nic93]). In OBSCURE wurde die Spezifikation ATOMAR der Atomaren Ebene auf der (gedachten) Spezifikation DATENMODELL aufbauend erstellt (siehe [Ben93]).

3.3 Aspekte der Übersetzung

3.3.1 Konzeptanleihe und Abstraktion

Die Verwendung der Übersetzungs–Funktionen *er2obs* und *er2spec* hat zwei wichtige Aspekte:

1. Den Aspekt der *Konzeptanleihe*:

   Die Theoretiker des Spezifizierens leihen sich hier ein wohlbewährtes Konzept von den Praktikern der Softwareerstellung: die Modellierung essentieller Speicher mit Hilfe von E/R–Modellen. Das ist *keine* Verlegenheitslösung und auch *kein* Eingeständnis eigener Unfähigkeit. Unabhängig vom konkreten Vorgehen im Falle des HDMS–A–Beispiels weist diese Idee der "Konzeptanleihe" den wohl einzigen möglichen Weg zur Anwendung formaler Spezifikationssprachen in der Praxis (vgl. dazu [Hus93] und [NW93]).

2. Den Aspekt der *Abstraktion*:

   Wie schon gesagt, werden die (unendlichen) Bildbereiche der Übersetzungs–Funktionen durch (endlich viele) OBSCURE– bzw. SPECTRUM–Schablonen beschrieben. Dadurch können viele Sorten/Operationen – wie z.B. die getE_i–Funktionen *schematisch* spezifiziert werden – also unabhängig von bestimmten Aspekten wie z.B:

- Wieviele Entitytypen und Relationshiptypen gibt es und wie sind sie benannt?
- Wieviele und welche Attribute haben die einzelnen Entities?
- Welche Grade haben die einzelnen Relationshiptypen?

Also kurz: "wie sieht das *spezifische* E/R–Schema aus?".

Die Abstraktion von spezifischen Aspekten eines E/R–Schemas führt zur Betrachtung seiner schematischen Anteile und dementsprechend beim Spezifizieren in OBSCURE bzw. SPECTRUM zur Trennung von spezifischen und schematischen Anteilen:

3.3.2 Schematisches und Spezifisches

In Ergänzung zu Abschnitt 2.5 kann nun das grundlegende Modularisierungsprinzip für die OBSCURE–Spezifikation der Funktionalen Essenz von HDMS–A formuliert werden:

Alles, was nicht für das HDMS–A–Beispiel (bzw. für die atomaren Funktionen seiner Abläufe) spezifisch ist, wird in dem *schematischen* Anteil der Gesamtspezifikation behandelt – d.h: in demjenigen Anteil, der durch die Übersetzung des E/R–Schemas nach OBSCURE entsteht.

Da der schematische Anteil der OBSCURE–Spezifikation den Namen "DATENMODELL" erhält, bedeutet dies, daß

- gilt: DATENMODELL := *er2obs*(HDMS–A–E/R–Schema) sowie, daß
- obiges Prinizip Auswirkungen hat auf die Definition der Übersetzungs–Funktion *er2obs*. Diese muß nämlich gerade derart erfolgen, daß im Bild $SP := er2obs(er\text{–}schema)$ eines jeden E/R–Schemas *er–schema* alle schematischen Anteile eines zu spezifizierenden Systems tatsächlich enthalten sind.

Dies soll nun an einem Beispiel studiert werden:

1. Man betrachte das Konzept der "Schlüssel"[8] (vgl. Abschnitt 5.1). Die Generierung neuer Schlüssel bei gegebenem Entitytyp E – etwa durch

[8] Ein Schlüssel für einen Entitytyp ist eine Teilmenge der Attribute dieses Entitytyps

eine Funktion new_keyE – ist von Entitytyp zu Entitytyp verschieden. Beispielsweise unterscheidet sie sich im HDMS–A–Falle mitunter schon in der Stelligkeit:

- $new\_keyPatient : Db \rightarrow KeyPatient$
- $new\_keyAufenthalt : Db \times PatId \rightarrow KeyAufenthalt$

Sie ist sogar von E/R–Schema zu E/R–Schema verschieden – und somit (zu) spezifisch, nicht schematisch behandelbar. Daher werden die entsprechenden Funktionen in der Atomaren Ebene (Modul ATOMAR) spezifiziert (*nicht* in der Datenmodellebene) und bei der Definition von *er2obs* ist von ihnen nicht die Rede.

2. Nun muß man sich aber die Frage stellen, mit Hilfe welcher Funktionen aus der Datenmodellebene sich möglichst viele verschiedene *new_key*–Funktionen der obigen Art realisieren lassen. In unserem Falle fiel die Wahl auf die sogenannten *next*–Funktionen, die für alle Attribute spezifiziert werden, die als Bestandteile von Schlüsseln im entsprechenden E/R–Schema vorkommen. Also z.B. für das Attribut *PatId* des Entitytyps *Patient*:

$$nextPatient^{9} : Set\_of\_PatId \rightarrow PatId$$

Die *next*–Funktionen haben die schematische Eigenschaft, "neue" Schlüssel zu liefern:

> Für jeden Entitytyp E und jedes Attribut $Attr$ dieses Entitytyps, das zu dessen Schlüssel gehört, gilt:
>
> Für jede Menge $attr\_set$ von Attributwerten der Sorte $Attr$ gilt:
>
> $nextE(attr\_set)$ ist nicht in $attr\_set$ enthalten.

Diese Formulierung ist nichts anderes als eine *lose Spezifikation* der *next*–Funktionen. Die schematische Behandlung der *next*–Funktionen erfordert hier also eine Abstraktion im Sinne von Losheit. Da dies in OBSCURE nicht möglich ist, wird es mit Hilfe von OBSCURE–Schablonen "simuliert".

Dementsprechend wird bei der Definition der *er2obs*–Funktion im Zusammenhang mit den *next*–Funktionen eine OBSCURE–Schablone benutzt, wie sie im Abschnitt 3.1 beschrieben wurde (siehe die letzte der dort beschriebenen Schablonen–Arten!).

[9] Der Name "*nextPatient*" mag verwundern, weil darin der Attributname "*PatId*" nicht vorkommt: siehe dazu [Aut93]

3.4 OBSCURE–spezifische Details

3.4.1 Konventionen

1. Moduln werden mit Großbuchstaben benannt, z.B: ATOMAR, DATENMODELL, HK_UNTERSUCHUNG, BASIS. Sortennamen beginnen mit einem Großbuchstaben, der Rest wird klein geschrieben, z.B: *Patient*, *Aufenthalt*, *Nat*. Allerdings werden "zusammengesetzte" Sortennamen aus naheliegenden Gründen wie in folgenden Beispielen geschrieben: *Set_of_Patient*, *Name_x_Geschlecht*. In Operationsnamen schließlich kommen normalerweise nur Kleinbuchstaben vor.

   Mit diesen Konventionen gerät man in Konflikt zu der Idee, die Benennungen in den SPECTRUM– und OBSCURE–Spezifikationen zur Funktionalen Essenz von HDMS–A möglichst einheitlich vorzunehmen. Der Einheitlichkeit wird dann der Vorzug gegeben!

2. Suffixe der Form "__L", "__D" (usw.) in Modulnamen haben eine rein "technische" Bedeutung und können ignoriert werden. Insbesondere denke man sich an der Stelle eines jeden "Dummys" SP__D den "richtigen" Modul SP (vgl. auch im nächsten Abschnitt über Dummies).

3. Die Konstruktoren von *Tupelsorten* sind Mixfix–Operationen folgender Bauart:

   - Eintupel : { _ }
   - Zweitupel : { _ / _ }
   - Dreitupel : { _ / _ / _ }

   ...

   Die entsprechenden Projektionsfunktionen sind *fst*, *snd*, *trd*. Tupelsorten und –operationen werden von den "Tupelmoduln" MK_N-TUPEL bzw. MK_PAIR generiert: siehe dazu 3.5.

3.4.2 Dummies

Es gibt in der OBSCURE–Spezifikation der Funktionalen Essenz von HDMS–A an einigen Stellen Moduln, die lediglich aus einer (mehr oder weniger langen) Liste importierter Sorten und/oder Operationen bestehen. Diese Sorten/Operationen werden unverändert exportiert. Solche (nur) scheinbar sinnlosen Moduln dienen unterschiedlichen Zwecken:

- Wenn ihr Name den Suffix "__D" hat, dann handelt es sich um sogenannte "*Dummy*"–Moduln. Die Verwendung von Dummies hat rein technische Gründe; zum Verständnis einer Spezifikation mit einem Dummy SP_D genügt es, sich an Stelle des Dummies den "richtigen" Modul SP vorzustellen. Einen richtigen Modul mit entsprechendem Namen gibt es in solchen Fällen *immer*.

- Der "Durchreiche–Modul" in der Attributebene der Datenmodellebene (siehe [Aut93]) soll dagegen eine (simple) Tatsache veranschaulichen: nämlich, daß an dieser Stelle bestimmte importierte Sorten "ohne weiteres" auch wieder exportiert werden können (im *Gegensatz* zu bestimmten anderen Sorten).

- Eine dritte Art der Verwendung liegt bei den *Schnittstellen*–Moduln BASIS_SCHNITTSTELLE und HK_SCHNITTSTELLE in der Atomaren Ebene vor: siehe dazu im Abschnitt 2.3 (unter "Wichtige Schnittstellen").

3.4.3 Standard–Moduln und Parameter

Zu *Standard–Moduln* sowie zu Bedeutung und Verwendung von *Parametern* in OBSCURE siehe im nächsten Abschnitt!

3.5 Standard–Moduln

Auf die Wiedergabe der Spezifikation von "Standard–Moduln" – wie z.B. dem Modul MK_MONOLIST zum Kreieren von Monolisten[10] – wird in den vorliegenden Berichten [Hec93a], [Aut93] und [Ben93] verzichtet. Stattdessen wird von diesen Moduln der jeweils relevante Auschnitt ihrer Import– und ihrer Export–Schnittstelle angegeben.
Erstere enthält alle Sorten und Operationen, die vom betreffenden Modul benutzt, nicht aber von ihm selbst kreiert werden. Die Export–Schnittstelle enthält alle Sorten und Operationen, die der Modul "nach außen" zur Verfügung stellt. Die Auszeichnung einer importierten oder exportierten Sorte/Operation als "*Parameter*" bedeutet lediglich, daß man bei Verwendung des Moduls diese Sorte/Operation "leichter" umbenennen kann: statt der üblichen Rename–Konstruktionen kann man vereinfachenden syntaktischen Zucker verwenden.
Der Zusatz

[10] Zum Begriff der "Monoliste" siehe Abschnitt 4.5!

```
PARAMS   (SORTS El
          OPNS  leq : El, El -> bool)
         [SORTS Set
          OPNS  empty_set : -> Set]
```

zu einem Modul SET, der insbesondere die Sorte *El* sowie die Operation *leq* importiert und die Sorte *Set* sowie die Operation *empty_set* exportiert, würde beispielsweise bedeuten, daß sich bei einer Verwendung von SET:

```
## Module SET_OF_NAT is
(INCLUDE SET)
  (SORTS Nat
   OPNS  kleiner_gleich : Nat, Nat -> bool)
  [SORTS Set_of_Nat
   OPNS  empty : -> Set_of_Nat]
```

die Moduln SET und SET_OF_NAT *ausschließlich* durch folgende Umbenennungen voneinander unterscheiden:

El	zu	*Nat*
leq	zu	*kleiner_gleich*
Set	zu	*Set_of_Nat*
empty_set	zu	*empty*

3.5.1 Die Tupelmoduln

Die bereits in Abschnitt 3.4 erwähnten Tupelsorten und –operationen werden in den "Tupelmoduln" MK_N–TUPEL bzw. MK_PAIR generiert. Dabei sehen die in den Schablonen der Datenmodellebene (vgl. [Aut93]) verwendeten Moduln MK_N–TUPEL ganz analog aus zum Spezialfall MK_PAIR = MK_2_TUPEL.
Der Modul MK_PAIR liefert zu je zwei Elementsorten *Sort_1* und *Sort_2* die Sorte

$$Pair := Sort\_1 \times Sort\_2$$

Import– und Export–Schnittstelle von MK_PAIR sehen so aus:

1. *Vererbt* (d.h. importiert und exportiert) werden von MK_PAIR folgende Sorten:

   SORTS

   Sort1 Sort2

2. *Kreiert* (d.h. nicht importiert, aber exportiert) werden von MK_PAIR folgende Sorten und Operationen:

   SORTS

   Pair

   OPNS

   { _ / _ } : Sort1 Sort2 –> Pair

   set_snd : Sort2 Pair –> Pair

   set_fst : Sort1 Pair –> Pair

   get_fst : Pair –> Sort1

   get_snd : Pair –> Sort2

3. *Import–Parameter* sind:

   SORTS

   Sort2 Sort1

4. *Export–Parameter* sind:

   SORTS

   Pair

   OPNS

   set_snd : Sort2 Pair –> Pair

   set_fst : Sort1 Pair –> Pair

   get_snd : Pair –> Sort2

   get_fst : Pair –> Sort1

   { _ / _ } : Sort1 Sort2 –> Pair

3.5.2 Der Standard–Modul MK_MONOLIST

Der Modul MK_MONOLIST liefert *Monolisten* der Sorte *MLst*, d.h. Listen ohne doppelte Vorkommen von Elementen. Ausführliche Anmerkungen dazu, warum wir statt *Mengen* den Datentyp *Monoliste* verwenden, finden sich in Abschnitt 4.5.

1. *Vererbt* (d.h. importiert und exportiert) werden von MK_MONLIST folgende Sorten:

   SORTS

   Nat El

2. *Kreiert* (d.h. nicht importiert, aber exportiert) werden von MK_MONLIST folgende Sorten und Operationen:

   SORTS

   MLst

   OPNS

   empty_mlst : –> MLst

   ins : El MLst –> MLst

   _ u _ : MLst MLst –> MLst

   # : MLst –> Nat

   del : El MLst –> MLst

   _ n _ : MLst MLst –> MLst

   _ - _ : MLst MLst –> MLst

   _ is_in _ : El MLst –> bool

   _ c _ : MLst MLst –> bool

   _ = _ : MLst MLst –> bool

   tail : MLst –> MLst

   head : MLst –> El

   choose : MLst –> El ## anderer Name für head !

   last : MLst –> El

   allbutlast : MLst –> MLst

3. *Import–Parameter* sind:

   SORTS

   El

4. *Export–Parameter* sind:

   SORTS

   MLst

   OPNS

   empty_mlst : –> MLst

3.5.3 Der Standard–Modul MK_BINARY_RELATION

Der Modul MK_BINARY_RELATION liefert "binäre Relationen" der Sorte *Binary_Relation*. Eine solche binäre Relation auf zwei Elementsorten *Sort_1* und *Sort_2* ist nichts anderes als eine (endliche) Monoliste von Tupeln aus *Sort_1* $\times$ *Sort_2*.

1. *Vererbt* (d.h. importiert und exportiert) werden von MK_BINARY_RELATION folgende Sorten und Operationen:

   SORTS

   Sort1 Sort2 Nat

   OPNS

   succ : Nat –> Nat

   null : –> Nat

2. *Kreiert* (d.h. nicht importiert, aber exportiert) werden von MK_BINARY_RELATION folgende Sorten und Operationen:

   SORTS

   Binary_Relation

   Pair MLstSort1 MLstSort2

   OPNS

   empty_Binary_Relation : –> Binary_Relation

   eBR : –> Binary_Relation

   { _ / _ } : Sort1 Sort2 –> Pair

   get_fst : Pair –> Sort1

   get_snd : Pair –> Sort2

   # : Binary_Relation –> Nat

   _ leq _ : Pair Pair –> bool

   _ is_in _ : Pair Binary_Relation –> bool

   _ c _ : Binary_Relation Binary_Relation –> bool

   _ = _ : Binary_Relation Binary_Relation –> bool

   _ = _ : Pair Pair –> bool

   _ u _ : Binary_Relation Binary_Relation –> Binary_Relation

   ins : Pair Binary_Relation –> Binary_Relation del : Pair Binary_Relation –> Binary_Relation

_ n _ : Binary_Relation Binary_Relation –> Binary_Relation

_ - _ : Binary_Relation Binary_Relation –> Binary_Relation

choose : Binary_Relation –> Pair

set_snd : Sort2 Pair –> Pair

set_fst : Sort1 Pair –> Pair

domain : Binary_Relation –> MLstSort1,

codomain : Binary_Relation –> MLstSort2,

reachable_from : Sort1 Binary_Relation –> MLstSort2,

reachable : Sort2 Binary_Relation –> MLstSort1,

isomorph : Binary_Relation Binary_Relation –> bool,

3. *Import–Parameter* sind:

   SORTS

   Sort1 Sort2

4. *Export–Parameter* sind:

   SORTS

   Binary_Relation

   OPNS

   empty_Binary_Relation : –> Binary_Relation

4 Kleinere Anmerkungen

4.1 Wichtige Begriffe aus der HDMS–A–Welt

1. *Atomare Funktion*: jede Funktion (Operation), die in der Aktionenebene spezifiziert wird und auch "nach außen" sichtbar ist. Jede atomare Funktion ist entweder eine *elementare Transaktion* oder eine *Umweltfunktion* (zu beiden Begriffen siehe weiter unten).

   Die Atomaren Funktionen sind diejenigen Operationen aus den Krankenhaus–Abläufen — Aufnahme, Arzt– und Behandlungs–Ablauf, Labor–Ablauf, Herzkatheter–Untersuchung und Entlassung — die in der Datenflußebene zu komplexen Abläufen verknüpft werden sollen.

In den Datenfluß-Diagrammen des Berichts [SNM+93] entspricht jedes rechteckige (*nicht* fett umrandete) Kästchen genau einer atomaren Funktion und umgekehrt.

Beispiele sind:

- *anmeldung* aus dem Ablauf Aufnahme zur Anmeldung von Patienten im Krankenhaus,
- *behandeln* aus dem Arzt–Ablauf zur Erstellung einer Behandlungs–Anordnung,
- *klBbAnalyse* aus dem Labor–Ablauf zur Bestimmung des kleinen Blutbilds,
- *HK_Untersuchung* aus dem Ablauf Herzkatheter–Untersuchung zur Ermittlung der sogenannten HK–Daten (im verwaltungstechnischen Sinne) und
- *entlassung* aus dem Ablauf Entlassung zur Erlösung von Patienten.

2. *Elementare Transaktion*, *eTa*: diejenigen atomaren Funktionen, die vom Benutzer des erwünschten HDMS–A–Systems aufrufbare Prozeduren modellieren, werden als "elementare Transaktionen" bezeichnet. Sie können auch *Top-Level*– oder *Benutzer*–Funktionen genannt werden.

   Details zum Begriff der "elementaren Transaktion" finden sich in [Huß93]; wichtig ist insbesondere:

   - eTa's sind atomare Funktionen *mit* Datenbankzugriff; in der Stelligkeit einer eTa kommt die Sorte *Db* ("Datenbank") sowohl als Zielsorte als auch genau einmal als Argumentsorte vor.
   - Wird eine eTa auf eine "integre" Datenbank angewendet, so liefert sie auch wieder eine integre Datenbank. Siehe dazu in Abschnitt 4.4 zu den "Integritätsbedingungen".
   - *Aus Benutzersicht ist eine elementare Transaktion die kleinste Einheit der Systemreaktion, die durch Eingaben an der Benutzerschnittstelle angesteuert werden kann. Menüs und Dialogabläufe sind im allgemeinen Zusammenfassungen solcher elementaren Transaktionen und werden in der funktionalen Essenz noch nicht betrachtet.* (Zitat aus [Huß93])
   - *Eine elementare Transaktion muß entweder als ganzes oder überhaupt nicht ausgeführt werden.* (ebenda)

In den Datenfluß–Diagrammen von [SNM+93] befinden sich eTa's stets *innerhalb* der großen, rechteckigen Boxen – und sind selbst von einer kleinen rechteckigen Box umrandet. Gemäß Einleitung und [MP88] sind die eTa's gerade die *essentiellen Aktivitäten* unseres Systems.

3. *Umweltfunktion*: diejenigen atomaren Funktionen, die Vorgänge aus der Umwelt des HDMS–A–Systems modellieren, werden "Umweltfunktionen" genannt. Es handelt sich also um system*externe* Funktionen, wie z.B. die Operation *med_hku*, die den Vorgang der Ermittlung der HK–Daten beschreibt (im *medizinischen* Sinne).

   Umweltfunktionen sind atomare Funktionen *ohne* Datenbankzugriff; in ihrer Stelligkeit kommt die Sorte *Db nicht* vor. In den Datenfluß–Diagrammen von [SNM+93] befinden sich Umweltfunktionen stets *außerhalb* der großen, rechteckigen Boxen und haben eine rechteckige, nicht–fette Umrandung.

4. *Ablauf*: der Begriff "Ablauf" wird leider in zwei zwar eng zusammenhängenden, aber doch unterschiedlichen Bedeutungen gebraucht:

   - Zum einen bezeichnet er die Verknüpfung atomarer Funktionen zu komplexeren Vorgängen; Abläufe in diesem Sinne werden visualisiert durch die Datenfluß–Diagramme in [SNM+93].
   - Zum andern bezeichnet er das, was weiter oben auch schon "Fachgebiet" genannt wurde, d.h. die Gesamtheit aller atomaren Funktionen sowie auch deren Zusammenspiel als Ablauf im obigen Sinne innerhalb einer bestimmten Abteilung des betrachteten Krankenhauses.

5. *Datenbankoperation*: eine Operation auf Datenbanken, d.h. eine Operation mit Zielsorte *Db* und/oder mit genau einem Vorkommen der Sorte *Db* in den Argumentsorten. ABER – im Unterschied zu eTa's – eine Operation aus dem *schematischen* Anteil der Gesamtspezifikation (also aus der Datenmodellebene).

   Kommt eine solche Operation insbesondere aus der Datenbankebene (eine Unterebene der Datenmodellebene), so wird sie auch *primitiv* genannt. Primitiv sind z.B. die *put*– und *update*–Operationen, nicht aber die Operationen aus der Komfortebene.

6. *Primitive Datenbankoperation*: siehe "Datenbankoperation"!

7. *Benutzer–Funktion*: siehe "elementare Transaktion"!

8. *Top-Level–Funktion*: siehe "elementare Transaktion"!

4.2 Die Systemzeit DateTime

In einigen Entities der HDMS–A–Datenbank – z.B. in denen vom Typ *HK_Daten* – werden bestimmte Zeitangaben abgespeichert. Mit anderen Worten: sie enthalten Attribute – z.B. *Anfangszeit* und *Endezeit* – deren Werte aus einem Domain *DateTime* zur Modellierung von Zeitangaben kommen. Diesem Domain entspricht in der OBSCURE–Spezifikation die Trägermenge einer Sorte *DateTime*, die von der Datenmodellebene geliefert wird (wie die Domains aller anderen Attribute auch). Die Atomare Ebene ATOMAR geht davon aus, daß die Sorte *DateTime* Zeitangaben modelliert, die sowohl das Kalenderdatum als auch die Tageszeit umfassen.
Es gibt elementare Transaktionen (z.B. *init_HKP*), die – werden sie aufgerufen – automatisch die Zeit von einer "eingebauten Systemuhr" ablesen und eine entsprechende Zeitangabe in bestimmten Entities eintragen sollen. Wie soll unsere Spezifikation so etwas modellieren? Wie modelliert man "Zeit"? Wie verhalten sich eTa's und "Systemuhr" zueinander?

1. Erster Versuch: wir spezifizieren eine (Hilfs–)Funktion

   $$datetime : Db \rightarrow DateTime$$

   Das ist offensichtlich Blödsinn: es ist natürlich nicht möglich, aus dem Zustand einer Datenbank eine Zeitangabe zu berechnen. Möglich wäre dies allerdings dann, wenn die Datenbank selbst eine eingebaute Uhr enthielte:

2. Der zweite Versuch spezifiziert daher zunächst einen Domain von "<u>D</u>aten<u>b</u>anken <u>m</u>it <u>e</u>ingebauter <u>U</u>hr":

   $$DbmeU := Db \times DateTime$$

   Die Funktion $datetime : DbmeU \rightarrow DateTime$ ist dann nichts anderes als die entsprechende Projektionsfunktion. Es könnte durchaus sein, daß man bei der Spezifikation des HDMS–A–Systems früher oder später um einen Domain *DbmeU* (oder etwas ähnliches) nicht herumkommt, *ABER*: arbeiten die atomaren Funktionen tatsächlich auf diesem Domain? Hat also etwa die Funktion *init_HKP* die Stelligkeit

   $$init\_HKP : DbmeU \times PatId \times ArztId \rightarrow DbmeU \times HkId \quad ?$$

   Das erscheint nicht sinnvoll! Eine solche Spezifikation würde ein System modellieren, in dem die atomaren Funktionen "schreibenden Zugriff" auf die Systemuhr hätten – das ist sicherlich nicht erwünscht.

In der Aktionenebene behandeln wir daher Zeitangaben in einer noch einfacheren – und im Grunde genommen auch viel naheliegenderen – Art und Weise. Elementare Transaktionen, die automatische Eintragungen von Zeitangaben vornehmen sollen, erhalten ein zusätzliches Argument der Sorte *DateTime*:

$$init\_HKP : \; Db \; \times PatId \times ArztId \times DateTime \rightarrow Db \times HkId$$

Eine solche Funktion modelliert genau das, was im erwünschten HDMS–A–System passieren soll: der Aufruf der atomaren Funktion *init_HKP* erfolgt

- *bei* einem ganz bestimmten Datenbankzustand,
- *zu* einem ganz bestimmten Zeitpunkt und
- *mit* ganz bestimmten Eingaben,

wobei es an dieser Stelle *nicht* interessiert, "woher" Datenbankzustand, Zeitpunkt und (sonstige) Eingaben kommen. Man beachte:

- Nicht nur die Argumentsorte *DateTime* modelliert etwas ganz anderes (nämlich den Zugriff auf eine Systemuhr) als z.B. die Argumentsorte *PatId* (nämlich eine Eingabe, die als Ausgabe der atomaren Funktion *überweisen* entsteht), sondern auch die Argumentsorten *ArztId* und *Db* modellieren jeweils wiederum völlig verschiedene Dinge: erstere eine Eingabe des Benutzers, letztere einen aktuellen, internen Zustand des Systems.
- Diese Überlegungen beleuchten den ganz allgemeinen Bezug zwischen formalen Spezifikationen, deren Semantik und den zu spezifizierenden Softwaresystemen. Die Semantik einer Spezifikation (hier also Algebren) dient als Modell für das erwünschte System. Folglich sind formale Spezifikationen absolut *wertlos*, wenn sie nicht mit präzisen Angaben zum Bezug zwischen Modell und System versehen werden (Beispiele für solche Bezüge befinden sich wenige Zeilen weiter oben!). Es ist zu befürchten, daß diesem Aspekt bei allen spezifikativen Arbeiten zur Fallstudie HDMS–A zuwenig Beachtung geschenkt worden ist – nicht zuletzt aus Zeitmangel. Das gilt *auch* für die vorliegenden Arbeiten [Hec93a, Aut93, Ben93].

4.3 Definiertheitsprädikate

Betrachte die OBSCURE–Definition der Predecessor–Funktion *pred* auf den (wie üblich definierten) natürlichen Zahlen der Sorte *Nat*:

```
pred(n)  <-  CASE n OF
               null: ERROR(Nat)
               succ(n'): n'
             ESAC
```

Für die Zahl Null liefert *pred* ein ausgezeichnetes Element "Bottom" aus der Trägermenge der Sorte *Nat* (in OBSCURE–Spezifikationen denotiert mit "ERROR(Nat)"). Das Bottom–Element modelliert (zumindest in den vorliegenden Arbeiten [Hec93a, Aut93, Ben93]) stets einen *Fehlerfall*: würde man obige Funktion implementieren, so müßte sie für das Argument Null (terminieren und) irgendeine Fehlermeldung ausgeben. Eine ausführliche Anmerkung zu Bottom–Elementen in OBSCURE findet sich in Abschnitt 5.2. Wir schreiben nun mitunter Definitionen wie z.B:

```
make_odd(n) <- IF (even(n)=true) THEN pred(n) ELSE n FI ;
```

in der etwas ausführlicheren – jedoch äquivalenten – Form:

```
make_odd(n)  <-  IF (make_odd_def(n)=true)
                   THEN IF (even(n)=true) THEN pred(n) ELSE n FI
                   ELSE ERROR(Nat)
                 FI
```

... wobei natürlich *make_odd_def* definiert wird durch:

```
make_odd_def(n) <- IF (n=null) THEN false ELSE true FI ;
```

Das "**Definiertheitsprädikat**" *make_odd_def* gibt also an, in welchen Fällen die Funktion *make_odd* fehlerfrei ausgeführt werden kann. Es gibt zwei Gründe für die Verwendung solcher – von der erwünschten Semantik her überflüssigen – Definiertheitsprädikate:

- Liegen die Definitionen von *pred* und *make_odd* "weit auseinander", so hat der Leser unter Umständen längst vergessen, daß der Aufruf von *pred* und damit auch der von *make_odd* zu einem Fehler führen kann. Die erste der obigen Definitions–Versionen für *make_odd* ist zwar semantisch äquivalent zur zweiten, läßt aber die Möglichkeit eines Fehlerfalles nicht erkennen.

- Wir verwenden Definiertheitsprädikate genau an den Stellen, an denen die Spezifikationen aus [Het93, SNM+93] das (polymorphe) Definiertheitsprädikat der Sprache SPECTRUM verwenden, und erhoffen uns dadurch ein erhöhtes Maß an Vergleichbarkeit. Das ist auch der Grund für den in unserem Kontext äußerst unglücklichen Namen *Definiertheits*prädikat, der eigentlich durch *Fehlerfall*prädikat ersetzt werden müßte.

4.4 Statische Integritätsbedingungen

Das Konzept der "*statischen Integritätsbedingung*" (im folgenden kurz "Integritätsbedingung" genannt) kommt üblicherweise in E/R–Modellen vor. Wir behandeln es analog zu [Het93] — mit *einem* Unterschied: sowohl die Beachtung der Schlüsseleigenschaft (siehe dazu auch 5.1) als auch die Beachtung obligatorischer (zwingender) Attribute gelten in unseren Schablonen ([Aut93]) als Integritätsbedingungen. Somit werden mit diesem Konzept behandelt:

1. Nicht nur die von [Het93] sogenannten "Allgemeinen Integritätsbedingungen" (die im Hdms–A–E/R–Modell nicht formuliert werden können, wohl aber in Spectrum und Obscure),

2. sondern auch:

   - Schlüssel
   - Optionale/Obligatorische Attribute
   - Grade von Relationshiptypen[11] ("Kardinalitätsangaben") – dies in Übereinstimmung mit [Het93].

   In [Hoh93] werden diese drei speziellen Arten von statischen Integritätsbedingungen auch "strukturelle Einschränkungen" genannt.

Die Modellierung von Integritätsbedingungen mittels Funktionen der Form

$$C : Db \rightarrow bool$$

findet sich sowohl in der Datenmodellebene (für die im Hdms–A–E/R–Modell formulierbaren Bedingungen) als auch in der Atomaren Ebene (für die "allgemeinen Integritätsbedingungen"). Übrigens ist das im Abschnitt 2.3 erwähnte "OK–Prädikat" nichts anderes als die Konjunktion sämtlicher in der Datenmodell– und der Atomaren Ebene auftretender Integritätsbedingungen.

Man mache sich klar, daß unsere Spezifikation häufig gegen solche Integritätsbedingungen verstößt – genauer: es gibt durchaus Träger *db* der Datenbank–Sorte *Db*, die das OK–Prädikat nicht erfüllen (also auch mindestens eine der Integritätsbedingungen nicht). Diese Träger sind keineswegs "Junk", sondern können mittels Datenbankoperationen kreiert werden. Es gilt jedoch folgendes:

[11] Dazu gehören insbesondere auch sogenannte "zwingende Partizipationen" von Entities an Relationships

Jede elementare Transaktion der Atomaren Ebene liefert (zumindest) immer dann eine integre Datenbank *db'* (*OK*(*db'*)=true), wenn sie mit einer integren Datenbank *db* aufgerufen wird (*OK* (*db*)=true) und wenn kein Fehlerfall eintritt (im Fehlerfall liefert sie ein Bottomelement).

Genau das ist die Aussage des Exportaxioms in der Atomaren Ebene (vgl. in deren Visualisierung an der Stelle `<Anwendung des OK-Pr"adikats als Ex port-Axiom auf:>`); das Exportaxiom macht also eine Zusicherung über die von der Atomaren Ebene gelieferten eTa's, die natürlich *bewiesen* werden müßte!

4.5 Monolisten

4.5.1 Das Phänomen

Beim algorithmischen Spezifizieren sind *endliche Mengen* so gut wie unbrauchbar. Wir wollen uns das klar machen anhand einer Spezifikation SET endlicher Mengen der Sorte *Set_of_El* über einer Elementsorte *El*. In den meisten Fällen wird man

- entweder eine Funktion *choose* : *Set_of_El* $\rightarrow$ *El* benötigen bzw. eine Funktion *rep* : *Set_of_El* $\rightarrow$ *List_of_El* (was äquivalent ist!),
- oder man braucht Funktionen wie *test* : *Set_of_El* $\times$ ... $\rightarrow$ *Bool* , die feststellen, ob in einer Menge Elemente mit bestimmten Eigenschaften enthalten sind. Im Falle von HDMS–A wäre z.B. eine Funktion interessant, die die aktuelle Datenbank (und damit eine endliche Menge von Entities) nach Entities mit einem bestimmten konkreten Schlüssel durchsucht.

Nun gibt es zwei Möglichkeiten:

1. Entweder stellt SET selbst bereits eine *choose*–, eine *rep*– oder eine sonstige, äquivalente Funktion zur Verfügung. Dann kann man zwar auch beliebige andere Funktionen auf Mengen definieren (z.B. obige *test*–Funktionen), spricht aber eigentlich nicht mehr über endliche Mengen, sondern über (endliche) *Listen* – genauer: über *Monolisten*. Die Existenz solcher Funktionen bedeutet nämlich, daß die Elemente in einer Menge stets in eindeutiger Weise angeordnet sind – genau das aber ist dann eine *Monoliste*: eine (endliche) Liste von Elementen, in der deren Reihenfolge berücksichtigt wird, nicht aber die Häufigkeit (wiederholter) Vorkommen. Eine Monoliste ist also eine Liste *ohne* doppelte Elementvorkommen.

Genau genommen gilt: die Menge aller endlichen Mengen (mit *choose*!) über einer Elementsorte verhält sich genau so wie eine Teilmenge aller endlichen Monolisten über dieser Elementsorte. Diese Teilmenge von Monolisten erhält man z.B. dadurch, daß man jeder *Menge* auf folgende Art eine *Monoliste* zuordnet:

Beispiel: der Menge *set* := $\{el_1, el_2, el_3\}$ entspricht die Monoliste $\langle el_1, el_2, el_3 \rangle$, wobei gelte:

$$el_1 := choose(set)$$
$$el_2 := choose(delete(el_1,set))$$
$$el_3 := choose(delete(el_2,delete\ (el_1,set)))$$

2. Oder andernfalls kann nicht mehr jede beliebige Funktion auf Mengen "im Nachhinein" spezifiziert werden. Man überlege sich, daß nicht nur *choose* oder *rep nicht* (algorithmisch) mit Hilfe der übrigen Mengenoperationen definiert werden können, sondern auch beispielsweise die obigen *test*–Funktionen nicht.

Diese Überlegungen legen es nahe, von vorneherein nicht den Modul SET – und seine endlichen Mengen – zu spezifizieren und zu verwenden, sondern einen Modul mit endlichen Monolisten. In unserem Falle: nicht den Standard–Modul MK_SET, sondern den Standard–Modul MK_MONOLIST.

4.5.2 Die Konsequenzen

Die erste Konsequenz ist eine Definition:

Eine *Monoliste* ist eine Liste, in der jedes Element genau einmal oder aber überhaupt nicht vorkommt.

Eine weitere Konsequenz ist die Tatsache, daß unsere OBSCURE–Spezifikation der Funktionalen Essenz nicht den Standard–Modul MK_SET benutzt, sondern den Standard–Modul MK_MONOLIST. Da Monolisten ein "Zwischending" sind zwischen Mengen und Listen, erinnern die von diesem Modul gelieferten Funktionen auf Monolisten (Sorte *MLst*) sowohl an Mengenoperationen als auch an Listenoperationen; es gibt hier sowohl ein

$$del : El \times MLst \rightarrow MLst$$

als auch ein

$$tail : MLst \rightarrow MLst$$

Obwohl für unsere Zwecke ohne Belang, sei dazu angemerkt, daß weder die Vereinigung noch der Schnitt auf Monolisten kommutativ ist. Dazu betrachte man zwei Monolisten l_1 und l_2:

- Ihre Vereinigung $l := l_1 \cup l_2$ erhält man, indem man die Elemente aus l_2 der Reihe nach an die Monoliste l_1 anhängt, sofern sie in dieser nicht schon vorhanden sind. Das gleiche Resultat würde die *Konkatenations*–Funktion liefern.

- Ihren Schnitt $l := l_1 \cap l_2$ erhält man, indem man aus l_1 alle Elemente entfernt, die *nicht* in l_2 vorkommen.

Eine letzte Konsequenz ist eher *fiktiv*; man *kann* nämlich durchaus auch beim algorithmischen Spezifizieren endliche Mengen "benutzen" – allerdings nicht im Sinne eines eigenständigen Mengenmoduls SET (bzw. MK_SET). Vielmehr müßte man dann innerhalb einer jeden (Gesamt–)Spezifikation Listen oder Monolisten verwenden und diese Gesamtspezifikation in einem "letzten Schritt" mit zahlreichen SUBSET– und/oder QUOTIENT–Konstruktionen umgeben, die aus den entsprechenden Listen–Sorten wirkliche *Mengen*–Sorten machen würden. Es handelt sich dann sozusagen um die Beschreibung der Benutzung des Datentyps *Menge* ohne die Benutzung dieses Datentyps zur Beschreibung (es gibt keinen SET–Modul und "innerhalb" der Spezifikation auch keine Sorte *Set_of_El*!). Dieses Verfahren ist natürlich indiskutabel!

5 Größere Anmerkungen

5.1 Schlüssel

5.1.1 Vorbemerkung

Das Konzept des "*Schlüssels*" kommt üblicherweise in E/R–Modellen vor – so auch in "unserem" E/R–Modell, dem HDMS–A–E/R–Modell. Dessen Definition findet man (implizit gegeben) in [Het93] bzw. in [Aut93] (also bei der Beschreibung der Übersetzungsfunktionen *er2spec* bzw. *er2obs*). Zu Begriffen wie "E/R–Modell" und "E/R–Schema" siehe Abschnitt 3.1. Zu Begriffen wie "Entity", "Relationship" und "Attribut" siehe [Aut93]: dort befinden sich präzise (und z.T. mathematisch rigorose) Definitionen.
Die auf all diesen Begriffen aufbauenden Definitionen zum Begriff "Schlüssel" unterscheiden zwischen *konkreten Schlüsseln* und *abstrakten Schlüsseln*. Diese Unterscheidung ist spezifisch für das *Saarbrücker* E/R–Modell und erhebt nicht den Anspruch, dem Standardvorgehen zu entsprechen. Wir sind

uns durchaus *nicht* im klaren darüber, ob man eine solche Unterscheidung wirklich unbedingt braucht, oder ob "ganz normale" (konkrete) Schlüssel ausreichend wären.

5.1.2 Konkrete Schlüssel

Ein "*konkreter Schlüssel*" für einen Entitytyp (z.B. *Patient*) ist eine Menge von Attributen – oder genauer: eine Menge von Attributnamen, die in diesem Entitytyp vorkommen. Beispiel: die einelementige Menge $\{Name\}$ für *Patient*. Ein "(benannter) Entityp mit konkretem Schlüssel" ist einfach ein (benannter) Entitytyp, für den eine solche Menge von Attributnamen explizit als konkreter Schlüssel ausgezeichnet ist. Ein "*Schlüsselattribut(name)*" (eines solchen benannten Entitytyps mit konkretem Schlüssel) ist eines der zu diesem Schlüssel gehörenden Attribute (bzw. dessen Name).
Die Auszeichnung eines konkreten Schlüssels für einen Entitytyp entspricht der Formulierung einer Anforderung an die Datenbank – nämlich: es darf darin nie zwei *verschiedene* Entities (des entsprechenden Typs) geben, deren "Schlüsselwerte" gleich sind[12]. Dabei ist der "*Schlüsselwert*" die Menge der dem konkreten Schlüssel entsprechenden Attribute (inklusive ihrer Werte!).
Beispiel: gibt es in dem mit *Patient* benannten Entitytyp einen Attributnamen *name* und ist dieser als konkreter Schlüssel für *Patient* ausgezeichnet, dann gilt für die entsprechenden Datenbanken stets: wenn zwei Entities aus der zu *Patient* gehörenden Entitymenge dengleichen Namen haben, dann sind sie identisch.
Eine Formulierung/Auszeichnung dieser Art ist *immer* möglich, da trivialerweise immer erfüllt: die Menge *aller* Attributnamen eines Entitytyps *muß* ein konkreter Schlüssel für diesen Typ sein.

5.1.3 Konkrete Schlüssel von Objekten aus der Umwelt

Wir wollen nun den (benannten) Entitytyp *Patient* unter dem Aspekt betrachten, daß es einen entsprechenden "Typ" in der wirklichen Welt gibt. Jede Entity des Typs *Patient* ist ein elektronisches bzw. konzeptionelles Abbild eines wirklich existierenden Patienten (eines Menschen).
Die menschlichen Patienten haben eine wichtige Eigenschaft: ihre Position in der Reihe all derjenigen Menschen, die jemals in unserem Krankenhaus behandelt worden sind, zur Zeit dort noch behandelt werden, oder jemals dort in Zukunft behandelt werden. Mit anderen Worten: die Abzählungsnummer im Sinne der Reihenfolge ihrer (Erst–)Aufenthalte. Uns interessiert also die (injektive) Abzählung

[12] Das ist die schon früher erwähnte "Schlüsseleigenschaft"!

$$patid : Menge\_aller\_menschlichen\_Patienten \rightarrow PatId$$

Dabei bezeichnet *PatId* die Menge aller Abzählungsnummern.
Diese Eigenschaft sollte in unserem Bild[13]von der Welt (d.h. in unserem E/R–Schema) nicht fehlen: wir nehmen sie als Attribut(namen) in den Entitytyp *Patient* mit auf – ein ganz normales Attribut also, eine Eigenschaft wie jede andere (Name, Alter, ...) auch. Ganz offensichtlich *muß* dieser Attributname (bzw. die entsprechende einelementige Menge) als konkreter Schlüssel ausgezeichnet werden – andernfalls stimmt unser (Gesamt-)Bild von der wirklichen Welt nicht.
Behauptung: *alle* Entitytypen mit Analogon in der wirklichen Welt "haben" eine solche Abzählungsnummer und

- diese sollte als Attribut berücksichtigt,
- als konkreter Schlüssel ausgezeichnet
- und auch als *abstrakter* Schlüssel (dazu gleich mehr) verwendet werden.

5.1.4 Konkrete Schlüssel und die Identifizierung von Objekten aus der Umwelt

Von der Philosophie der "Funktionalen Essenz" her (vgl. [MP88]) – insbesondere im Sinne der Beschränkung auf den "Kern" von Systemen – ist die "*Identifizierung* von Objekten aus der Umwelt (d.h. die Feststellung ihrer Abzählungsnummer) ganz einfach; im Falle der Patienten sieht das beispielsweise so aus:

- Bei der Aufnahme(prozedur) im Krankenhaus wird jeder Patient P gefragt, ob er dort schon einmal behandelt worden ist:
  - falls ja, dann muß er seine Abzählungsnummer *patid(P)* nennen, die man ihm bei seinem letzten Aufenthalt mitgeteilt hat.
  - Falls nein, dann muß er der *nächste* neue Patient sein, d.h: seine Abzählungsnummer muß die nächste, auf die bisher in der Datenbank vorhandenen folgende Abzählungsnummer sein. Diese wird in der entsprechenden (neu anzulegenden) Entity des Typs *Patient* als Attribut *patId* eingetragen und ihm außerdem natürlich mitgeteilt.

[13]Man könnte es auch "Modell" von der Welt nennen – das wäre hier jedoch verwirrend, weil der Begriff "E/R–Modell" bereits mit einer (völlig anderen!) Bedeutung versehen wurde!

Man mache sich zweierlei klar:

1. Von der Möglichkeit, daß Patienten absichtlich oder unabsichtlich die Unwahrheit sagen oder ihre Abzählungsnummer einfach vergessen, wird hier also abstrahiert. Allerdings muß die Spezifikation der Funktionalen Essenz genügend Hilfsfunktionen zur Verfügung stellen, mit deren Hilfe ein Paket von Identifizierungsoperationen beschrieben werden könnte, wie es in einer den Systemkern umgebenden Hülle sicher vorhanden sein muß. Vielleicht sollte man daher die in [SNM+93] für den Ablauf "Aufnahme" gegebene Funktion *identifikation* weniger als Beispiel für eine verbindliche Identifizierungsoperation sehen, sondern vielmehr als *ein* Bestandteil eines solchen Pakets.

   Es gibt kein perfektes Identifizierungspaket!

   Gerade das Identifizierungspaket des HDMS–A–Systems ist eine Komponente, die extrem dem Problem der *Erweiterbarkeit* bzw. der *Modifizierbarkeit* unterworfen ist.

2. Falsche Identifizierungen können natürlich niemals wirklich vollständig ausgeschlossen werden. Oder allgemeiner: das durch den aktuellen Zustand der HDMS–A–Datenbank gegebene Bild von der wirklichen Welt kann durchaus jederzeit Unkorrektheiten enthalten. Statt also zuviel Energie für die Erstellung "perfekter" Identifizierungsoperationen zu verschwenden, sollte man besser für möglichst leicht durchführbare Korrekturmaßnahmen sorgen.

5.1.5 Abstrakte Schlüssel

Abstrakte Schlüssel sollen dem (eindeutigen) Zugriff auf Entities innerhalb des HDMS–A–Systems dienen. Im einfachsten Falle stimmen sie mit den konkreten Schlüsseln überein. In manchen Fällen aber könnte es von Vorteil sein, stattdessen die "kodierte" Version eines konkreten Schlüssels zu benutzen. Ein "*abstrakter Schlüssel*" ist eine solche Kodierung eines konkreten Schlüssels.
Als Beispiel betrachten wir einen dem Entitytyp *Arzt* in einer 1:1–Beziehung zugeordneten Entitytyp "*Arzt_Control*". Dieser Typ beinhalte insbesondere zwei Attribute:

- *name*; dessen Werte sollen aus dem Domain *Name* sein und natürlich jeweils den Namen (d.h. Vornamen und Familiennamen) des entsprechenden Arztes modellieren;

- *passWd*; dessen Werte sollen aus dem Domain *Passwd* sein und jeweils ein Paßwort modellieren – etwa im Sinne eines Benutzer–Paßwortes für das HDMS–A–System.

Die Menge dieser beiden Attribut(nam)e(n) sollte als konkreter Schlüssel für den Entitytyp *Arzt_Control* ausgezeichnet werden. Er kann dann beim Zugriff auf Entities vom Typ *Arzt_Control* in eindeutiger Weise benutzt werden – *aber* ganz bestimmt nicht "wörtlich", sondern nur in seiner codierten Form. D.h: wir setzen eine Funktion

$$crypt : Name \times PassWd \rightarrow Abstr\_Arzt\_Ctrl\_Key$$

voraus, die aus je einem Namen und einem Paßwort einen abstrakten Schlüssel irgendeiner Sorte *Abstr_Arzt_Ctrl_Key* macht. Wenn diese Sorte beispielsweise mit *Nat* identisch ist, dann ist der abstrakte Schlüssel für *Arzt_Control* nicht nur ein den konkreten Schlüssel geheimhaltender, sondern darüberhinaus sogar ein "leicht zu handhabender".
Wir beschließen diesen Abschnitt mit

- einer Bemerkung: die Existenz von Attributen impliziert *nicht* die Existenz entsprechender Recordfelder im lauffähigen System. Es ist also beispielsweise vorstellbar, daß die Paßwörter aus den Entities vom Typ *Arzt_Control* in der Datenbank des HDMS–A–Systems nicht zu finden sein werden – genau so wenig, wie die Paßwörter in UNIX–Systemen im Dateibaum zu finden sind.

- und einem Fazit:

  1. Im allgemeinen gibt es zu jedem Entitytyp nicht nur einen konkreten Schlüssel, sondern auch einen *abstrakten* – realisiert durch mehr oder weniger aufwendige Funktionen der Art von obigem *crypt*. Abstrakte Schlüssel dienen dem eindeutigen Zugriff auf Entities, was Zugriffe mittels des konkreten Schlüssels durchaus nicht generell ausschließen soll.
  2. In Spezialfällen – z.B. bei *allen* Entitytypen mit Analogon in der wirklichen Welt wählt man (implizit oder explizit) als Kodierung die Identität; z.B:

     $$\begin{array}{l} codierung : PatId \rightarrow PatId \\ \text{mit } codierung(pi) := pi \end{array}$$

  3. In anderen Spezialfällen geht man dagegen beliebig subtil vor; z.B. im Falle der Funktion

     $$codierung = crypt : Name \times PassWd \rightarrow Abstr\_Arzt\_Ctrl\_Key$$

5.1.6 Schlüsselgenerierung

Der Begriff Schlüssel*generierung* ist problematisch. Wir haben in den vorangegangenen Abschnitten gesehen, daß (konkrete) Schlüssel *Eigenschaften* von bzw. *Informationen* über Objekte(n) entsprechen – genau wie andere Mengen von Attributen auch. Man kann sie also eigentlich nicht generieren, sondern nur *feststellen*. Das ändert allerdings nichts an der Tatsache, daß beispielsweise für den Entitytyp *Patient* eine Funktion

$$new\_keyPatient : Db \to KeyPatient$$

benötigt wird – nämlich genau dann, wenn ein Patient bei der Aufnahme angibt, noch nie in unserem Krankenhaus behandelt worden zu sein. Analog braucht man für den Entitytyp *Aufenthalt* eine Funktion

$$new\_keyAufenthalt : Db \times PatId \to KeyAufenthalt$$

Welche (Mindest–)Eigenschaft solche Funktionen *immer* haben müssen und wie sie in der Obscure–Spezifikation realisiert werden, wurde in Abschnitt 3.3 ("Schematisches und Spezifisches") beschrieben.
Man beachte, daß die Spectrum–Spezifikation hier einen völlig anderen Weg geht: sie kennt weder *next*– noch *new_key*–Funktionen. Stattdessen werden von der Datenmodellebene Funktionen

$$genkeyE : Db \times (KeyE \to Bool) \to KeyE$$

zur Verfügung gestellt. Diese Funktionen erfüllen stets die – schematisch formulierbare – Eigenschaft, "neue" Schlüssel zu liefern. Zusätzliche (spezifische!) Eigenschaften der Schlüssel können in der Atomaren Ebene durch Übergabe eines entsprechenden Prädikats $p : KeyE \to Bool$ gefordert werden. Insbesondere sind dort möglich Aufrufe wie z.B:

$$genkeyHK\_Befund(db\,,\ \lambda x.\ true)$$

Die Übergabe des Prädikats $(\lambda x.\ true)$ bedeutet, daß der zu "generierende" Schlüssel keine weitere Eigenschaft erfüllen muß außer der, "neu" zu sein (also obige Mindesteigenschaft). In der Obscure–Spezifikation würde an dergleichen Stelle ein Aufruf

$$new\_keyHK\_Befund(db)$$

stehen. Selbstverständlich sind (beim Aufruf der Spectrum–Funktionen) jederzeit Prädikate formulierbar, die der (von der Obscure–Funktion gelieferte) Schlüssel $new\_keyHK\_Befund(db)$ (i.a.) *nicht* erfüllt. Mit anderen Worten: die Obscure–Spezifikation legt für jeden Entitytyp E die Eigenschaften neu generierter Schlüssel ein für allemal fest – und zwar durch die Definition der Funktion new_keyE. Andere Eigenschaften können "im Nachhinein" nicht mehr gefordert werden.

5.2 Bottomelemente

5.2.1 Bottomelemente als Fehlermeldungen

Jede Trägermenge (einer Sorte S) in einer OBSCURE–Spezifikation enthält stets einen ausgezeichneten Träger, dessen Existenz "automatisch" gewährleistet ist, der "*Bottomelement*" genannt wird und der mit '$\perp_S$' denotiert wird (auch kurz mit '$\perp$'). Beim Erzeugen einer Sorte S wird auch automatisch ein Konstantenzeichen *ERROR(S)* der Sorte S kreiert:

$$\mathit{ERROR(S)} : \text{->}\ S$$

Dieses Zeichen wird in jeder Algebra mit dem entsprechenden Bottomelement $\perp_S$ belegt.
Was modellieren Bottomelemente? In unseren Spezifikationen modellieren sie *Fehlerfälle* bzw. *Fehlermeldungen* und nichts anderes! Spezifizieren wir also beispielsweise eine Funktion *pred*[14] durch:

- $pred : Nat \rightarrow Nat$
- mit pred(n) <– **CASE** n **OF**
  null: ERROR(Nat)
  succ(n'): n'
  ESAC

so ist sie als Modell für eine Prozedur zu verstehen, die – aufgerufen mit dem Argument Null – eine Fehlermeldung liefert (und zwar nach endlicher Zeit !). Dazu noch drei Anmerkungen:

1. Zunächst zur Syntax: die Schreibweise "*ERROR(Nat)*" erinnert zwar an die Bildung (echter) Terme, hat damit aber nichts zu tun: es handelt sich lediglich um vordefinierte Konstantenzeichen. Nun zurück zur Bedeutung von Bottomelementen:

2. Bottomelemente haben nichts mit *Nichtterminierung* zu tun! Zwar ist es so, daß auch die Definition

   pred(n) <– ...
   null: pred(null)
   ...

[14] Wir unterscheiden im folgenden nicht sonderlich scharf zwischen Funktions*zeichen* (pred, null) und deren Interpretation (*pred*, Null)!

eine Funktion *pred* liefert, die für das Argument Null den Wert $\perp$ zurückgibt (die Definition ist zur weiter oben gegebenen äquivalent). Dennoch wäre es nicht sinnvoll zu sagen, daß *pred* in dem einen Fall eine Fehlermeldung produziert und im anderen überhaupt nicht terminiert (jeweils für das Argument Null). Denn dann hinge ja die Bedeutung der Semantik einer Spezifikation von der verwendeten Ausdrucksweise ab, was sicher nicht erstrebenswert ist.

Es gibt allerdings (mindestens) zwei Alternativen zu der von uns gewählten Fehlerfall–Bedeutung:

- Man könnte auch festlegen, daß das Bottomelement stets Nichtterminierung modelliert,
- oder man könnte festlegen, daß es stets "Fehlerfall ODER Nichtterminierung" modelleriert: und zwar in einem losen Sinne, derart daß sowohl Prozeduren, die eine Fehlermeldung liefern, als auch solche, die nicht terminieren, zulässige Implementierungen wären (unabhängig vom Spezifikationstext natürlich).

Solche Festlegungen widersprechen aber der Spezifikationspraxis, in der man oft genug die *ERROR*–Konstanten explizit benutzt, um einen Fehlerfall anzudeuten. Auf diese Möglichkeit der Fehlermodellierung wollten wir auf keinen Fall verzichten – wohl aber auf die Möglichkeit festzulegen, eine Prozedur müsse oder dürfe unendlich lange laufen!

Die Tatsache, daß so mancher Interpretierer (z.B. der aktuelle im OBSCURE–System) mit der zweiten Definition von *pred* nicht zurecht kommt, braucht uns als *Spezifizierer* nicht zu interessieren. Der Interpretierer würde – völlig unzulässiger Weise – bei einem Aufruf mit "*pred(null)*" auf Terminierung verzichten, statt – korrekter Weise – das Bottomelement $\perp_{Nat}$ zu liefern. Aber das ist ein Problem des Interpretierens, nicht des Spezifizierens...

3. Das Bottomelement hat auch nichts mit *Undefiniertheit* zu tun. Die obige Funktion *pred* z.B. ist an der Stelle Null durchaus definiert – sie liefert schließlich ein Element aus ihrem Wertebereich (nämlich das Bottomelement). Man könnte also bestenfalls sagen, die Funktion *pred* modelliere eine Prozedur, die an der Stelle Null undefiniert ist. Was aber soll das dann bedeuten?

   - Losheit? Oder mit anderen Worten: sie modelliert nicht *eine* Prozedur, sondern mehrere? Dieser Gedanke führt in unserem Kontext ein wenig zu weit – und vor allem: wir erhielten damit nur einen "sehr mageren" Formalismus zur Behandlung von Losheit: man sollte sie also besser mit Hilfe einer modifizierten Semantik behandeln.

- Nichtterminierung? Das haben wir bereits verworfen!
- Nichtdeterminismus? Für diesen Gedanken gilt dasselbe wie für Losheit.

Fazit: der Begriff der "undefinierten Prozedur" ist ziemlich sinnlos!

5.2.2 Unglückliche Benennungen und leere Attribute

Zunächst sei an dieser Stelle noch einmal auf die Definiertheitsprädikate aus Abschnitt 4.3 verwiesen: ihr Name ist irreführend, weil sie etwas mit Fehlerfällen und dem Bottomelement zu tun haben, nicht aber mit Definiertheit/Undefiniertheit. Der Name kommt vom gleichnamigen Prädikat der Sprache SPECTRUM.
Eine weitere sehr unglückliche Namensgebung tritt in der Attributebene der Datenmodellebene auf. Man braucht dort zusätzliche Elemente in bestimmten Domains, um "leere" (oder unbesetzte) Attribute modellieren zu können. Die entsprechenden Konstantenzeichen werden *UNDEF_S* genannt. Als Beispiel betrachte man einen Entitytyp *Patient* mit einem Attribut (einem "Eintrag") *name*. Die Werte für dieses Attribut kommen aus einem Domain *Name*. Werden Namen als Strings repräsentiert, dann genügt jedoch nicht die Wahl *Name* := *String* – vielmehr setzt man:

$$Name := String \cup \{UNDEF\_Name\}$$

Dabei ist *UNDEF_Name* eine benutzerdefinierte Konstante. Attribute, denen nicht explizit ein Wert zugewiesen wird (z.B. beim Kreieren einer Entity), erhalten den (Default–)Wert *UNDEF_S* (hier *UNDEF_Name*). Mit anderen Worten: der Wert *UNDEF_Name* modelliert die Tatsache, daß das Attribut *name* in einer Entity vom Typ *Patient* (noch) leer bzw. unbesetzt ist. Dazu ist anzumerken:

1. Der Name "*UNDEF_S*" kommt aus [Het93]; er ist zumindest hier ein wenig irreführend, weil er Assoziationen weckt, die in die Richtung undefinierter Funktionswerte, nichtterminierender Aufrufe oder auch in die von Bottomelementen gehen. Diese Assoziationen sind *nicht* zutreffend!

2. Könnte man nicht die – ohnehin stets vorhandenen – Bottomelemente die Rolle leerer Attributwerte spielen lassen? Man kann nicht! Und das gleich aus zwei Gründen:

- Zum einen ist es "rein technisch" gar nicht möglich. Da OBSCURE–Funktionen stets strikt sind in allen Argumenten, würde beispielsweise die Funktion zur Generierung einer Entity vom Typ *Patient* – aufgerufen mit dem Wert $\perp_{Name}$ für das Attribut *name* – nicht etwa eine Entity mit einem leeren Attribut liefern, sondern das Bottomelement $\perp_{Patient}$ der Sorte *Patient*.
- Zum andern ist es auch gar nicht sinnvoll. Denn selbst wenn es Entities gäbe, deren Attribut *name* mit $\perp_{Name}$ belegt ist, was sollte es dann bedeuten, wenn die Zugriffsfunktion

  $$name : Patient \rightarrow Name$$

  für eine Entity P dieses Bottomelement liefern würde:

  $$name(P) = \perp_{Name} \quad ?$$

  Soll das bedeuten, daß ein Fehler aufgetreten ist? Oder daß ein Fehler "geliefert" wird? Oder daß das Attribut leer ist?

3. Das Kreieren einer Entity mit einem leeren Attribut ist a priori kein Fehler, sondern eben ein ganz normaler Vorgang. Nur bei sogenannten "zwingenden" (oder: "obligatorischen") Attributen würde eine solche Kreation eine Fehlermeldung verursachen:

$$createPatient(\ldots, UNDEF\_Name, \ldots) = \perp_{Patient}$$

5.2.3 Bottomelemente und Funktionale Essenz

In der OBSCURE–Spezifikation der Funktionalen Essenz von HDMS–A (und übrigens auch in der entsprechenden SPECTRUM–Spezifikation) gibt es einige Fälle, in denen (primitive) Datenbankoperationen (wie z.B. die *put*– und die *update*–Funktionen), aber auch elementare Transaktionen das Bottomelement liefern. Nun ist die von solchen Bottomelementen modellierte Fehlerbehandlung sicher sehr rudimentär – sie ist weder ausgefeilt im Sinne detaillierter Fehlerdiagnosen noch im Sinne einer sorgfältigen Fehlerbehandlung oder gar Fehlervermeidung. Dies wird folgendermaßen gerechtfertigt:

1. Was primitive Datenbankoperationen (und überhaupt *alle* Operationen außer den atomaren) in Fehlerfällen tun, ist ohnehin uninteressant. In der Gesamt– und Anforderungs–Spezifikation für das HDMS–A–System sollten sie "nach außen" nicht mehr sichtbar sein.

2. Die rudimentäre Fehlerbehandlung bei elementaren Transaktionen erklärt sich aus der Philosophie der "Funktionalen Essenz" heraus (vgl. Einleitung und [MP88]). *Ein* Aspekt dieser Philosophie ist die Trennung zwischen Systemkern und Systemoberfläche. In unserem Falle bedeutet das: eine sorgfältigere Fehlerbehandlung bleibt einer eigens zu spezifizierenden Systemoberfläche vorbehalten.

   In diesem Zusammenhang lohnt sicher ein Blick auf die bzgl. der HDMS–A–Oberfläche geleistete Arbeit: siehe dazu [Shi94] und [MZ94].

6 Reflexion

Am Schluß dieser Abhandlung soll über die darin beschriebene Arbeit und ihren Sinn nachgedacht werden; beantwortet werden sollen die Fragen

1. Was ist uns (den Urhebern) an dieser OBSCURE–Spezifikation wichtig?

2. Welches sind die Erkenntnisse über OBSCURE?

3. Welches sind die allgemeineren Erkenntnisse?

6.1 Was uns wichtig ist

6.1.1 Die Modularisierung

Wichtig ist uns die *Modularisierung* der Spezifikation. Wir haben deshalb versucht, sie in zahllosen Bildern dem Leser zu vermitteln. Erste Visualisierungen dieser Art finden sich bereits in Abschnitt 2.3 des vorliegenden Berichts; verfeinerte Versionen werden in [Aut93] und [Ben93] gegeben.
Das Modularisieren von Spezifikationen dient zwei wesentlichen Zwecken: dem der *Wiederverwendung* und dem der *Handhabbarkeit*. *Ein* Aspekt der letzteren ist die *Verstehbarkeit*: wir begreifen die Struktur einer Spezifikation (inklusive ihrer Visualisierung) als "Schlüssel zum Verständnis":

> Specification–building operations are used to build specifications
> in a structured manner and to exploit this structure as a guide
> in their use and understanding.

(Zitat aus [vL90]; vgl. auch [Hec93b]). Die hier verwendeten Strukturbilder – und insbesondere die darin verwendeten Benennungen – können vom Leser dazu benutzt werden, sich in der gesamten Dokumentation zurechtzufinden. Deren Teile richten sich nach der Struktur der Spezifikation – beginnend bei der Unterteilung in die Abhandlungen [Aut93] (zum Modul DATENMODELL) und [Ben93] (zum Modul ATOMAR) bis hin zur Gliederung des unkommentierten Quelltextes für ATOMAR im Anhang des letzteren. Schließlich sei noch angemerkt:

- Zur Modularisierung siehe auch Abschnitte 2.3 (Struktur der OBSCURE–Spezifikation) sowie 2.5 und 3.3 – "Schematisches und Spezifisches" – (jeweils zu den Modularisierungsprinzipien).

- Der Quelltext der OBSCURE–Spezifikation ist recht umfangreich (vgl. z.B. [Aut93] mit [Het93]). Dieser Umfang rührt aber nicht her von der sorfältigen Modularisierung, sondern vielmehr wurde umgekehrt letztere gerade wegen dieses Umfangs unumgänglich. Siehe dazu auch 6.2.

6.1.2 Der Algorithmische Sprachstil

Wir betrachten die OBSCURE–Spezifikation der Funktionalen Essenz von HDMS–A und stellen uns folgende Frage:

> Wovon unterscheidet sich algorithmisches Spezifizieren (bzw. die algorithmische Spezifikationsmethode) von strukturiertem Programmieren? Hätte man das alles nicht auch (ohne wesentliche Änderungen!) in einer semantisch wohldefinierten funktionalen Programmiersprache mit Modularisierungskonzept schreiben können?

Selbstverständlich! Nehmen wir also an, man habe eine funktionale Programmiersprache zur Verfügung – und nehmen wir ferner an:

- Sie hat eine wohldefinierte Semantik, die im wesentlichen aus *Algebren* besteht, d.h. aus Trägermengen und Funktionen auf diesen Trägermengen. Programmiersprachentypische Konzepte wie z.B. die *Zustände* der Sprache ML sind darin *nicht* zu finden!!

- Sie hat ein sauberes Modularisierungskonzept, das zumindest über die Konstruktionen Anreicherung (COMPOSE), Kombination (PLUS), Umbenennung (RENAME) und Hiding (FORGET) verfügt.

DANN *kann* man damit algorithmisch spezifizieren – allerdings unter zwei Voraussetzungen:

- Man verwendet die Sprache "richtig" – also im Sinne des *Spezifizierens*, nicht des *Programmierens*. Dies gilt ebenso für OBSCURE selbst und bedeutet z.B., daß Fragen nach Effizienz keinerlei Rolle spielen. Ein konkretes Beispiel ist ein Algorithmus zur Beschreibung einer *Funktion* (!), die Listen sortiert; dieser Algorithmus muß "*einfach*" sein, nicht *effizient*. Siehe dazu auch [Hec93b].

- Man verzichtet auf das Subset– und das Quotient–Konstrukt der Sprache OBSCURE sowie auch auf deren Import– und Export–Axiome. Tatsächlich werden diese Konstrukte in der vorliegenden OBSCURE–Spezifikation nur sehr spärlich benutzt: siehe z.B. die Verwendung des OK–Prädikats im Export–Axiom des Moduls ATOMAR (in dessen Visualisierung zu finden als `<Anwendung des OK-Pr"adikats als Export-Axiom auf:>`).

Wichtig ist uns die Kombination der Idee des Spezifizierens mit dem Begriff des Algorithmus als Beschreibungsmittel.

... weniger wichtig ist uns, daß diese Kombination hier in Gestalt der Spezifikationssprache OBSCURE erfolgt ist und nicht z.B. in Gestalt einer "Programmier"sprache der oben beschriebenen Art. Eine präzise Beschreibung der algorithmischen Spezifikationsmethode findet sich in [Loe87].

6.2 Erkenntnisse über OBSCURE

Eine Beurteilung des Einsatzes der Sprache OBSCURE in Fallbeispielen (insbesondere im Fallbeispiel HDMS–A !) findet sich in [LZ94]. Weitere Überlegungen, die über den Rahmen von OBSCURE und der algorithmischen Spezifikationsmethode hinausgehen, gibt die Abhandlung [Hec93b]. Auch an dieser Stelle soll versucht werden, die Frage nach der Angemessenheit von OBSCURE kurz zu beantworten:

6.2.1 Fehlende Sprachkonzepte

Die Sprache OBSCURE erlaubt weder die Beschreibung höherer Funktionen, noch die Verwendung von Polymorphie noch ist sie lose oder abstrakt. Welche Bedeutung hatte das Fehlen dieser Konzepte bei der Spezifikation der Funktionalen Essenz von HDMS–A ?

1. *Höhere Ordnung*: Funktionen höherer Ordnung sind uns nur an *einer* Stelle begegnet – nämlich bei der Generierung von Schlüsseln. Wie bereits in den Abschnitten 3.3 ("Schematisches und Spezifisches") und 5.1 ("Schlüsselgenerierung") beschrieben, müssen die Funktionen

$$genkeyE : Db \times (KeyE \rightarrow Bool) \rightarrow KeyE$$

(für jeden Entitytyp E) aus [Het93] mittels der *next*– und *new_key*–Funktionen "simuliert" werden.

2. *Polymorphie*: ihr Fehlen stört wesentlich häufiger; Beispiele sind

   - die Einführung von Monolisten: sie muß für jede Elementsorte unter Verwendung des Moduls MK_MONOLIST gesondert durchgeführt werden (gleiches würde natürlich auch für *Mengen* oder *Listen* gelten);
   - die Erweiterung der Attributdomains um die "leeren Attribute" *UNDEF_S*; auch sie muß für jeden Domain gesondert durchgeführt werden.

3. *Losheit, Abstraktheit*: der üblicherweise beim Spezifizieren verwendete axiomatische Sprachstil erlaubt bekanntermaßen nicht nur *loses* Spezifizieren (d.h. die Beschreibung *mehrerer* Modelle), sondern auch *abstraktes* (d.h. die Beschreibung des *WAS* statt des *WIE*). Beide Möglichkeiten fehlen in OBSCURE fast vollständig – eine sehr schwache Version axiomatischen Spezifizierens gestatten in diesem Sinne lediglich die Importaxiome.

   Den Autoren der OBSCURE–Spezifikation erscheint dieser Mangel als der schwerwiegendste. Er führt zur Mißachtung von Prinzipien wie dem der Vermeidung von "overspecification" (wegen der fehlenden Losheit!) und dem der Knappheit von Spezifikationen (wegen der fehlenden Abstraktheit!). In der Praxis des Spezifizierens führt dies zu einem erheblichen (Mehr–)Aufwand, der sich z.B. in der Existenz der sogenannten "Komfortebene" innerhalb der Datenmodellebene niederschlägt. In der Komfortebene müssen explizit Funktionen definiert werden, die dann in der Aktionenebene statt der in [SNM+93] verwendeten Existenzquantoren benutzt werden müssen.

Das Fehlen von Funktionen höherer Ordnung, von Polymorphie und axiomatischen Sprachmitteln führt zum einen zu einem verstärkten Bedarf an OBSCURE–*Schablonen* anstelle (korrekter) OBSCURE–Moduln. Man beachte, daß zwar die Beschreibung der Übersetzungsfunktion *er2obs* ohnehin nicht auf Schablonen verzichten kann, daß wir diese aber darüberhinaus sozusagen zur "Kompensation" der fehlenden Sprachmittel eingesetzt haben.
Es führt zum andern zu einem umfangreicheren Spezifikationstext und zu umfangreicherer Modularisierung. Letzteres übrigens nicht nur wegen des

Umfangs des zu strukturierenden Quelltextes, sondern auch "*direkt*" wegen der oben beschriebenen Mängel. Das Paradebeispiel für diese Phänomene ist die Generierung von Schlüsseln:

- In der SPECTRUM–Spezifikation (vgl. [Het93]) wird diese mittels der *genkeyE*–Funktionen höherer Ordnung in abstrakter und loser Weise behandelt. Insbesondere werden dabei die Formulierung einer *schematischen* Eigenschaft ("der generierte Schlüssel muß neu sein") getrennt von der Formulierung *spezifischer* Eigenschaften. Erstere erfolgt bei der *Definition* der *genkeyE*–Funktionen, letzere bei deren *Anwendung* (mittels des jeweils übergebenen Prädikats).
- In der OBSCURE–Spezifikation muß die Behandlung von Schlüsseln sorgfältig über mehrere Ebenen verteilt werden: innerhalb der Attributebene werden die *next*–Funktionen definiert (und zwar mittels Losheit simulierender Schablonen!), innerhalb der Basisebene werden darauf aufbauend die *new_key*–Funktionen definiert und diese werden schließlich in der Aktionenebene an den entsprechenden Stellen aufgerufen. Alles in allem: mehr Schablonen, mehr Quelltext und mehr "Verteilung" dieses Textes.

Siehe dazu auch den letzten Abschnitt zu Schlüsseln (in 5.1) sowie den Abschnitt "Schematisches und Spezifisches" (in 3.3).
Fazit: auf der Datenmodell– und der Atomaren Ebene ist das Fehlen der eingangs beschriebenen Sprachkonzepte *störend*, aber nicht prinzipiell hinderlich. Für die hier vernachlässigte (und ebenfalls zur Funktionalen Essenz gehörende) Datenflußebene gilt dies allerdings nicht mehr: ein Vorgehen wie in [Nic93] wäre mit OBSCURE *nicht* möglich. Inwieweit es zu diesem Vorgehen "OBSCURE–freundlichere" Alternativen gäbe, konnte leider nicht mehr untersucht werden.

6.2.2 Technische Details

Neben der Untersuchung der im vorangegangen Abschnitt beschriebenen Mängel bzgl. wichtiger Sprachkonzepte hat die Behandlung von Fallbeispielen auch zur Identifizierung und teilweisen Behebung kleinerer "technischer" Unzulänglichkeiten in der Sprache OBSCURE geführt. Als Beispiele seien die Einführung des LET–Konstrukts und des EXTENDED_BY–Konstrukts genannt.
Die schwerwiegendste unter diesen "kleinen" Unzulänglichkeiten ist zweifelsohne das sogenannte "Sortenclash"–Problem. Dieses Problem besteht darin,

daß man ein und dieselbe Sorte nicht an zwei verschiedenen Stellen kreieren kann – auch dann nicht, wenn diese Stellen aus je einem Aufruf ein und desselben Moduls bestehen. Benutzen also zwei verschiedene Moduln einen gemeinsamen Teilmodul NAT zur Spezifikation natürlicher Zahlen, so können sie mit keinem der Sprachkonstrukte von OBSCURE zusammengesetzt werden. Der einzige Ausweg besteht darin, beide Moduln die Sorten und Operationen aus NAT importieren zu lassen (statt NAT zu erweitern), sie dann zuerst untereinander zusammenzusetzen und erst anschließend mit NAT zu verbinden. Man kann sich leicht überlegen, daß dies dem Prinzip der Wiederverwendung völlig zuwider läuft und in der Praxis zu erheblichen Schwierigkeiten führt. Die in 3.4 angesprochenen *Dummies* dienen übrigens der Abmilderung genau dieses Problems.

6.3 Erkenntnisse allgemein

Wozu die Spezifikation in OBSCURE?

> Wir haben OBSCURE benutzt, um festzustellen, daß man es nicht benutzen sollte — daß man aber davon lernen sollte.

Lernen kann man von OBSCURE algorithmisches Spezifizieren. Der algorithmische Sprachstil hat nämlich durchaus nicht nur die – hinreichend geschilderten – Nachteile, sondern auch folgende Vorteile:

1. Bei vielen Gelegenheiten ist er ganz einfach der *angemessene* Stil: beispielsweise sehen sich auf der Aktionenebene die SPECTRUM– und die OBSCURE–Spezifikation sehr ähnlich. Diese Ebene läßt sich weitgehend algorithmisch beschreiben. Ähnliches gilt für die gesamte Datenmodellebene – auf Ausnahmen wurde bereits eingegangen.

2. Er begünstigt naturgemäß die Ausführbarkeit von Spezifikationen. Zwar scheint das im vorliegenden Beispiel wegen der Existenz ausgefeilter Programm–Generatoren für E/R–Schemata nicht relevant – was aber ist mit der auf dem Datenmodell aufsetzenden Atomaren Ebene? Wie soll hier "rapid prototyping" betrieben werden? Dasselbe gilt übrigens auch für die Spezifikation einer Benutzungsoberfläche.

3. Schließlich kann man auch in einem ganz anderen Sinne von OBSCURE lernen: dem "Einsteiger" in die Welt des Spezifizierens wird der Übergang vom *Programmieren* zum *Spezifizieren* gerade durch den algorithmischen Sprachstil erleichtert – wozu auch die (ansonsten eher nachteilige!) Einfachheit der Sprache beitragen dürfte.

Nicht benutzen sollte man OBSCURE über Übungszwecke hinaus – jedenfalls nicht vor einer Behebung der in Abschnitt 6.2 geschilderten Mängel.

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.

[BFG+93] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, and K. Stølen. The Requirement and Design Specification Language SPECTRUM – An Informal Introduction, Version 1.0. Technical report, Technische Universität München, 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.

[Hec93a] 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.

[Hec93b] Ramses A. Heckler. Beobachtungen zur algebraischen Spezifikationsmethode – Mit Bezügen zu modellbasierter und algorithmischer Methode. (37 Seiten). Interner Bericht, September 1993. Universität des Saarlandes.

[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–Relationsship–Modells*. TEUBNER–TEXTE zur Informatik. B.G. Teubner Verlagsgesellschaft, 1993.

[Hus93] Heinrich Hussmann. Synergy Between Formal and Pragmatic Software Engineering Methods, 1993. München.

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

[LL93] Thomas Lehmann and Jacques Loeckx. Obscure, A Specification Language for Abstract Data Types. *Acta Informatica*, 30(FASC.4):303–350, 1993.

[Loe87] Jacques Loeckx. Algorithmic Specifications: A Constructive Specification Method for Abstract Data Types. *TOPLAS*, 9(4):646–685, 1987.

[LZ94] Jacques Loeckx and Jörg Zeyer. Experiences with the Specification Environment *OBSCURE*. *LNCS*, ???, 1994. !! UNPUBLISHED !!, TO APPEAR.

[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.

[NW93] F. Nickl and M. Wirsing. A Formal Approach to Requirements Engineering. In *Proceedings of the International Symposium on Formal Methods in Programming and their Applications, Novosibirsk*. Springer LNCS, July 1993. Also appeared as technical report no. 9314 at the Ludwig-Maximilians-University München.

[Reg93] Franz Regensburger. An extended abstract for the specification language SPECTRUM. Noch nicht veröffentlicht, März 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.

[vL90] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*, chapter M. Wirsing: Algebraic Specification, pages 675–788. North–Holland, 1990.