Eine Fallstudie zur Spezifikation von Systemanforderungen in der Spezifikationssprache OBSCURE

Diplomarbeit von Christoph Benzmüller

Angefertigt nach einem Thema von Prof. Dr.-Ing. Loeckx am Fachbereich 14,
Informatik, der Universität des Saarlandes, Saarbrücken.

Diese Diplomarbeit wurde erstellt in Zusammenarbeit mit dem
Fraunhofer-Institut Biomedizinische Technik (IBMT) in St. Ingbert
unter der Leitung von Prof. Dr. med. Gersonde.

5. Juli 1994

Hiermit erkläre ich an Eides Statt, daß ich diese Arbeit nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Saarbrücken, den 5. Juli 1994

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation einer formalen Software-Entwicklung

In den 50er und 60er Jahren wurden Softwaresysteme typischerweise zur Lösung mathematischer Probleme, wie z. B. zum Lösen einer Differentialgleichung, eingesetzt. Die damaligen Problemstellungen waren meist überschaubar und konnten relativ leicht in einer geeigneten Programmiersprache implementiert werden. Spezielle Konzepte, Techniken, Werkzeuge und Sprachen waren dazu nicht nötig. Erst in der Folgezeit, als zunehmend komplexere Softwaresysteme geplant und erstellt wurden (z. B. in den 70er Jahren zur Steuerung der Mondfahrt), stand man den Problemstellungen öfter hilflos gegenüber. Die geplanten Systeme wurden mit der Zeit so komplex, daß es einer geeigneten Zerlegung in einzelne Komponenten und der Abstraktion von Details bedurfte, um diese als Ganzes zu überschauen. Weil die Systeme wegen ihrer Größe nicht mehr von einzelnen Personen erstellt werden konnten, mußten mehrere in den Systementwicklungsprozeß integriert und die Aufgaben entsprechend verteilt werden. Während D. Knuth (siehe [Knu81]) die Entwicklung großer Softwaresysteme als eine künstlerische Tätigkeit ansah, besannen sich andere auf die Tugenden ingenieurwissenschaftlicher Bereiche. Dort hatte man für komplexe Aufgabenstellungen (z. B. dem Bau einer Brücke, eines Hochhauses oder eines Flugzeuges) geregelte Vorgehensweisen entwickelt, die eine Lösung vereinfachten oder erst möglich machten und eine objektive Beurteilung der Qualität des fertigen Produktes erlaubten. Man erkannte die Parallelen zu diesen ingenieurwissenschaftlichen Bereichen und begann, eigene Vorgehensweisen, Werkzeuge und Sprachen zur Entwicklung und Beschreibung großer Softwaresysteme zu entwerfen (siehe z. B. [DeM78], [Jac83], [YC79], [GM86], [FW83], [Jon90], [You89] und [Bud93]).

In der Praxis haben sich solche standardisierten Vorgehensweisen und die Verwendung bestehender Werkzeuge und Sprachen heute vielerorts durchgesetzt. Zwar läßt sich nicht bestreiten, daß die Entwicklung eines Softwaresystems auch große kreative Aspekte beinhaltet (dies ist auch beim Brücken-, Hochhaus- oder Flugzeugbau der Fall), doch kann diese Tätigkeit heute insgesamt nicht mehr als eine künstlerische, sondern muß eher als eine ingenieurwissenschaftliche Tätigkeit ange-

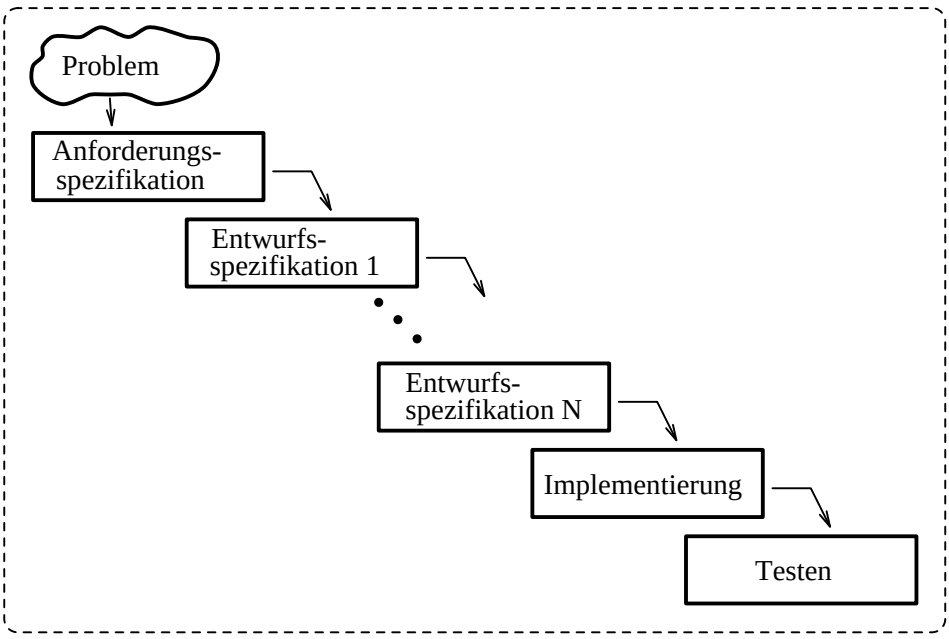

Abbildung 1.1: Wasserfallmodell

sehen werden.

Den meisten bestehenden Vorgehensweisen zur Software-Entwicklung gemein ist die konzeptionelle Aufteilung der Entwicklungstätigkeiten in mehrere Phasen. Abbildung 1.1 beschreibt ein solches Phasenmodell zur Entwicklung großer Softwaresysteme. Dieses vereinfachte und idealisierte Modell wird in der Literatur oft als **Wasserfallmodell** bezeichnet. Es geht aus von einer stufenweisen Umsetzung einer Problemstellung über mehrere Phasen, die sich in ihrem Abstraktionsniveau unterscheiden, hin zu einer Implementierung in einer (oder mehreren) Programmiersprachen. Zunächst werden dabei die Anforderungen an das zu erstellende System spezifiziert, d. h. möglichst präzise beschrieben. Danach wird ein Systementwurf ausgearbeitet, der dann direkt oder über weitere Entwurfsebenen in eine Implementierung des Systems umgesetzt wird. Der Übergang von einer höheren Ebene zu einer niedrigeren wird als Verfeinerungsschritt bezeichnet. Den letzten Übergang, hin zur Implementierung, nennt man auch Implementierungsschritt. Ein Testen der Implementierung gegenüber der Problemstellung und der Anforderungsspezifikation, soll dann das Vertrauen in die Validität[1] und die Korrektheit[2] des Systems vertiefen. Es hat sich herausgestellt, daß dieses idealiserte Top-Down-Vorgehensmodell in der Praxis kaum anwendbar ist. Zyklen, d. h. mehrfache Wiederholungen einzelner Entwicklungsphasen, sind meist notwendig, um ein System letztendlich erfolgreich

[1] Zur Überprüfung der Validität eines Softwaresystems stellt man sich die Frage: Wurde das *richtige System erstellt*, also, wurde die Problemstellung richtig erkannt und umgesetzt?

[2] Zur Überprüfung der Korrektheit stellt man sich die Frage: Wurde das *System richtig erstellt*, also, hat man im Entwurfs- bzw. Implementierungsprozeß keine Fehler gemacht?

zu implementieren. Es wurden dazu entsprechende Vorgehensmodelle entwickelt, die das idealisierte Wasserfallmodell ersetzen oder erweitern. Ein Beispiel ist das Spiralmodell nach Boehm (siehe [Boe88]). Zur Darstellung der Ergebnisse der Anforderungs- und Entwurfsphasen verwendet man heute in der Praxis meist informale oder semiformale Beschreibungsmittel [3]. Nachteil von informalen oder semiformalen Beschreibungen ist, daß diese Mehrdeutigkeiten enthalten können, die möglicherweise Mißverständnisse im Entwicklungsprozeß zur Folge haben. Aufgrund der fehlenden präzisen Semantik können auch keine mathematischen Beweise, z. B. zur Korrektheit eines Verfeinerungsschritt, geführt werden.

Um diesen Nachteil beheben, beschäftigt man sich seit einigen Jahren in Forschung und Praxis mit der Entwicklung formaler Spezifikationssprachen zur Beschreibung der Ergebnisse der Anforderungs- und Entwurfsphasen. Spezifikationen in einer formalen Sprache sollen informale und semiformalen Beschreibungen ergänzen oder ersetzen. Mehrdeutigkeiten, die bisher auftreten konnten, werden durch die präzise, formale Semantik dieser Sprachen ausgeschlossen. Außerdem eröffnet sich bei einer formalen Software-Entwicklung — d. h. bei einer Software-Entwicklung mit Einbezug formaler Spezifikationssprachen zur Beschreibung der Ergebnisse der einzelnen Entwicklungsebenen — prinzipiell die Möglichkeit zur Verifikation. Die Korrektheit der einzelnen Verfeinerungsschritte oder die Gültigkeit eines ebenfalls formal beschriebenen Sicherheitsmodells kann formal bewiesen werden. Probleme beim Beweis der Korrektheit des letzten Schrittes (dem Implementierungsschritt), können dadurch vermieden werden, daß der Programmcode automatisch oder halbautomatisch durch Programmtransformation (siehe z. B. [Mee86]) aus der letzten und datailliertesten Entwurfsspezifikation generiert wird. Ein anderes Problem, nämlich das der Validierung der Anforderungspezifikation gegenüber der Problemstellung, kann (unter bestimmten Bedingungen) durch ein Rapid-Prototyping unterstützt werden [4]. Insgesamt kann das bisher erforderliche Testen der Implementierung also größtenteils durch formale Beweisführungen und/oder ein Rapid-Prototyping ersetzt werden.

Allgemeine Einführung in das Thema formale Spezifikation bieten z. B. [EM85] und [EM90] oder auch [LEW].

Beispiele für formale Spezifikationssprachen bzw. Spezifikationsmethoden sind: VDM (siehe [Jon90]), Z (siehe [PST91], [Woo93]), CLEAR (siehe [BG80]), ASL (siehe [SW83]), PLUSS (siehe [Bid91]), SPECTRUM(siehe [BFG+91]) und OBSCURE (siehe [LL93]).

Zwei aktuelle Projekte zur Untersuchung und Weiterentwicklung von Methoden und Werkzeugen zur formalen Software-Entwicklung sind: KORSO (siehe [Be94], und [BJ94]) und VSE (siehe [BCC+93]).

Diese Arbeit hat das Hauptziel, anhand eines Fallbeispiels die erste Phase einer formalen Software-Entwicklung zu untersuchen. Untersuchungsgegenstand ist das

[3] Zum Beispiel Entity-Relationship- und Datenflußdiagramme, siehe auch Seiten 18 bis 20, insbesondere auch Fußnote 3 auf Seite 18.

[4] Weitere Erläuterungen zu den Vorteilen einer formalen Spezifikation werden auf Seite 53 vorgestellt. Auf das Thema Rapid-Prototyping geht diese Arbeit in Abschnitt 3.5 ein.

am Fraunhofer-Institut Biomedizinische Technik (IBMT) in St. Ingbert in der Entwicklung befindliche System SUNRISE (siehe [BSSG93]). Zu diesem System (bzw. zu einem Teil des Systems) soll zunächst — gemäß des idealisierten Vorgehens nach dem Wasserfallmodell — eine informale/semiformale Anforderungspezifikation ausgearbeitet werden, um dann aufzuzeigen, wie diese Beschreibung durch eine formale Spezifikation in der Spezifikationssprache OBSCURE (siehe [LL93]) ergänzt werden kann. Weitere Aspekte der Arbeit betreffen die Themen Rapid-Prototyping, Beziehung der erstellten Anforderungspezifikation zur aktuellen Implementierung und Gütezertifikate zur Sicherheit von Softwaresystemen. Ein Beispiel zu einer formalen Beweisführung konnte in dieser Arbeit nicht durchgeführt werden. Die Gründe hierfür werden auf Seite 105 erläutert.

Eine ausführliche Beschreibung der Aufgabenstellung ist durch [Ben93a] und [Ben93b] gegeben.

1.2 OBSCURE

1.2.1 Die Spezifikationssprache OBSCURE

Die algorithmische **Spezifikationssprache** OBSCURE[5] wurde gegen Ende der 80er Jahre am Lehrstuhl von Prof. Dr.-Ing. Loeckx entwickelt. Im Gegensatz zu den abstrakten Spezifikationen in einer axiomatischen Spezifikationsprache — in einer solchen wird die Semantik von Sorten und Operationen nur durch die Angabe von Eigenschaften festgelegt —, wird in OBSCURE die Semantik einer Sorte definiert als die Termsprache über den explizit anzugebenden Konstruktoren dieser Sorte. Zusätzliche Operationen werden durch rekursive Programme beschrieben. Die algorithmische Spezifikationsmethode (siehe [Loe87]) ist damit weit weniger abstrakt als die axiomatische bzw. algebraische (siehe [EM85], [EM90], [EGL89], [LEW] und [vL90]). Vor allem ist es nicht möglich, eine Operation nur durch ihre Eigenschaften (z. B. Vor- und Nachbedingungen) zu charakterisieren, sondern der Spezifizierer wird gezwungen, einen rekursiven Algorithmus anzugeben. Ein Vorteil dabei ist, daß dadurch ein Rapid-Prototyping möglich wird, bei dem vom Benutzer angegebene Terme zu Konstruktortermen ausgewertet werden. In diesem Sinne kann man eine algorithmische Spezifikationssprache also durchaus mit einer Programmiersprache vergleichen. Der abstrakte Charakter einer rein algorithmischen Spezifikation besteht darin, daß man den zur Charakterisierung einer Operation angegeben Algorithums nicht als Implementierung auffaßt, sondern als eine Beschreibung ihrer logischen Eigenschaften.

Für OBSCURE gelten die erwähnten Beschränkungen einer algorithmischen Spezifikationssprache nicht absolut: Durch die Verwendung von Importaxiomen kann prinzipiell axiomatisches Spezifizieren simuliert werden, allerdings mit dem entschei-

[5] Die reine Spezifikationssprache OBSCURE dient der Verknüpfung atomarer Spezifikationen und ist unabhängig von der Spezifikationsmethode, die in diesen atomaren Spezifikation angewendet wurde. Im OBSCURE-System wird die Spezifikationssprache OBSCURE allerdings nur im Zusammenhang mit der algorithmischen Spezifikationsmethode verwendet.

denden Nachteil, daß ein Rapid-Prototyping dann nicht mehr möglich ist. Auch die in OBSCURE erlaubte Bildung von Teil- oder Quotientenalgebren verläßt den Rahmen rein algorithmischen Spezifizierens.

Eine Beschreibung der Sprache OBSCURE ist gegeben durch [LL93]. Weitere Erläuterungen finden sich auch in [FHM+91], [Zey92] und [Leh90]. In dieser Arbeit wird weder die Sprache OBSCURE, noch das OBSCURE-System näher vorgestellt.

1.2.2 Die Spezifikationsumgebung OBSCURE-System

Das OBSCURE-System (siehe [FHM+91]) unterstützt als **Spezifikationsumgebung** die Erstellung, Verwaltung und Weiterbearbeitung von Spezifikationen. Das OBSCURE-System besteht aus einem Parser, einer Moduldatenbank, einem Interpreter, mehreren Übersetzern, einem Pretty-Printer und einem Visualisierungswerkzeug.

Der Parser (siehe [Zey89]) führt die syntaktische Analyse von Spezifikationen durch und überprüft Kontextbedingungen.

Erfolgreich geparste Spezifikationen werden durch die Moduldatenbank (siehe [Mei89]) benutzerspezifisch verwaltet. Eine Standardmoduldatenbank enthält zusätzlich solche Spezifikationsmodule, die allen Benutzern standardmäßig zur Verfügung stehen.

Der Interpreter (siehe [Sto91]) ermöglicht ein Rapid-Prototyping. Terme können dazu vom Benutzer eingegeben und dann vom Interpreter zu Konstruktortermen ausgewertet werden.

Derzeit wird eine Übersetzung von OBSCURE-Spezifikationen in die funktionale Programmiersprache ML (siehe [JWS94]), nach C++ und in die Eingabesprache des Verifikationssystems INKA[6] (siehe [Lat92]) unterstützt. Nach ML und INKA ist derzeit allerdings nur die Übersetzung atomarer, d. h. nicht strukturierter, Spezifikationen möglich.

Die Ausgabe von Spezifikationstexten und von Import-/Exportsignaturen der Spezifikationen erfolgt durch den Pretty-Printer.

Die Modulstruktur großer Spezifikation kann mit dem Visualisierungstool VIRUS (siehe [RM92a], [RM92b] und [RM92c]) übersichtlich dargestellt werden. Derzeit ist allerdings nur eine Ausgabe auf einen Bildschirm möglich.

Die Anbindung des OBSCURE-Systems an den Emacs-Editor, der als Eingabeoberfläche dient, ist in [Hof89] beschrieben.

1.3 SUNRISE

Das am Fraunhofer-Institut Biomedizinische Technik (IBMT) in St. Ingbert in der Entwicklung befindliche System SUNRISE (siehe [BSSG93]) dient der Verwaltung,

[6] INKA (siehe [BHHW86]) ist ein Beweissystem der Prädikatenlogik erster Stufe mit Induktion. Es wurde ursprünglich in Karlsruhe entwickelt und wird derzeit, innerhalb des VSE-Projekts, am Deutschen Forschungszentrum für Künstliche Intelligenz (DFKI) in Saarbrücken eingesetzt und weiter ausgebaut.

Visualisierung (Stichwort: Digitale Lichtbox) und Weiterbearbeitung von medizinischen Untersuchungsdaten (z. B. Röntgendaten). Außerdem ermöglicht SUNRISE die Entwicklung und Integration neuer Datenbearbeitungsoperationen (Beispiel ist ein sogenannter Graustufenfilter oder eine Fourier-Transformation), sowie neuer Datenarten[7] zur Laufzeit. Untersuchungsdaten, die an unterschiedlichen medizinischen Analysegeräten entstehen, werden beim Import in das System einer Datenart zugeordnet. Dadurch wird festgelegt, welche Bearbeitungsoperationen des Systems auf die importierten Daten anwendbar sind und welche nicht. Dies erfolgt, um z. B. die erforderliche, logisch unterschiedliche Handhabung binärer Bilddaten von Spektren zu realisieren. Dem Anspruch einer medizinischen Datenbank trägt das System Rechnung, indem es die importierten Untersuchungsdaten verwaltet in Bezug zu Aufenthalts-, Patienten- und Hospitalinformationen. Die einmal importierten Daten können visualisiert, d. h. auf einen Bildschirm ausgegeben werden, oder auch weiterbearbeitet werden mit den zur Verfügung stehenden Bearbeitungsoperationen. Ein Beispiel ist: Röntgendaten zum fiktiven Patienten H. M., dessen Patienten- und Aufenthaltsdaten zuvor in der Systemdatenbank angelegt wurden, werden in das System importiert, als Datenart *image* deklariert und den Aufenthalts- und Patientendaten von H. M. zugeordnet. Ein Arzt hat im folgenden die Möglichkeit, sich die Röntgendaten zum Patienten H. M. mit den Datenbank-Verwaltungsoperationen aus der Datenbank auszuwählen und auf einen Bildschirm ausgeben zu lassen. Möglicherweise hat er den Wunsch, die Röntgendaten mit einer Datenbearbeitungsoperation weiterzubearbeiten, um so wesentliche Informationen der Röntgenaufnahme von unwesentlichen zu trennen und sich dann die weiterbearbeiteten Röntgendaten auf dem Bildschirm anzuschauen. Auf diese Weise erzeugte Untersuchungsdatenobjekte können auf Wunsch aus dem lokalen Speicherbereich jedes Systembenutzers in den weiterbearbeitete Datenobjekte zunächst aufgenommen werden, in das globale Datenarchiv übertragen werden. Weitere, arbeitserleichternde Optionen stehen zur Verfügung: Zum Beispiel ein Undo/Redo-Mechanismus zum Hin- und Herblättern zwischen den Resultaten von Weiterbearbeitungsschritten und die Möglichkeit zur Definition von Makrooperationen (Verknüpfung einzelner Bearbeitungsoperationen).

Neben diesen Systemeigenschaften, die besonders für den routinemäßigen Einsatz in einem Hospital vorgesehen sind, bietet das SUNRISE-System die Möglichkeit zur Entwicklung und Einbindung neuer Datenbearbeitungsoperationen zur Laufzeit. Dazu stellt es dem Entwickler eine Bibliothek von Systemschnittstellenfunktionen zur Verfügung. Auch die Definition und Einbindung neuer Datenarten zur Laufzeit ist vorgesehen. Somit kann das System ständig weiterentwickelt und der jeweiligen Systemumwelt angepaßt werden.

SUNRISE wurde zwar primär für den Einsatz in medizinischem Umfeld konzipiert, ist aber nicht auf dieses beschränkt.

[7] Die Datenart eines Untersuchungsdatenobjekts (z. B. Röntgenbild) gibt einen Hinweis auf die logischen Eigenschaften dieser Daten. Beispielsweise bedürfen Spektren einer anderen logischen Handhabung als Röntgendaten.

1.4 Überblick zu den einzelnen Kapiteln

Kapitel 2 beschäftigt sich mit der Systemanalyse zum SUNRISE-System. Ausgangspunkt ist die Auswahl einer Vorgehensweise zur Durchführung einer Systemanalyse. Zu dieser Arbeit ausgewählt wurde die Strategie der *Strukturierten Systemanalyse* nach [MP88]. Die Ergebnisse der Anwendung dieser Strategie auf das SUNRISE-System — dies sind semiformale Beschreibungen der *logischen* Systemanforderungen durch Entity-Relationship- und Datenflußdiagramme — werden in diesem Kapitel vorgestellt. Ferner wird eine typische Aussage zur Sicherheit (Datenintegrität) einer medizinischen Datenbank ausgearbeitet.

In Kapitel 3 werden die semiformalen Beschreibungen der logischen Anforderungen an das SUNRISE-System aus Kapitel 2 als Ausgangspunkt zur Erstellung einer formalen Spezifikation in der Spezifikationssprache OBSCURE verwendet. Dabei findet u. a. ein, im Rahmen der Fallstudie KORSO-HDMS-A (siehe [CHL94], [CHL94] und [CKL93]) entwickeltes, Transformationsschema (siehe [Aut93] und [Het93]) von Entity-Relationship-Diagrammen in OBSCURE-Spezifikationen Anwendung. Insgesamt wird versucht, die Vorgaben der semiformalen Beschreibungen genau zu berücksichtigen und einen fließenden Übergang von semiformalen Beschreibungen zu formalen Spezifikationen zu demonstrieren. Dabei wird auch die exemplarische Aussage zur Sicherheit des SUNRISE-Systems berücksichtigt und formal spezifiziert. Außerdem geht dieses Kapitel auf das Thema Rapid-Prototyping ein. Dazu wird ein Auszug einer Interpretersitzung mit dem OBSCURE-System vorgestellt und erläutert. (Einzelne Teile dieses Kapitels sind in den Anhang ausgegliedert.)

Kapitel 4 versucht, die Beziehung zwischen der aktuellen Implementierung des SUNRISE-Systems und der, in dieser Arbeit vorgestellten, *logischen Essenz* zu beschreiben. Zunächst werden dazu einzelne Entwurfs- bzw. Implementierungsentscheidungen anhand eines Beispiels erläutert, um dann die Bestandteile der logischen Essenz möglichst präzise auf die Implementierung abzubilden.

Kapitel 5 stellt zunächst eine Übersicht zu den Informationstechnologie-Sicherheitskriterien [ITS91] vor. Anschließend wird diskutiert, welche Sicherheitskriterien durch diese Arbeit abgedeckt werden bzw. welche nicht.

Schließlich faßt Kapitel 6 die Diplomarbeit kurz zusammen und präsentiert wichtige Anmerkungen bzw. Ergebnisse aufgeteilt nach den Themengebieten: Systemanalyse, SUNRISE und OBSCURE. Desweiteren wird erläutert, warum in dieser Arbeit keine formale Verifikation durchgeführt werden konnte. Letztlich wird ein Vorschlag zur Fortführung der Fallstudie vorgestellt und diskutiert, ob eine Fortführung auch sinnvoll ist.

1.5 Anmerkungen zur Form

Die Konventionen zur Form, die dieser Arbeit zugrunde liegen, werden nun kurz erläutert:

- Konventionen für den allgemeinen Text
  - Wichtige Begriffe im laufenden Text werden durch **Fettdruck** vom übrigen Text abgesetzt. Insbesondere in Kapitel 2 wird davon Gebrauch gemacht.
  - Sorten-, Variablen- und Operationsnamen der formalen Spezifikation, die im laufenden Text genannt werden, erscheinen in der Schriftart *Slanted* (Ausnahme: die Operation WSAV, diese wird wegen ihrer besonderen Bedeutung im Schriftstil SMALL CAPS notiert.).
  - Systemnamen und Projektbezeichnungen erscheinen meist in der Schriftart SMALL CAPS.

- Umlaute in den Abbildungen und Spezifikationstexten
  Die Abbildungen wurden mit dem Zeichenprogramm XFIG erstellt. Dieses Programm ermöglicht nicht die Darstellung von Umlauten. Anstelle von *ä*, *ö*, *ü* und *ß* wird in Abbildungen also *ae*, *oe*, *ue* und *ss* verwendet. Auch in den Spezifikationstexten der Sprache OBSCURE können keine Umlaute verwendet werden. Sorten- und Operationsbezeichnungen der Spezifikation, die im laufenden Text genannt werden, erscheinen deshalb ebenfalls ohne Umlaute.

  - Literarisches Spezifizieren
    Diese Arbeit verfolgt den Ansatz des Literarischen Spezifizierens. Es soll kurz erläutert werden, was diese Arbeit darunter versteht: Besonders in der Programmierpraxis, werden Quelltexte oft nur spärlich kommentiert. Systemdokumente enthalten meist, wenn diese überhaupt auf Quelltexte eingehen, einfache Abschriften der Quellen, deren Aktualisierung bei Änderungen des Systems häufig vergessen wird. In der vorliegenden Arbeit ist dies nicht der Fall. Sämtliche Spezifikationstexte die hier abgebildet werden, sind identisch mit denen, die in das OBSCURE-System eingelesen wurden, weil in dieser Arbeit die Originalquelltexte der Spezifikation gleichsam als LaTeX[8] -Quellen dienen. Die Quelltexte enthalten beides: den reinen Spezifikationstext und die Erläuterungen, die in der vorliegenden Arbeit zu sehen sind[9]. Durch Anwendung zweier Programme[10], kann man aus einem solchen Spezifikationsquelltext einerseits eine LaTeX-Datei erzeugen und diese, wie hier geschehen, in ein Dokument einbinden, andererseits eine OBSCURE-Datei erzeugen, die dem OBSCURE-Parser übergeben werden kann. Geht man von der Korrektheit der beiden Übersetzungsprogramme aus, so kann man sich sicher

[8] Mit dem Textverarbeitungsprogramm LaTeX wurde diese Arbeit erstellt.

[9] Hauptsächlich aus Umfangsgründen wird in den Spezifikationsquelltexten zur Spezifikation der Datenmodellebene auf ausführliche Kommentare verzichtet. Hier werden kurze Erläuterungen im sonstigen *normalen* Text angegeben.

[10] Diese beiden Programme wurden von Ramses A. Heckler, einem der Betreuer dieser Arbeit, zur Verfügung gestellt, dem dafür mein Dank gebührt. Das momentan am Lehrstuhl von Prof. Dr.-Ing. Loeckx in Entwicklung befindliche System OPTICAL, das ein Literarisches Spezifizieren in OBSCURE weiter vereinfachen soll, konnte noch nicht eingesetzt werden.

sein, daß die in dieser Arbeit abgebildeten Spezifikationstexte, ständig die aktuellen Versionen wiedergeben.
Die Teile dieser Arbeit, die aus einem solchen Spezifikationsquelltext erzeugt wurden, können vom übrigen Text unterschieden werden. Sie beginnen mit einem Kommentar '*Modul: <Modulname>*' und enden mit einem Kommentar '*Ende Modul <Modulname>*'. Der gesamte Kommentartext eines solchen Teils ist durch die Schriftart *Italic* vom übrigen Text und auch vom reinen Spezifikationstext in der Sprache OBSCURE optisch abgesetzt. In diesen Kommentartexten werden Sorten- und Operationsnamen der OBSCURE-Spezifikation nicht weiter hervorgehoben.

- Kommentare zur Interpretersitzung
  In Anhang B findet sich ein kommentierter Auszug einer Interpretersitzung. Auch dort erscheint der Kommentartext in der Schriftart *Italic*.
- Namenskonventionen der formalen Spezifikation
  * Sortennamen werden im allgemeinen mit großen Anfangsbuchstaben geschrieben (z. B. Sunrise-Datenobjekt).
  * Bis auf wenige Ausnahmen beginnen Operationsnamen mit einem kleinen Buchstaben (z. B.: get-Sunrise-Datenobjekt).
  * Variablen werden klein geschrieben (z. B.: x).
  * Modulnamen werden in Großbuchstaben notiert (z. B.: SCHNITTSTELLENMODUL).
  * Die Sprachkonstrukte von OBSCURE heben sich durch **Fettdruck** und auschließliche Verwendung von Großbuchstaben vom übrigen Spezifikationstext ab (z. B.: **X_COMPOSE**).

Danksagung

Ich möchte mich besonders bei meinen beiden Betreuern Roland Brill und Ramses A. Heckler bedanken für die zahlreichen und nützlichen Diskussionen, insbesondere zum Thema Systemanalyse. Sie standen mir stets als Ansprechpartner zur Verfügung und leisteten hilfreiche Unterstützung beim Korrekturlesen dieser Arbeit. Ferner gilt mein Dank Prof. Dr.-Ing. Loeckx und Prof. Dr. med. Gersonde für die interessante Themenstellung. Sehr hilfreich für die fristgemäße Anfertigung dieser Arbeit war auch die Unterstützung durch das IBMT in St. Ingbert. Hervorzuheben sind die Gewährung einer ständigen Zugangsmöglichkeit zu den Räumlichkeiten und den Rechnern, sowie das freundliche Entgegenkommen der Mitarbeiter. Bei meiner Verlobten Nicole Clemens möchte ich mich für das lästige Korrekturlesen unter Zeitdruck bedanken und bei Jörg Zeyer für seine Ablenkungsmanöver im Rechnerraum, die mich manchmal auch auflockerten und vor einer zu verbissenen Arbeitshaltung bewahrten.

Kapitel 2

Systemanalyse

Dieses Kapitel beschäftigt sich mit der Analyse der Anforderungen an das SUNRISE-System. Der Begriff *Anforderungen* bezeichnet dabei diejenigen geplanten Systemeigenschaften, die unabhängig von einer bestimmten Implementierungstechnologie existieren. Eine Anforderungsanalyse ist ein wichtiger erster Schritt, der einer formalen Anforderungspezifikation vorausgehen sollte. Nur bei kleineren Problemstellungen oder leicht überschaubaren Systemen, kann bereits ein *intensives Nachdenken* als Grundlage für eine formale Anforderungsspezifkation genügen. Auch in der Praxis der Software-Entwicklung gewinnt eine sorgfältige Systemanalyse zunehmend an Bedeutung. Im Falle einer Neuimplementierung eines Systems auf veränderter Technologie, für Wartungsaufgaben, das Heranführen neuer Mitarbeiter an ein, für diese unbekanntes, System oder zu Dokumentationszwecken ist eine technologieunabhängige, abstrakte Beschreibung wünschenswert. Auch die Tatsache, daß (Computer-)Systeme zunehmend komplexer werden und oft von einzelnen Personen kaum überschaut werden können, unterstreicht die Bedeutung einer verständlichen und von unwesentlichen Details befreiten Beschreibung eines Systems. In der Praxis wurden dazu Methoden und Werkzeuge (siehe z. B. [You89], [MP88], [Bal92]) entwickelt. Ein wichtiger Aspekt dabei ist, die Systemanalysetätigkeit möglichst unabhängig von den Fähigkeiten einzelner Personen zu gestalten und die Ergebnisse der Analysetätigkeiten angemessen darzustellen. Auf jeden Fall beinhaltet eine Systemanalysetätigkeit einen nicht zu unterschätzenden Anteil an Kreativität. Zum einen lassen sich die Systemziele nicht immer exakt bestimmen, was zu unterschiedlichen Anforderungsprofilen führen kann, zum anderen fehlen oft geeignete Sprachmittel, um das Ergebnis einer Systemanalyse verständlich und dennoch frei von Implementierungsdetails beschreiben zu können. Dies führt dazu, daß zur Beschreibung dann doch häufig einfache Detailentscheidungen getroffen werden. Auch die vorliegende Arbeit ist von dieser Problematik betroffen; auf Seite 28 wird dies näher erläutert.

Innerhalb dieser Diplomarbeit konnte die Systemanalyse auf der Grundlage bereits existierender Systemdokumente durchgeführt werden. Allerdings standen keine, von Implementierungsdetails befreiten, Systemdokumente zur Verfügung, die einen Großteil der Analysetätigkeit überflüssig gemacht hätten. Die zur Zeit verfügbaren Dokumente haben leider den Nachteil, daß sie zwar sehr ausführlich

sind, aber viele Systemeigenschaften in Bezug auf eine bestimmte Implementierungstechnologie beschreiben. Das heißt, grobe Entwurfsentscheidungen werden gemeinsam mit feinen Detailentscheidungen erläutert. Aus diesem Grund war eine erneute Systemanalyse auf der Grundlage der existierenden Dokumente und der aktuellen Implementierung als Einstieg in die vorliegende Arbeit unvermeidbar. Auch konnte sich die Systemanalyse nicht auf den zur weiteren Untersuchung ausgewählten Ausschnitt **duales Speicherkonzept** alleine konzentrieren, weil dadurch wichtige Anforderungen, von anderen Systemkomponenten an diesen Ausschnitt, womöglich übersehen worden wären. Eine Fokussierung auf den Ausschnitt **duales Speicherkonzept** fand allerdings insofern statt, daß dieser mit besonderer Sorgfalt analysiert wurde, während die übrigen Ausschnitte nur im Hinblick auf mögliche Auswirkungen auf das **duale Speicherkonzept** untersucht wurden. Auf Probleme und eine subjektive Einschätzung der Qualität der Systemanalyse geht Abschnitt 6.2 noch näher ein.

Die Vorgehensweise der, innerhalb dieser Diplomarbeit durchgeführten, Systemanalyse ist angelehnt an ein Werk von Stephen M. McMenamin und John F. Palmer (siehe [MP88]). Die Autoren stellen eine Strategie zur **Strukturierten Systemanalyse (Structured Analysis)** vor, die eine recht eindeutige Bestimmung der *wahren Anforderungen* an ein System ermöglichen und die sich in der Praxis bereits mehrfach bewährt haben soll. Einige der in [MP88] eingeführten Begriffe werden im folgenden kurz vorgestellt. Dies erfolgt zum einen, weil dem Leser dadurch ein intuitives Verständnis zum Begriff der **Systemanforderungen** vermittelt wird, und zum anderen, weil diese Arbeit ein grobes Verständnis einiger Begriffe vorraussetzt. Ebenfalls kurz erläutert werden die Beschreibungsmittel Entity-Relationship-Diagramm (E/R-Diagramm) und Datenflußdiagramm (DFD-Diagramm), bevor dann das Ergebnis der Systemanalyse präsentiert wird.

2.1 Strategie, Begriffe und Beschreibungsmittel

In diesem Abschnitt werden wichtige Begriffe definiert[1] und die angewendete Strategie aus [MP88] vorgestellt. Eine ausführlichere Erläuterung der Begriffe und der Vorgehensweise kann in [MP88] nachgeschlagen werden.

Zunächst wird der Begriff **perfekte Technologie** definiert. Auf diesem baut dann die Definition der **logischen Essenz** eines Systems auf. Zur Definition von **perfekte Technologie** müssen allerdings zuvor noch die Begriffe **Prozessor**, **Behälter**, **perfekter Prozessor** und **perfekter Behälter** eingeführt werden.

- **Prozessor**
  Ein **Prozessor** ist eine Einheit, die eine oder mehrere Aktivitäten ausführen kann. Der Begriff ist nicht auf elektronische Komponenten beschränkt, sondern allgemein gefaßt. Auch Menschen können Prozessoren sein.

[1] Diese Definitionen sind informale Begriffsabgrenzungen und keine mathematisch rigorosen Definitionen.

- **Behälter**
  Ein **Behälter** speichert Daten (oder Objekte).

- **perfekter Prozessor**
  Ein **perfekter Prozessor** besitzt uneingeschränkte Leistungsfähigkeit, macht keine Fehler, ist unendlich schnell und verursacht keine Kosten.

- **perfekter Behälter**
  Ein **perfekter Behälter** gewährt einfachsten Zugriff auf Daten (oder Objekte), besitzt unendliche Kapazität und verursacht keine Kosten.

- **perfekte Technologie**
  Eine **perfekte Technologie** besteht aus perfekten Prozessoren und perfekten Behältern.

Eine perfekte Technologie ist — wie man leicht einsieht — nicht weiter zu verbessern. Sie hält allen zukünftigen Entwicklungen stand. Die Annahme einer perfekten Technologie als Implementierungstechnologie ist der Ausgangspunkt zur folgenden Definition der **logischen Essenz** eines Systems. Die Suche nach den logischen Systemeigenschaften bzw. den logischen Anforderungen an ein System ist nach [MP88] gleichbedeutend mit der Suche nach der **logischen Essenz**.

- **logische Essenz eines Systems**
  Annahme: Zur Implementierung eines Systems kann auf eine perfekte Technologie zurückgegriffen werden.
  Sämtliche Aktivitäten, die ein System auch unter dieser Annahme durchführen muß, und sämtliche Informationen, die es sich trotz der perfekten Technologie merken muß, bilden die **logische Essenz** des Systems.

- **logische Systemeigenschaften**
  Eine Systemeigenschaft ist **logisch**, wenn sie unabhängig von einer bestimmten Implementierungstechnologie ist.

- **physikalische Systemeigenschaften**
  Eine Systemeigenschaft ist **physikalisch**, wenn sie aufgrund einer bestimmten Implementierungstechnologie entstanden ist.

Zur Bestimmung der logischen Anforderungen an das SUNRISE-System ist es also notwendig, aus dem Gemisch von physikalischen und logischen Eigenschaften nur die logischen zu extrahieren. Diese logische Essenz beschreibt das SUNRISE-System im Hinblick auf die bestmögliche Implementierungstechnologie oder, um es anders auszudrücken, unabhängig von einer konkreten, eingeschränkten Technologie. Dadurch erhält man eine Beschreibung, die zu jeder Neuimplementierung oder Veränderung des Systems, auch auf verbesserter Technologie, als Ausgangspunkt genutzt werden kann.

Bevor die Arbeit auf die Ergebnisse der Systemanalyse von SUNRISE eingeht, wird die angewendete Strategie zumindest in ihren Grundzügen erläutert: Als erstes

erfolgt eine Analyse und Beschreibung der nicht perfekten **Systemumwelt**. Es schließt sich eine möglichst präzise und vollständige Formulierung der **Systemziele** an. In der Praxis stellt sich hier ein ernstzunehmendes Problem. Die Systemziele liegen zu Beginn eines Projektes selten vollständig vor. Oftmals werden diese im Entwicklungsprozeß ergänzt oder verändert[2]. Nach der Feststellung der Systemziele folgt die Bestimmung der **externen** und **zeitlichen Ereignisse** des Systems:

- **externe Ereignisse**
  Ein geplantes interaktives System soll auf ganz bestimmte Ereignisse reagieren, die von Objekten oder Personen aus der Systemumwelt ausgelöst werden. Diese Ereignisse heißen **externe Ereignisse**.

- **zeitliche Ereignisse**
  Ereignisse, die durch das Erreichen eines bestimmten Zeitpunktes ausgelöst werden, heißen **zeitliche Ereignisse**.

Die externen und zeitlichen Ereignisse legen die logische Kommunikation mit der Systemumwelt fest. Nicht beschrieben wird durch diese die physikalische Realisierung einzelner Schnittstellen.

Ausgehend von den Systemzielen sowie den externen und zeitlichen Ereignissen, werden die **essentiellen Aktivitäten** des Systems und die **essentiellen Informationen** abgeleitet. Letztere bilden den **essentiellen Speicher** des Systems.

- **essentielle Aktivitäten**
  Die Aktivitäten, die ein System auch bei perfekter Technologie ausführen muß, heißen **essentielle Aktivitäten**. Zwei Arten essentieller Aktivitäten werden unterschieden:

  - **grundlegende Aktivitäten**
    Diese tragen direkt zum Erreichen der Systemziele bei.
  - **Verwaltungsaktivitäten**
    Sie dienen der Aktualisierung des essentiellen Speichers.

- **essentielle Informationen**
  Die Informationen, die sich ein System auch bei perfekter Technologie merken muß, heißen **essentielle Informationen**.

- **essentieller Speicher**
  Die essentiellen Informationen eines Systems bilden zusammen den **essentiellen Speicher**.

Mit den essentiellen Aktivitäten und dem essentiellen Speicher ist die logische Essenz eines Systems bestimmt. Ein System auf der Basis einer perfekten Implementierungstechnologie müßte sich keine weiteren Informationen merken und müßte auch keine weiteren Aktivitäten ausführen.

[2] Obwohl SUNRISE von dieser Problematik anfänglich auch betroffen war, konnte sich diese Arbeit auf ein, hinsichtlich grundlegender Systemziele, mittlerweile recht konstantes System konzentrieren.

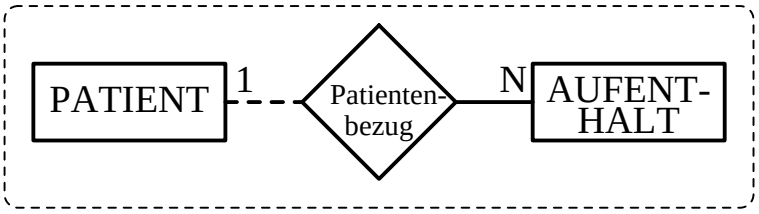

Abbildung 2.1: E/R-Diagramm *Patientenakte*

Es stellt sich das Problem, wie man die Ergebnisse der Ausarbeitung der logischen Essenz verständlich beschreibt. Zunächst bietet sich die Möglichkeit einer Beschreibung in Prosatext an. Eine solche ist für die essentiellen Aktivitäten vielleicht noch sinnvoll, für den essentiellen Speicher jedoch zu unübersichtlich. Deshalb werden in [MP88] semiformale Methoden[3] wie **Datenstrukturdiagramme, Entity-Relationship-Diagramme (E/R-Diagramme)** und **Mini-Spezifikationen** sowie **Datenflußdiagramme (DFD-Diagramme)** zur Beschreibung des essentiellen Speichers und der essentiellen Aktivitäten vorgeschlagen. In dieser Arbeit werden zur vorläufigen Beschreibung der logischen Essenz Entity-Relationship-Diagramme in Kombination mit Datenflußdiagrammen verwendet. Allerdings erfolgt eine semiformale Beschreibung hier nur als Zwischenziel zu einer formalen Spezifikation der logischen Essenz. Sie dient eher als Gedankenstütze und ist deshalb weniger ausführlich und umfangreich als in [MP88] vorgeschlagen. Die spätere formale Spezifikation wird den Detaillierungsgrad der, in [MP88] vorgeschlagenen, Beschreibungen in jedem Fall erreichen.

Durch zwei einfache Beispiele werden Entity-Relationship- und Datenflußdiagramme kurz erläutert. Weitere Erläuterungen zur Entity-Relationship-Modellierung findet man z. B. in [Che89], [CK91], [HNSE87], [BCN92], [GL89] und [Hoh93]. Ein erweitertes Entity-Relationship-Modell, mit formaler Semantik, wird in [Gog93] vorgestellt. Datenflußdiagramme werden z. B. in [MP88] näher beschrieben. Empfehlenswert ist insbesondere [Bud93]. Dort werden viele der wichtigsten semiformalen Beschreibungsmittel im Zusammenhang mit unterschiedlichen Vorgehensweisen zur Software-Entwicklung erläutert.

- **Entity-Relationship-Modellierung (E/R-Modellierung)**
  Ein Entity-Relationship-Diagramm ist ein graphisches Ausdrucksmittel zur Modellierung von Objekten (**Entities**) und zur Darstellung von Beziehungen (**Relationships**). Objektspeicher werden als Kästchen dargestellt, Beziehungen zwischen Objekten durch Rauten mit Kanten. Abbildung 2.1 zeigt ein Beispiel: Es gibt zwei **Entitytypen**: Patient und Aufenthalt (Hospitalaufenthalt). Hier ist nicht dargestellt, daß diese Entitytypen weiter strukturiert

[3] Der Begriff *semiformale Methoden* wird an dieser Stelle bewußt auch für Methoden mit formalen Charakter wie zum Beispiel die Entity-Relationship-Modellierung verwendet. Dies geschieht zur Abgrenzung einer, für diese Arbeit, vorläufigen Beschreibung der logischen Essenz von einer formalen Spezifikation.

sind. Der Entitytyp Patient soll z. B. folgende **Attribute** besitzen: Patientenidentifikationsnummer (PatId), Patientenname, Geburtstag, Geschlecht, usw. Aufenthalt soll als Attribute aufweisen: Aufenthaltsidentifikationsnummer (AufId), Einlieferungsdatum, Entlassungsdatum, Station, Zimmernummer, behandelnder Arzt, Diagnose, usw. Die beiden Entitytypen sind verbunden durch einen **Relationshiptyp** Patientenbezug (jeder Aufenthalt steht in Bezug zu einem Patienten). Zu diesem **Relationshiptyp** werden im Diagramm weitere einschränkende Aussagen getroffen: Es handelt sich hier um eine Relationship mit dem **Grad 1:N**. Einem einzelnen Vertreter des Entitytyps Patient (einer **Entity** Patient) sind möglicherweise N Vertreter (Entities) des Typs Aufenthalt zugeordnet. Umgekehrt wird jede Entity des Typs Aufenthalt auf maximal eine Entity des Typs Patient abgebildet. Im allgemeinen wird zwischen den Graden **1:1, 1:N (N:1)** und **N:M** unterschieden. Eine weitere Aussage zu dem Relationshiptyp Patientenbezug wird durch die gestrichelte Kante formuliert. Sie drückt eine zwingende Teilnahme (**zwingende Partizipation**) jeder Entity des Typs Aufenthalt am Relationshiptyp Patientenbezug aus. Es darf somit in einer konkreten **Datenbank** des modellierten Datenbanktyps Patientenakte keine Entity des Typs Aufenthalt existieren, der nicht einer Entity des Typs Patient zugeordnet ist.

Die Aussagen zu den Graden von Relationshiptypen und zur zwingenden Partizipation gehören zu den **Integritätsbedingungen** zum modellierten Datenbanktyp. Im allgemeinen formuliert eine Integritätsbedingung eine Einschränkung an den Datenbanktyp, die statisch für jeden Vertreter abprüfbar sein muß. Solche Integritätsbedingungen die bereits im E/R-Diagramm dargestellt werden können, weil dazu graphische Ausdrucksmittel vorgesehen sind (z. B. Grade von Relationshiptypen), heißen im folgenden **schematische Integritätsbedingungen**. Solche die nicht dargestellt werden können, weil die Ausdrucksmittel dazu fehlen, heißen **spezifische Integritätsbedingungen**. Hier nicht erläutert wurden die Ausdrucksmittel zur Formulierung der **Schlüsseleigenschaft**[4] von Attributeinträgen und zur Forderung nach **nichtleeren Attributeinträgen**[5].

Anmerkung: In dieser Arbeit erfolgt eine Beschränkung auf **binäre Relationships**. Nichtbinäre Relationships können, wie z. B. in [CK91] beschrieben, in binäre transformiert werden.

- **Datenflußdiagramme (DFD-Diagramme)**
  Datenflußdiagramme dienen der abstrakten Beschreibung essentieller Systemaktivitäten. Folgende Aspekte werden durch ein DFD-Diagramm berücksichtigt:

[4] Diejenigen Attribute eines Entitytyps, die durch einen Zusatz *primary key* ausgezeichnet wurden, sollen gemeinsam einen Schlüssel zu diesem Entitytyp bilden, d. h. Vertreter dieses Typs eindeutig identifizieren.

[5] Trägt eine Attribut den Zusatz *mandatory*, so wird dadurch gefordert, das unter diesem Attribut niemals der leere Attributwert eingetragen sein darf.

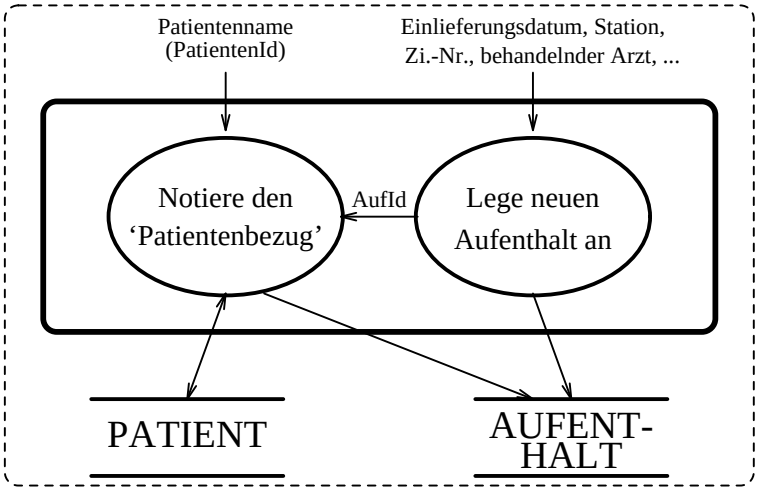

Abbildung 2.2: DFD-Diagramm *Patientenaufnahme*

.

- Wie kann eine essentielle Aktivität in Unteraktivitäten zerlegt[6] werden und wie interagieren diese?
- Welche Daten erhält eine essentielle Aktivität (und ihre Unteraktivitäten) aus der Systemumwelt?
- Was sind die Auswirkungen der Aktivität (und der Unteraktivitäten) auf den essentiellen Speicher? Auf welche Komponenten des essentiellen Speichers wird lesend zugegriffen und auf welche schreibend?

Abbildung 2.2 zeigt ein Beispiel eines DFD-Diagramms, das auf dem E/R-Diagramm aus Abbildung 2.1 aufbaut. Es beschreibt eine essentielle Aktivität **Patientenaufnahme**. Damit das Beispiel nicht zu kompliziert wird, beschränkt es sich auf den Fall der Wiederaufnahme eines Patienten, d. h. zu einem aufzunehmenden Patienten existieren bereits Patientendaten im essentiellen Speicher. Zwischen zwei parallelen Linien werden die **Datenspeicher** angezeigt (Patient und Aufenthalt). Nicht eingezeichnet werden Beziehungstypen. Die Aktivität **Patientenaufnahme** wird durch den dick umrandeten Bereich repräsentiert. Sie zerfällt in die zwei Unteraktivitäten **Lege neuen Aufenthalt an** und **Notiere den Patientenbezug**. Weiterhin werden die aus der Systemumwelt eingehenden Daten beschrieben. Dies sind zum einen Daten, die zur Initialisierung einer neuen Entity des Typs Aufenthalt benötigt werden (Einlieferungsdatum, Station, Zi.-Nr., behandelnder Arzt, ...), zum anderen Daten zur Identifikation des aufgenommenen Patienten im essentiellen

[6] Die Zerlegung einer essentiellen Aktivität in Unteraktivitäten steht in gewisser Weise in Widerspruch zum Begriff der logischen Essenz. Diese Zerlegung könnte auf andere Art erfolgen, ist damit im strengen Sinne also ein Implementierungsdetail. Doch zu Beschreibungszwecken ist eine solche, der Aufnahmefähigkeit des Menschen angepaßte Zerlegung ratsam, wenn nicht sogar notwendig.

Speicher (Patientenname oder besser Patienten-Identifikationsnummer). Die Pfeile beschreiben die Richtung des Datenflusses. Die Unteraktivität **Lege neuen Aufenthalt an** legt in dem Datenspeicher Aufenthalt eine neue Entity des Typs Aufenthalt an — dabei berechnet sie eine neue Aufenthaltsidentifikationsnummer (AufId) — und trägt die Initialisierungsdaten aus der Systemumwelt ein (schreibender Zugriff auf den Datenspeicher). An die Unteraktivität **Notiere den Patientenbezug** leitet sie dann die berechnete AufId weiter. Diese zweite Unteraktivität identifiziert zunächst die Patienten-Entity des Patienten, der, durch die aus der Systemumwelt eingehenden Daten, eindeutig beschrieben wird (lesender Zugriff auf den Datenspeicher), und registriert dann die Beziehung **Patientenbezug** zwischen der Aufenthalts-Entity, die durch die AufId identifiziert wird, und der Patienten-Entity. Dazu erfolgt ein schreibender Zugriff auf den Speicher des Relationshiptyps Patientenbezug. Weil dieser im Diagramm jedoch nicht berücksichtigt ist, wird je ein Pfeil zu den Datenspeichern der betroffenen Entitytypen eingezeichnet. Damit liegt nun eine abstrakte Beschreibung der essentiellen Aktivität **Patientenaufnahme** vor, die als Grundlage für eine weitere, präzisere Beschreibung — ob informal, semiformal oder formal — verwendet werden kann.

Hier nicht vorgestellt, aber dennoch wichtig für die folgende Arbeit, ist ein unbenannter Pfeil zwischen zwei Unteraktivitäten. Dadurch wird eine zeitliche Abhängigkeit zwischen beiden Aktivitäten ausgedrückt. Die Aktivität, auf die der Pfeil gerichtet ist, darf erst dann mit ihrer Arbeit beginnen, wenn die andere beendet ist.

Anhand der Abbildung 2.3 wird die Strategie der Systemanalyse zusammengefaßt. Eine formale Spezifikation ist allerdings in [MP88] nicht vorgesehen. Deshalb trennt eine gestrichelte, horizontale Linie die Vorgehensweise nach [MP88] (oberhalb dieser Linie), von der Erweiterung um eine formale Ebene, wie sie diese Arbeit demonstriert.

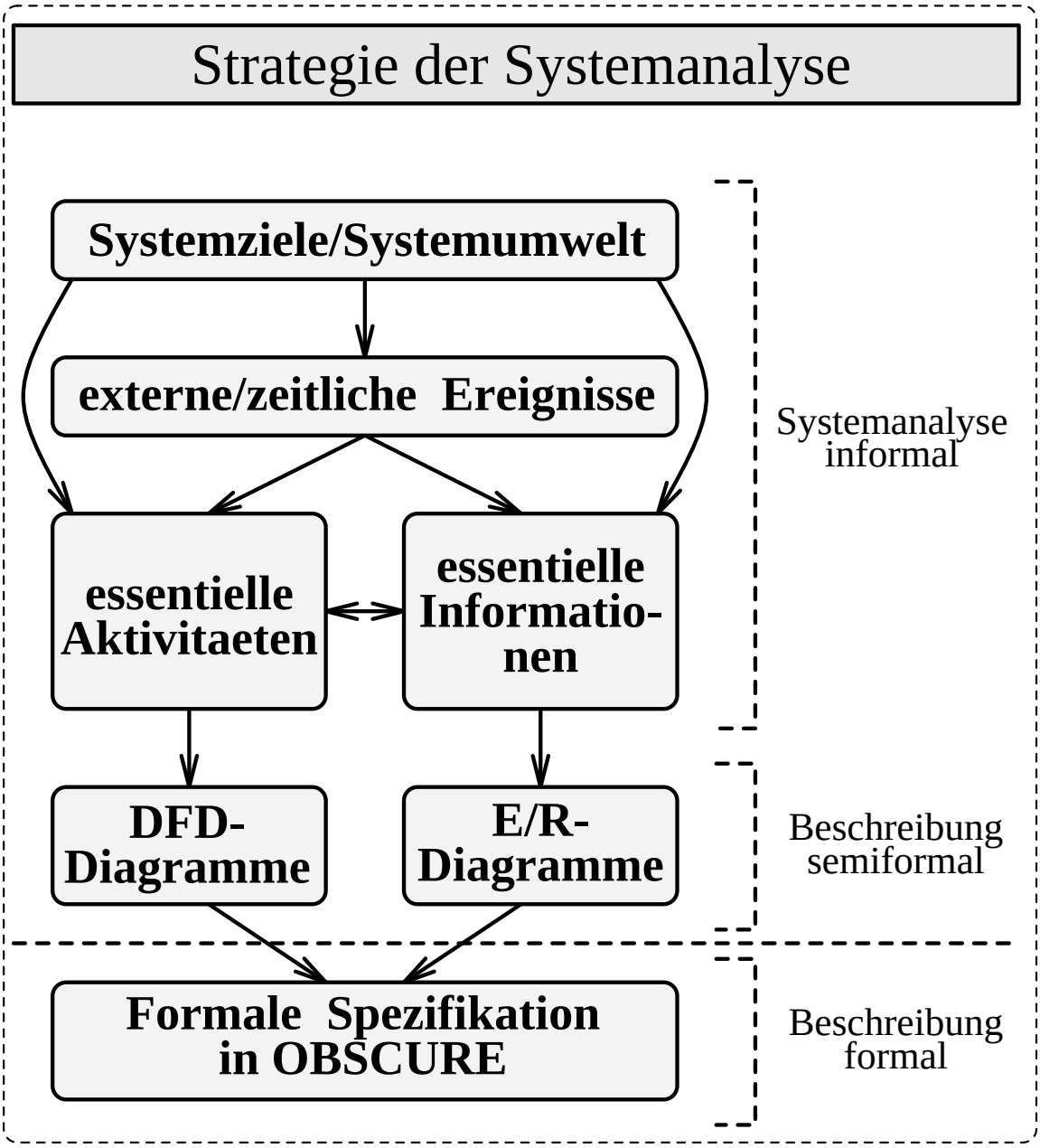

Abbildung 2.3: Strategie zur Systemanalyse

2.2 Analyse des SUNRISE-Systems

In diesem Abschnitt werden die Ergebnisse der Systemanalyse vorgestellt, die gemäß der erläuterten Strategie durchgeführt wurde. Nacheinander werden Systemumwelt, Systemziele, externe und zeitliche Ereignisse, sowie essentielle Aktivitäten und essentielle Informationen zum SUNRISE-System diskutiert. Die Reihenfolge entspricht dabei nicht unbedingt der Reihenfolge der Ausarbeitung. Zum Beispiel wird die Systemumwelt zuerst vorgestellt, weil dies ein leichteres Nachvollziehen der Systemziele ermöglicht, während die Ausarbeitung von Systemumwelt und Systemzielen eher in umgekehrter Reihenfolge erfolgte.

2.2.1 SUNRISE-Systemumwelt

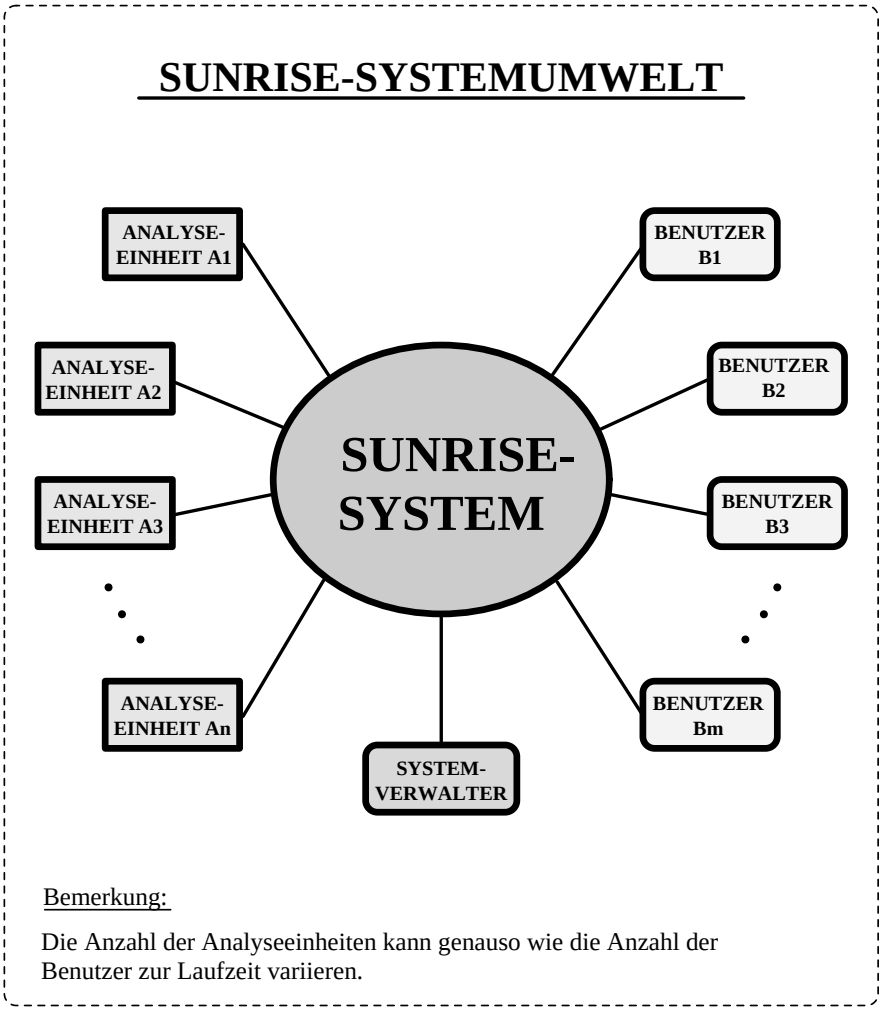

Abbildung 2.4: Systemumwelt von SUNRISE

Die Abbildung 2.4 erläutert die Systemumwelt graphisch. Diese besteht einerseits aus verschiedenen medizinischen Analysegeräten. Beispiele hierfür sind Röntgengeräte, Ultraschallstationen, Kernspintomographen usw. Untersuchungsdaten, die an diesen Geräten ermittelt werden, können vom SUNRISE-System zur Verwaltung und Weiterverarbeitung importiert werden. Zum anderen ist das SUNRISE-System ein Mehrbenutzer-System. Als Anwender sind primär Ärzte, Pflege- und Dienstpersonal einer Klinik vorgesehen. Sie können Verwaltungsaufgaben wahrnehmen, Untersuchungsdaten weiterbearbeiten mittels zur Verfügung gestellter Bearbeitungsoperationen (z. B. Graustufenfilter) und sogar neue Bearbeitungsoperationen selbst definieren und dem System hinzufügen. Ein Systemverwalter übernimmt die administrativen Aufgaben.

2.2.2 SUNRISE-Systemziele

Das SUNRISE-System verfolgt folgende Grundziele: Zum einen soll es die Verwaltung und Weiterbearbeitung unterschiedlicher Untersuchungsdaten (z. B. Röntgendaten oder Spektren) in einer Klinik ermöglichen, zum anderen aber auch die Entwicklung neuer Bearbeitungsoperationen und deren Integration zur Laufzeit. Die folgende Auflistung einzelner Systemziele ist gemäß dieser Grundideen aufgeteilt. Jedoch wird nur sehr oberflächlich auf die Systemziele eingegangen. Eine ausführlichere Beschreibung kann den Systemdokumenten [Sta93] und [BSSG93] entnommen werden. Allerdings sind dort teilweise Entwurfs- oder Implementierungsdetails mitberücksichtigt, von denen eine Analyse der logischen Essenz des Systems abstrahiert und die hier deshalb nicht berücksichtigt werden.

1. **Sunrise als System zur Verwaltung und Weiterbearbeitung von Untersuchungsdaten**

   (a) **Mehrbenutzersystem**
   Das System soll mehreren Benutzern gleichzeitig zur Verfügung stehen.

   (b) **Import und Export von Untersuchungsdaten**
   Daten sollen von unterschiedlichen Analyseeinheiten importiert werden können. Diese Daten unterscheiden sich einerseits bezüglich ihrer Art (Röntgendaten, Spektren usw.), andererseits bezüglich ihres Datenformats. Auch der Export von Daten — z. B. zu einem Drucker — soll ermöglicht werden. Beim Import werden die eingehenden Daten zu einem internen Untersuchungsdatenobjekt[7] zusammengefaßt.

   (c) **Visualisierung und Weiterbearbeitung**
   Eine wichtige Aufgabe des Systems ist die Visualisierung von Untersu-

[7] Im folgenden wird häufig der Begriff Datenobjekt oder Sunrise-Datenobjekt (S.-Datenobjekt) verwendet. In Kapitel 3 wird später der Entitytyp Sunrise-Datenobjekt genau definiert. Zunächst soll es genügen, sich unter einem Datenobjekt eine Ansammlung von Informationen unterschiedlicher Art vorzustellen, die in ihrer Gesamtheit das Ergebnis einer Analyseuntersuchung (z. B. an einem Röntgengerät) beinhalten und in das SUNRISE-System importiert wurden. Möglicherweise wurde ein Datenobjekt von SUNRISE bereits weiterverarbeitet, d. h. einzelne Informationen wurden verändert (z. B. durch einen Graustufenfilter).

chungsdaten im Sinne einer *digitalen Lichtbox* und die Möglichkeit zur Weiterbearbeitung von Untersuchungsdaten durch eingebaute Datenbearbeitungsoperationen (Beispiele hierfür sind Graustufenfilter).

(d) **komfortable Datenverwaltung**
In einem globalen Datenarchiv sollen die Untersuchungsdaten nach ihren Umweltbezügen (Bezug der Untersuchungsdaten zu Patienten, Aufenthalten, Ärzten, Kliniken) verwaltet werden. Dazu müssen Verwaltungsoperationen zur Verfügung gestellt werden. Die Datenbank soll möglichst klein gehalten werden. Ergebnisse von Bearbeitungen einzelner Untersuchungsdaten sollen nicht direkt in das globale Archiv übernommen werden, sondern nur auf expliziten Wunsch. Dazu ist jedem Benutzer ein lokales Archiv zuzuordnen, in dem die Zwischenresultate einzelner Bearbeitungschritte zunächst aufbewahrt werden.

(e) **Schutz der Benutzer vor gegenseitiger Datenmanipulation**
Das System muß eine unbeabsichtigte oder beabsichtigte Manipulation von Untersuchungsdaten, die dem Verantwortungsbereich eines bestimmten Benutzers (z. B. einem Arzt) zugeordnet sind, durch andere Benutzer verhindern.

(f) **benutzerspezifische Umgebungsdefinitionen**
Das System soll benutzerspezifische Konfigurationen ermöglichen. Gefordert sind spezifische Voreinstellungen der Parameter von Bearbeitungsoperationen, sowie der Sichtbarkeit von Bearbeitungsoperationen (an der Systemoberfläche).

(g) **Bearbeitungssequenzen**
Zusätzlich zu der benutzerspezifischen Konfiguration soll das System Benutzern die Möglichkeit bieten, Sequenzen von Bearbeitungsoperationen als persönliche Makrooperationen zu definieren.

(h) **Geschichtsinformation**
Zu jedem Untersuchungsdatenobjekt (Datenobjekt) soll sich das System eine präzise Geschichtsinformation merken, die eine Rekonstruktion dieses Datenobjekts und aller vorausgegangenen Zwischendatenobjekte aus einem importierten Datenobjekt ermöglicht. Zwischenresultate müssen somit nicht abgespeichert werden, sondern jeweils nur das letzte Datenobjekt einer Bearbeitungssequenz. Auch die Binärdateninformationen nicht importierter, sondern weiterbearbeiteter Datenobjekte müssen somit nicht explizit in die Datenbank aufgenommen werden, sondern lassen sich aus den jeweiligen importierten Vorgängerobjekten und der Information zu den angewendeten Bearbeitungsoperationen berechnen.

(i) **UNDO/REDO-Mechanismus**
Dem Anwender von Bearbeitungsoperationen soll ein UNDO/REDO-Mechanismus zur Verfügung gestellt werden, mit dem er innerhalb seines lokalen Archivs Bearbeitungsschritte zurücknehmen oder auch wiederholen kann. Dies soll ein schnelles Hin- und Herblättern in den Zwischen-

resultaten einer Bearbeitungssitzung ermöglichen. Das jeweils aktuelle Datenobjekt eines Benutzers soll auf dem Bildschirm visualisiert werden.

2. SUNRISE **als Entwicklungsumgebung für Bearbeitungsoperationen**

   (a) **Definition neuer Bearbeitungsoperationen**
   Das SUNRISE-System soll den Benutzern die Möglichkeit zur Definition neuer Bearbeitungsoperationen und Datenklassen bieten. Diese sollen zur Laufzeit in das System integriert werden können.

   (b) **Transparenz der Datenarten**
   Die intern vom System unterschiedenen Datenarten (im folgenden auch Datenklassen genannt), wie Spektren, Bilder usw., müssen für den Entwickler neuer Bearbeitungsoperationen transparent und verständlich sein.

   (c) **Bibliothek von Systemschnittstellenoperationen**
   Zur Definition neuer Bearbeitungsoperationen müssen dem Benutzer notwendige Systemschnittstellenoperationen zur Verfügung gestellt werden.

   (d) **Definitionen neuer Import-,Export- und Visualisierungsoperationen, sowie neuer Datenklassen**
   Analog zur Definition neuer Bearbeitungsoperationen oder Datenklassen, soll dem Benutzer die Möglichkeit zur Definition von Import-, Export- und Visualisierungsoperationen geboten werden. Auch diese sollen zur Laufzeit in das System integriert werden können.

2.2.3 Externe Ereignisse von SUNRISE

Wie in der Strategie vorgeschlagen, folgt der Analyse der Systemumwelt und der Systemziele die Ausarbeitung der externen und zeitlichen Ereignisse. Zeitliche Ereignisse konnten für SUNRISE nicht anlysiert werden. Beispiel für ein zeitliches Ereignis wäre eine Wochenabrechnung, wie sie in Buchungssystemen vorzufinden ist. Die Systemaktivität **Erstelle Wochenabrechnung** würde z. B. durch das Erreichen des Zeitpunktes Freitag 14:00 Uhr ausgelöst.

Die analysierten externen Ereignisse werden in der folgenden Auflistung nach ihren Auslösern unterschieden. Auslöser von Systemaktivitäten können ein *normaler* Benutzer (unter *normaler* Benutzer wird ein Benutzer ohne Entwicklungsabsichten verstanden), ein Entwickler von Bearbeitungsoperationen und der Systemverwalter sein.

1. **Externe Ereignisse, die von einem normalen Benutzer ausglöst werden:**

   (a) Import-/Export von Datenobjekten

   (b) Auswahl eines Datenobjekts nach dessen Umweltbezügen

   (c) Visualisierung eines Datenobjekts

   (d) Bearbeitung eines Datenobjekts mit einer Bearbeitungsoperation

(e) UNDO/REDO eines Bearbeitungsschritts
(f) Abspeichern oder Löschen eines Datenobjekts
(g) Veränderung der benutzerspezifischen Voreinstellungen
(h) Definition einer Bearbeitungsequenz (Makrodefinition)
(i) Löschen oder Hinzufügen eines Umweltbezugs

2. **Externe Ereignisse, die von einem Entwickler neuer Bearbeitungsoperationen ausgelöst werden:**
   (a) Definition und Integration einer Bearbeitungsoperation
   (b) Definition und Integration einer Import- oder Exportoperation
   (c) Definition und Integration einer Visualisierungsoperation
   (d) Definition und Integration einer Datenklasse
3. **Externe Ereignisse, die vom Systemverwalter ausgelöst werden:**
   (a) Anlegen oder Entfernen eines Benutzers
   (b) Löschen eines Datenobjekts (auch solche, die dem Verantwortungsbereich eines anderen Benutzers unterstehen)
   (c) Einrichten eines initialen Systemzustands

2.2.4 Essentielle Aktivitäten von SUNRISE

Die essentiellen Aktivitäten lassen sich bestimmen, indem man untersucht, welche Aktivitäten das System bereitstellen muß, um den Forderungen der Systemziele und damit denen der externen, sowie zeitlichen Ereignisse zu genügen. Prinzipiell gibt es zu jedem externen oder zeitlichen Ereignis eine essentielle Aktivität, die auf das Ereignis reagiert. Es ist aber auch denkbar, daß ein externes Ereignis in mehrere essentielle Aktivitäten zerfällt. Übersehen werden oft solche essentielle Aktivitäten, die lediglich der Verwaltung der essentiellen Informationen dienen.

Weil in diesem Fall die Liste der essentiellen Aktivitäten sehr ähnlich ist zur Auflistung der externen Ereignisse, wird hier auf diese verzichtet. Eine detaillierte Beschreibung der, für die weitere Untersuchung ausgewählten, essentiellen Aktivität WSAV **(speichere lokales Datenobjekt global ab)**, dem zentralem Untersuchungsgegenstand des **dualen Speicherkonzepts**, erfolgt im Abschnitt 2.3.

2.2.5 Essentielle Informationen von SUNRISE

In engem Zusammenhang zu der Analyse der essentiellen Aktivitäten erfolgt die Ausarbeitung des essentiellen Speichers. Unter der Annahme, daß zur Implementierung der essentiellen Aktivitäten und zur Verwaltung der zu analysierenden essentiellen Informationen eine perfekte Technologie eingesetzt werden kann, ist zu klären, welche Informationen sich das System dennoch merken muß. Zur Darstellung der essentiellen Informationen ist es sinnvoll, diese zu klassifizieren — möglichst nach Vorgaben aus der realen Umwelt. Ein Beispiel für SUNRISE ist: Die klassifizierte Einheit

Sunrise-Datenobjekt (Untersuchungsdatenobjekt) umfaßt die Einzelinformationen (Attribute) **DatenId, Binärdaten, Datenparameter, History** (Geschichtsinformation), **ROI** [8] und **CreationInfo**. Diese Einzelinformationen sind möglicherweise selbst weiter strukturiert. Die klassifizierte Einheit **Hospital** enthält die Einzelinformationen **HospitalId, Hospitalname, Adresse** und **CreationInfo**. Neben den klassifizierten Einheiten existieren Beziehungsinformationen (Relationshiptypen). Diese stellen die Beziehungen zwischen den klassifizierten Einheiten her. Ein Beispiel dazu ist: SUNRISE muß sich die Zuordnung von Untersuchungsdaten zu Patienten bzw. Aufenthalten merken.

Es ist wichtig hervorzuheben, daß die Klassifikation der Einzelinformationen teilweise im Widerspruch steht zum Begriff der logischen Essenz, weil es nicht das Ziel sein soll, für eine Implementierung des Systems vorzuschreiben, wie die Einzelinformationen zusammengefaßt werden sollen. Ziel ist es eigentlich, eine Beschreibung eines Systems zu erstellen, ohne solche Detailentscheidungen zu treffen. Doch ist eine Beschreibung eines essentiellen Speichers ohne eine, dem Menschen leicht verständliche, Klassifizierung der Einzelinformationen, schwierig und unübersichtlich. Auf jeden Fall gilt dies für Systeme mit großem essentiellen Speicher, also mit vielen Beziehungen zwischen den Einzelinformationen, die womöglich sehr unterschiedlicher Natur sind. Es bleibt festzuhalten, daß zur verständlichen Beschreibung des essentiellen Speichers von SUNRISE eine solche Klassifizierung der Einzelinformationen notwendig ist. Diese soll aber nicht notwendigerweise verbindlich sein im Hinblick auf die Implementierung[9].

Anmerkung: Tatsächlich weicht die aktuelle Implementierung zu SUNRISE von der, hier zu Beschreibungszwecken vorgenommenen Klassifizierung ab. Sie kennt keine klassifizierte Einheit **Datenklasse**. Die Datenklasseninformation wird analog zu einem Attribut eines Untersuchungsdatenobjekts (Sunrise-Datenobjekt) implementiert. In dieser Ausarbeitung der logischen Essenz von SUNRISE werden Datenklassen als eigenständige Einheiten angesehen, weil so die wichtige Bedeutung der Datenklassen für das SUNRISE-System deutlicher zu erkennen ist. Denn sämtliche Bearbeitungsoperationen stellen über den **Datenklassenbezug** eines Datenobjektes fest, ob sie auf diesem korrekt arbeiten können, also anwendbar sind oder nicht.

Im folgenden wird nun der essentielle Speicher des SUNRISE-Systems durch ein E/R-Diagramm (siehe Abbildung 2.5) beschrieben. Die Entitytypen dieses Diagramms beschreiben die klassifizierten Einheiten von Einzelinformationen. Die Einzelinformationen selbst finden sich in den Attributen der Entitytypen wieder und werden in diesem Diagramm, aus übersichtsgründen, nicht dargestellt. Für den Ausschnitt **duales Speicherkonzept** werden diese ab Seite 38 näher beschrieben, während auf die des restlichen Systems nicht weiter eingegangen wird. Die Art der Beziehungen zwischen den Einheiten von Einzelinformationen (Entities) werden durch die Relationshiptypen des Diagramms beschrieben. Ebenfalls nicht darge-

[8] Eine Region Of Interest enthält einen Hinweis auf einen Auschnitt des Binärdatenteils, der momentan von besonderem Interesse ist.

[9] Zum Zwecke der Verifikation könnte ein Beachten dieser Klassifizierung in den weiteren Implementierungschritten allerdings sehr sinnvoll sein.

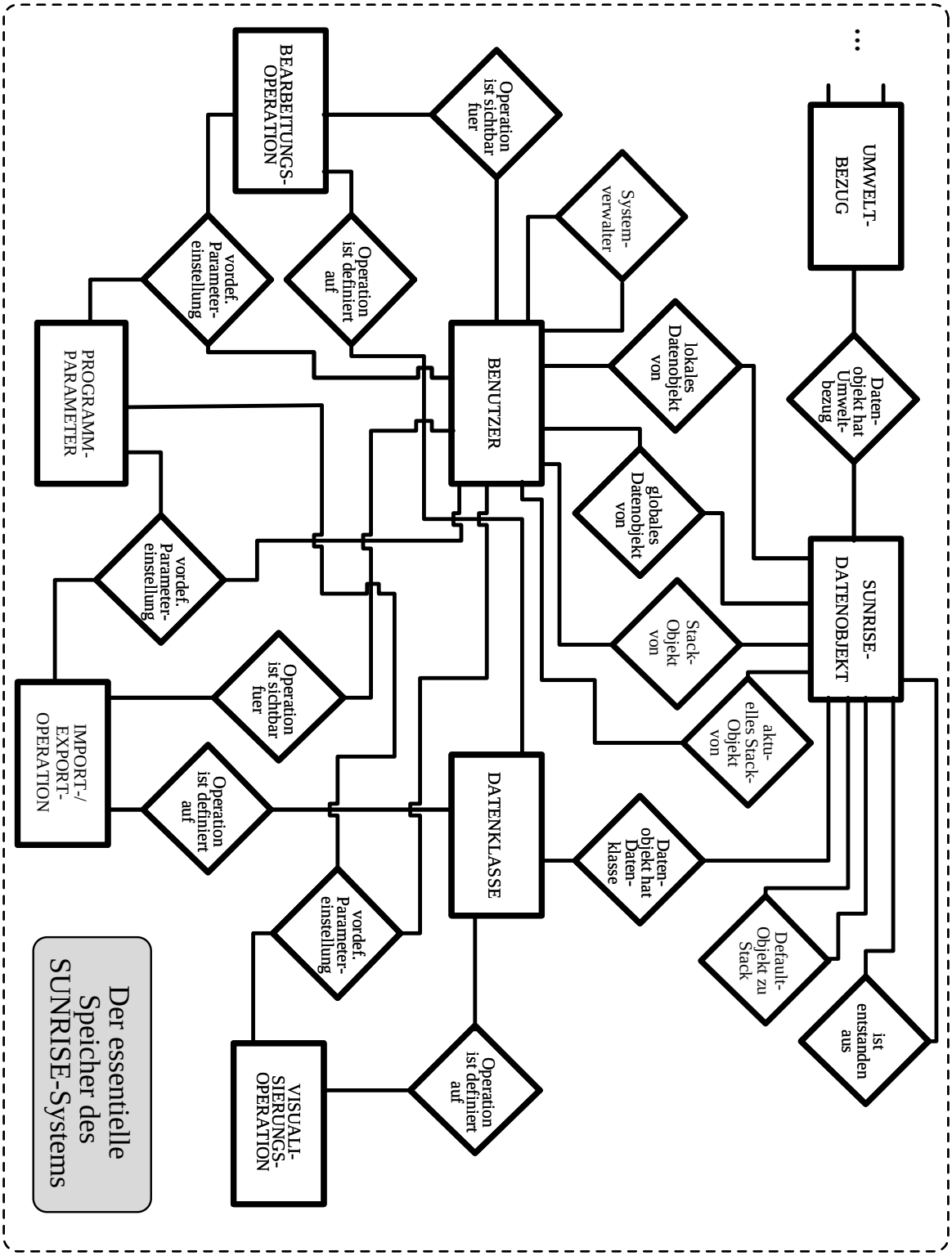

Abbildung 2.5: E/R-Diagramm zum essentiellen Speicher

stellt in diesem Diagramm werden Integritätsbedingungen. Für den ausgewählten Ausschnitt **duales Speicherkonzept** werden diese auf Seite 36 erläutert. Aus Platzgründen werden auch die Entitytypen **Aufenthalt, Patient, Hospital** und **Root** nicht einzeln eingezeichnet, sondern durch den Eintrag **Umweltbezug** ersetzt. Im Diagramm zum ausgewählten Ausschnitt (siehe Abbildung 2.8), sind diese detailliert eingezeichnet.

Auf die Gesamtanalyse zum SUNRISE-System und deren Ergebnisse wird nun nicht weiter eingegangen. Weitere Anmerkungen finden sich auch im Abschnitt 4.2.2. Der nächste Abschnitt konzentriert sich ausschließlich auf den Systemausschnitt **duales Speicherkonzept**, der im folgenden Kapitel formal spezifiziert werden soll.

2.2.6 Problemhinweis

Die Ausarbeitung des essentiellen Speichers zu SUNRISE enthält, neben den bereits angesprochenen Entscheidungen zur Klassifizierung von essentiellen Informationen, weitere Details, die nicht unbedingt aus den informalen Anforderungsbeschreibungen zu SUNRISE abgeleitet werden können. Dies liegt aber weniger daran, daß diese Arbeit versucht, Implementierungsdetails als Systemanforderungen zu präsentieren, sondern daran, daß die informalen Anforderungen an das SUNRISE System (im Sinne eines Lastenhefts) nicht genügend ausführlich und präzise sind. Dies ist ein in der Praxis häufig auftretendes Problem und damit nicht spezifisch für diesen Anwendungsfall. Ein Beispiel soll die erläuterte Problematik belegen: Aus der informalen Anforderungsbeschreibung zum SUNRISE-System geht nicht hervor, daß sich SUNRISE Informationen zum Familienstand von Patienten merken soll. In der aktuellen Implementierung sind solche Informationen zu finden, können aber nicht als ein Implementierungsdetail angesehen werden. Weil diese sicherlich nicht aus Willkür von einem Programmentwickler hinzugefügt wurden, sind solche Information zu den essentiellen Informationen des Systems zu zählen. Trotzdem existiert kein entsprechender Hinweis in den informalen Systemanforderungen, der diese Familienstand-Information als eine essentielle Information ausweist. Dies ist womöglich deshalb der Fall, weil die Familienstand-Information als ein sehr nebensächlicher Aspekt bei der informalen Anforderungsdefinition zum System bewußt übergangen wurde. Verständlicherweise konzentrierte man sich hauptsächlich auf die wesentlichen und problematischen Aspekte des zu erstellenden Systems und abstrahierte von solchen Aspekten, von denen man annehmen konnte, daß sie auch nachträglich noch leicht beachtet und verändert werden können.

Die Ausarbeitungen dieser Arbeit zum Systemausschnitt *duales Speicherkonzept* können deshalb nicht allein aus der Sicht der verfügbaren informalen Anforderungsbeschreibungen auf ihre Richtigkeit überprüft werden, sondern müssen auch im Hinblick auf die aktuelle Implementierung diskutiert werden. Diese liefert, wie erläutert, Hinweise zu essentiellen Systemanforderungen, die zwar de facto existieren, aber nicht niedergeschrieben wurden. Davon betroffen ist auch das *duale Speicherkonzept*.

2.3 Der Ausschnitt *duales Speicherkonzept*

In diesem Abschnitt wird das Ergebnis der Systemanalyse zum Ausschnitt **duales Speicherkonzept** beschrieben. Dazu wird zunächst näher erläutert, was unter dem Ausschnitt **duales Speicherkonzept** zu verstehen ist. Das zentrale Interesse gilt später der essentiellen Aktivität WSAV. Auch diese wird zunächst informal erklärt.

- **duales Speicherkonzept**
  Das duale Speicherkonzept unterscheidet zwischen **lokalen** und **globalen** Datenobjekten. Der globale Datenobjektspeicher[10]— besser: das globale Archiv — von SUNRISE umfaßt die globalen Datenobjekte aller Benutzer. Zur Weiterbearbeitung wählt ein Benutzer ein Untersuchungsdatenobjekt aus dem globalen Datenobjektspeicher aus. Dieses Datenobjekt definiert einen neuen Bearbeitungsstack[11] im lokalen Speicherbereich (lokalen Archiv) des Benutzers, d. h. es wird dadurch zu einem sogenannten **Stack-Objekt** dieses Benutzers. Weil jeder Benutzer gleichzeitig mehrere Arbeitsstacks öffnen kann, wird einer dieser Stacks als der aktuelle ausgezeichnet. Dies geschieht, indem eines der Stack-Objekte den zusätzlichen Status **aktuelles Stack-Objekt** erhält. Jeder Arbeitsstack enthält ein weiteres ausgezeichnetes Datenobjekt: Das **Default-Objekt** dieses Stacks für weitere Bearbeitungsschritte. (Das Default-Objekt des aktuellen Arbeitsstacks ist in der aktuellen Implementierung auch stets dasjenige, das auf dem Bildschirm visualisiert wird.) Jede Anwendung einer Bearbeitungsoperation oder auch von WSAV bezieht sich auf das Default-Objekt des aktuellen Arbeitsstacks des ausführenden Benutzers. Das resultierende Datenobjekt einer Operationsanwendung wird zunächst ein **lokales Datenobjekt** des Benutzers und auf dessen aktuellen Arbeitsstack abgelegt. Es wird dann das neue **Default-Objekt** für weitere Bearbeitungsschritte. Das Resultat eines erneuten Bearbeitungsschritts (auf dem Default-Objekt) wird wieder ein **lokales Datenobjekt** und **Default-Objekt** des aktuellen Arbeitsstacks. Auf diese Weise kann jeder Benutzer seine Arbeitsstacks anreichern. Mit dem UNDO-Befehl kann der Benutzer den Status des Default-Objekts an das Vorgängerobjekt des derzeitigen Default-Objekts des aktuellen Stacks übertragen - dies vermittelt den Eindruck, als wäre der letzte Bearbeitungsschritt rückgängig gemacht worden (insbesondere weil nun das neue Default-Objekt, also das Vorgängerobjekt des bisherigen Default-Objekts, auf dem Bildschirm angezeigt wird). Eine Anwendung des REDO-Befehls macht einen UNDO-Schritt wieder rückgängig, d. h. der Status des Default-Objekts

[10] Die Begriffe **lokaler** und **globaler Datenobjektspeicher** dürfen (beschränkt auf die **logische Essenz**) nicht mit physikalischen Speicherbereichen verwechselt werden. An dieser Stelle sind die Begriffe charakterisiert durch die logische Eigenschaft eines Datenobjekts, in lokaler oder globaler Beziehung zu einem Benutzer zu stehen.

[11] Die Begriffsbildung in dieser Arbeit ist teilweise angelehnt an die Systemdokumente zu SUNRISE. Hier wird von einem Bearbeitungsstack gesprochen, weil die Art, wie die Resultate von Weiterbearbeitungen dem lokalen Arbeitsbereich angefügt werden sollen, an das *pushen* bei einem Stack erinnert. Das oberste, also zuletzt eingefügte Objekt, wird zum aktuellen Objekt (**Default-Objekt** für Anwendungen).

wird in der Bearbeitungshierarchie wieder nach oben verschoben. Wenn durch Anwendung des UNDO-Befehls ein Datenobjekt zum Default-Objekt des aktuellen Stacks ausgezeichnet wurde, das nicht Top-Element dieses Stacks ist, dieses Datenobjekt also schon weiterbearbeitet wurde und die Resultate oberhalb des Default-Objekts auf dem Arbeitsstack abgelegt wurden, so werden bei einer erneuten Anwendung einer Bearbeitungsoperation (diese bezieht sich immer auf das Default-Objekt des aktuellen Stacks) die vorherigen Resultate aus dem Stack entfernt, und das neue Resultat wird oben auf dem Stack abgelegt.

Wichtig an diesem Konzept ist, daß die Resultate eines Bearbeitungsschritts nicht sofort als globale Datenobjekte aufgenommen werden und damit anderen Benutzern auch zunächst nicht zugänglich sind. Erst am Ende einer SUNRISE-Sitzung soll der Benutzer die wichtigen Resultate der Bearbeitungsschritte global abspeichern. Dies erfolgt durch Anwendung der Operation WSAV. Dabei sollte er darauf achten, möglichst nur die Endresultate (obersten Elemente der Bearbeitungsstacks) oder wichtige Zwischenresultate in den globalen Speicherbereich aufzunehmen, weil die Geschichtsinformation eines Endresultats die Rekonstruktion sämtlicher Zwischenergebnisse gewährleistet. Auf diese Weise kann das globale Archiv möglichst klein und übersichtlich gehalten werden. Zudem sind die Binärdaten der Datenobjekte für nicht-importierte Datenobjekte optional — die Geschichtsinformation, die bei Operationsanwendung im Resultatobjekt vermerkt wird, ermöglicht deren Berechnung aus den importierten Datenobjekten.

Hinweis: In dieser **informalen Sicht** auf das **duale Speicherkonzept** nehmen die **Stack-Objekte** eine besondere Rolle ein: Obwohl ein **Stack-Objekt** einen lokalen Arbeitsstack eines Benutzers definiert, ist es denoch ein globales und kein lokales Datenobjekt. Es wurde zwar aus dem globalen Archiv zur Weiterbearbeitung ausgewählt, deshalb aber noch lange nicht aus dem globalen Archiv entfernt. Die **Stack-Objekte** haben lediglich die Aufgabe, eine Verbindung zwischen den Arbeitsstacks sowie den, auf diesen abgelegten, *neuen* Datenobjekten zum globalen Archiv herzustellen, d. h. die **Stack-Objekte** *verweisen* auf Datenobjekte im globalen Archiv. Dieser Hinweis ist deshalb wichtig, weil eine Forderung an den **essentiellen Speicher** zum Ausschnitt **duales Speicherkonzept** lauten wird (siehe auch Seite 38): Jedes Datenobjekt ist entweder lokales oder globales Datenobjekt eines Benutzers, niemals aber beides zugleich oder keines von beidem.

Das **duale Speicherkonzept** ist hauptsächlich Resultat der Umsetzung der Systemziele 1a), 1d, 1d), 1e), 1h) und 1i) auf den Seiten 24 bis 25. Weitere Erläuterungen zu diesem Konzept sind in [BSS93] zu finden. Auch der Problemhinweis aus Abschnitt 2.2.6 ist hier zu beachten.

- WSAV
  WSAV ist die wichtigste Aktivität des **dualen Speicherkonzepts**. Sie speichert ein Datenobjekt aus einem lokalen Arbeitsstack eines Benutzers als ein **globales Objekt** des Benutzers im globalen Archiv ab. Dabei wird an das Da-

tenobjekt ein Datenname vergeben, der eine spätere Wiederauswahl erleichtern soll. Zudem werden die **Vorgänger** des Datenobjekts — diese werden über die Relationshiptyp **Entstanden-aus** verwaltet — im lokalen Arbeitsstack gelöscht. Dabei ist darauf zu achten ist, daß auch die Beziehungsinformation (Relationshipeinträge) der zu löschenden Datenobjekte eliminiert werden. Die entfernten Zwischenresultate können aufgrund der Geschichtsinformation des abgespeicherten Objekts bei Bedarf rekonstruiert werden. Das abgespeicherte Datenobjekt bildet, obwohl es nun gleichzeitig ein globales Datenobjekt des Benutzers ist, das neue definierende Objekt des aktuellen Arbeitsstacks. Es wird ferner das Default-Objekt für weitere Anwendungen. Weitere Informationen zu WSAV sind in [BSS93] zu finden.

Eine weitere Aktivität des **dualen Speicherkonzepts** ist WCLOS. Diese ist aber weit weniger interessant als WSAV, weil ihre Aufgabe lediglich darin besteht, das lokale Archiv eines Benutzers aufzuräumen, d. h. WCLOS löscht noch existierende Datenobjekte im lokalen Archiv und löscht die Informationen zu Stack- und Default-Objekten. Diese Operation wird im folgenden nicht weiter betrachtet.

Zunächst werden die Erläuterungen zum **dualen Speicherkonzept** und der essentiellen Aktivität WSAV durch ein Beispiel ergänzt. Danach wird das Ergebnis der Systemanalyse zum Ausschnitt **duales Speicherkonzept** semiformal beschrieben. Dazu werden die essentielle Aktivität WSAV und deren Unteraktivitäten durch DFD-Diagramme (Abbildungen 2.17, 2.18 und 2.19) beschrieben, während der essentielle Speicher des Ausschnitts durch ein E/R-Diagramm (Abbildung 2.8) präsentiert wird. Die Attribute der einzelnen Entities dieses E/R-Diagramms werden ebenfalls aufgeführt.

2.3.1 Ein Beispiel

Abbildung 2.6 beschreibt eine abstrakte Sicht[12] auf einen möglichen Zustand des essentiellen Speichers zum Ausschnitt **duales Speicherkonzept**. Erkennbar ist eine Trennung der lokalen Archive vom globalen Archiv. Das globale Archiv enthält die Datenobjekte **D9**, **D10**, **D14**, **D15** und **D16**[13] und verwaltet diese in Bezug zu Aufenthalten, Patienten, Hospitälern und Netzwerkadressen (dies ist hier nicht eingezeichnet). Der gewinkelte Pfeil zwischen **D14** und **D15** zeigt an, daß das Datenobjekt **D15** durch Anwendung einer Bearbeitungsoperation (z. B. eines Graustufenfilters) aus Datenobjekt **D14** erzeugt wurde. Die Datenobjekte des globalen Archivs stehen allen Benutzern zur Verfügung.

Die lokalen Archive sind den einzelnen Benutzern des Systems zugeordnet. Jedes dieser Archive enthält möglicherweise n Arbeitsstacks (in diesem Beispiel hat jeder

[12] Die Idee zu dieser abstrakten Sicht wurde motiviert durch das SUNRISE-Systemdokument [BSS93].

[13] Die etwas merkwürdige Numerierung der Datenobjekte und Benutzer erfolgte im Zusammenhang zu Beispielen und Abbildungen, die in dieser Arbeit zwar später vorgestellt, aber vor dieser Abbildung erstellt wurden. Damit die Beziehungen des Beispiels und dieser Abbildungen zu späteren deutlich wird, wurden diese Bezeichnungen auch hier gewählt.

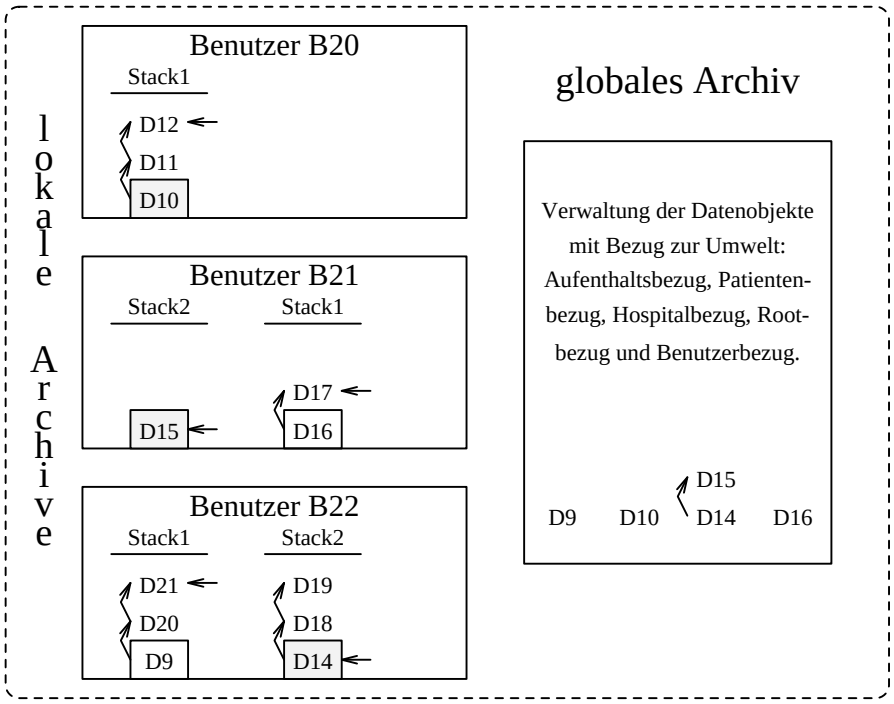

Abbildung 2.6: Beispielzustand 1 zum essentiellen Speicher

Benutzer höchstens zwei Arbeitsstacks geöffnet). Die Stacks enthalten Datenobjekte. Das unterste Element eines Stacks wird als Stack-Objekt[14] bezeichnet. In dieser Abbildung werden die Stack-Objekte durch eine zusätzliche Umrahmung angezeigt. Die grau unterlegten Stack-Objekte sind die aktuellen Stack-Objekte der jeweiligen Benutzer, sie zeigen also die aktuellen Arbeitsstacks der einzelnen Benutzer an. Auch hier gibt ein gewinkelter Pfeil an, welches Datenobjekt zu welchem weiterbearbeitet wurde. Ferner zeigt ein einfacher Pfeil das Default-Objekt eines Stacks an.

Es werden nun die Zustände der lokalen Archive für die einzelnen Benutzern getrennt erläutert:

- Benutzer **B20**
  Dieser hat ursprünglich das Datenobjekt **D10** zur Bearbeitung aus dem globalen Archiv ausgewählt und dadurch einen Arbeitsstack geöffnet. **D10** ist somit Stack-Objekt und gleichzeitig aktuelles Stack-Objekt von Benutzer **B20**. **D10** wurde weiterbearbeitet zu **D11** und **D11** wiederum zu **D12**. **D12** ist das Default-Objekt dieses Stacks für weitere Anwendungen.

- Benutzer **B21**
  Das Datenobjekt **D16** wurde aus dem globalen Archiv ausgewählt und weiterbearbeitet zu **D17**. **D16** ist ein Stack-Objekt von Benutzer **B21** und **D17** das

[14] Siehe Hinweis auf Seite 32: Stack-Objekte sind keine lokalen, sondern (nach wie vor) globale Objekte

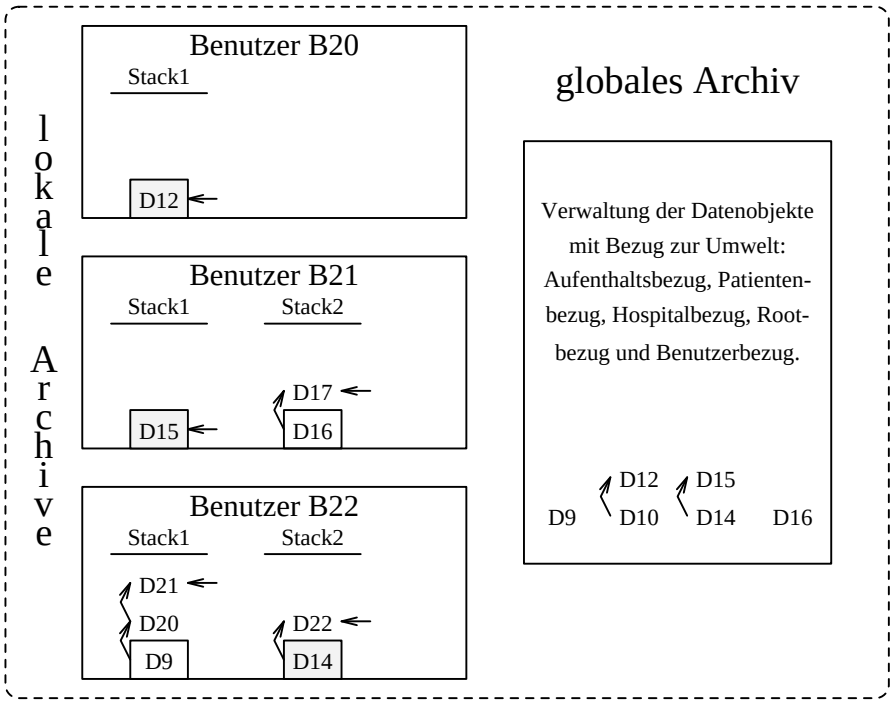

Abbildung 2.7: Beispielzustand 2 zum essentiellen Speicher

Default-Objekt dieses Stacks. Anschließend hat Benutzer **B21** ein weiteres Datenobjekt (**D15**) im globalen Archiv zur Bearbeitung ausgesucht. **D15** wurde dadurch zum Stack-Objekt, zum aktuellen Stack-Objekt und zum Default-Objekt des Stacks 2.

- Benutzer **B22**
  Zunächst wurde **D9** im globalen Archiv ausgewählt, dann weiterbearbeitet zu **D20** und **D21**. Danach wurde **D14** im globalen Archiv ausgesucht und weiterbearbeitet zu **D18** und **D19**. Anschließend hat der Benutzer **B22** zweimal den UNDO-Befehl angewendet. Dadurch wurde der Default-Objekt-Status von **D19** zu **D14** verschoben. Es liegt also folgende Situation vor: **D9** und **D14** sind Stack-Objekte von Benutzer B22, **D14** ist gleichzeitig aktuelles Stack-Objekt. **D21** ist das Default-Objekt von Stack 1, während **D14** das Default-Objekt von Stack 2 ist. Die Anwendung eines weiteren Bearbeitungsschritts durch Benutzer **B22** bezieht sich damit auf das Datenobjekt **D14**, weil dieses das Default-Objekt des aktuellen Stacks ist.

Abbildung 2.7 zeigt die abstrakte Sicht auf einen möglichen Folgezustand zum Zustand aus Abbildung 2.6. Die Situation wird wieder für die einzelnen Benutzer getrennt erläutert:

- Benutzer **B20**
  Benutzer **B20** hat die Operation WSAV angewendet. Dadurch wurde das

Default-Objekt des aktuellen Stacks (nach Datennamevergabe) in das globale Archiv übertragen (dabei wurde vermerkt, welcher Benutzer die Übertragung ausführte). Dort wird es nun als ein Nachfolgeobjekt von **D10** für andere Benutzer zugänglich verwaltet. Das Zwischenresultat **D11** der Bearbeitung von **D10** zu **D12** wurde gelöscht. Obwohl **D12** in das globale Archiv übertragen wurde, definiert es dennoch den aktuellen Arbeitsstack von Benutzer B20 und ist auch Default-Objekt für weitere Anwendungen.

- Benutzer **B21**
  Dieser war passiv und hat keine Systemfunktion angewendet.

- Benutzer **B22**
  Benutzer **B22** hat eine Bearbeitungsoperation angewendet. Dadurch wurde das Default-Objekt des aktuellen Stacks (**D14**) weiterbearbeitet zu **D22**. Die bisherigen Bearbeitungsresultate **D18** und **D19** wurden gelöscht. **D22** wurde zum neuen Default-Objekt dieses Stacks für weitere Anwendungen.

In Abschnitt 3.5 und in Kapitel 4 wird dieses Beispiel erneut betrachtet. Der Benutzer **B22** und dessen erläuterte Aktivität wird dabei aber nicht mehr berücksichtigt. Wichtigster Aspekt ist dann die Anwendung der Operation WSAV durch Benutzer **B20**.

2.3.2 Entity-Relationship-Diagramm zum Ausschnitt *duales Speicherkonzept*

Das E/R-Diagramm in Abbildung 2.8 beschreibt den essentiellen Speicher zum Ausschnitt **duales Speicherkonzept**. Dieser enthält die Entitytypen **Sunrise-Datenobjekt, Benutzer, Aufenthalt, Datenklasse, Root, Hospital** und **Patient**, sowie die Relationshiptypen **Aufenthaltsbezug, Lokal-fuer, Global-fuer, Stack-Objekt-von, Aktuelles-Stack-Objekt-von, Default-Objekt, Entstanden-aus, Datenklassenbezug, Rootbezug, Hospitalbezug** und **Patientenbezug**.

Direkt betroffen von der Aktivität WSAV sind allerdings nur die mit * gekennzeichneten Entity- und Relationshiptypen. Die restlichen wurden mitaufgenommen, weil sie entweder wichtig sind für die Sicherheitsaussage (siehe Abschnitt 2.4) oder weil sie in der aktuellen Implementierung des dualen Speicherkonzepts eine besondere Rolle einnehmen und somit für das Kapitel 4 von Bedeutung sind. Im Vergleich zum E/R-Diagramm in Abbildung 2.5 enthält dasjenige in Abbildung 2.8 Informationen zu schematischen Integritätsbedingungen[15]. Insgesamt sollen folgende schematische Integritätsbedingungen von jedem essentiellen Speicher zu SUNRISE berücksichtigt werden:

- **Grade von Relationshiptypen**
  Sie werden dargestellt durch die Zahlen an den Kanten der Relationshiptypen in Abbildung 2.8.

[15] Einführende Erläuterungen zum Begriff Integritätsbedingungen finden sich auf Seite 19.

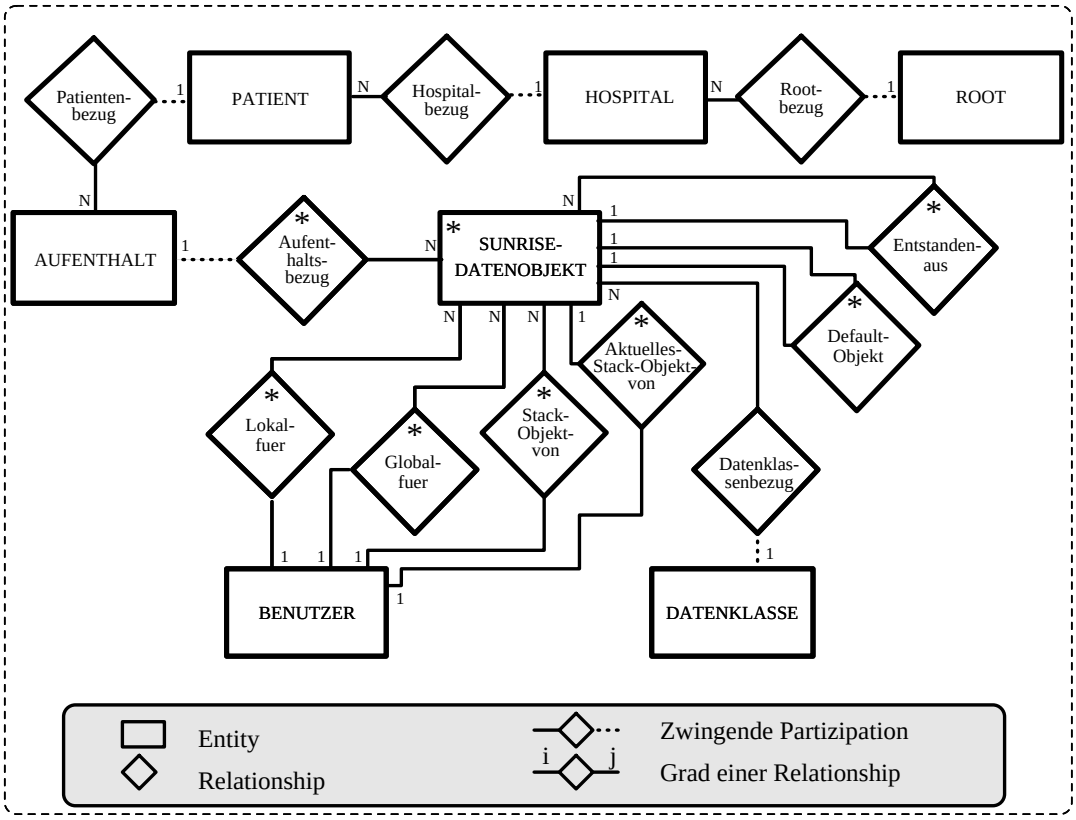

Abbildung 2.8: E/R-Diagramm zum Ausschnitt *duales Speicherkonzept*

- **Zwingende Partizipationen**
  Ausgedrückt wird die zwingende Teilnahme von Entities an einer Relationship durch gestrichelte Kanten in Abbildung 2.8.

- **Schlüsseleigenschaft bei Attributeinträgen**
  Entities sollen anhand ihrer Schlüsselattributeinträge eindeutig identifizierbar sein. Im E/R-Diagramm von Abbildung 2.8 sind diese Bedingungen zwar nicht enthalten, aber in den Diagrammen zu den einzelnen Entitytypen (Abbildungen 2.10 bis 2.16). Dort werden, durch den Zusatz **primary key**, diejenigen Attribute eines Entitytyps ausgezeichnet, die gemeinsam die Schlüsseleigenschaft erfüllen sollen.

- **Zwingende Attributeinträge**
  Ebenfalls in den Diagrammen zu den Entitytypen (Abbildungen 2.10 bis 2.16) werden verschiedene Attribute mit dem Zusatz **mandatory** ausgezeichnet. Dieser Zusatz formuliert die vorgeschriebene Existenz von nichtleeren Attributeinträgen für alle Entities dieses Typs. Diese Attribute müssen also stets einen nichtleeren Attributeintrag aufweisen.

Die spezifischen Integritätsbedingungen, die nicht in den semiformalen Beschreibungen berücksichtigt sind, werden nun informal beschrieben:

1. **spezifische Integritätsbedingung**:
   Zu einem Datenobjekt, das Stack-Objekt eines Benutzers ist, gibt es ein Default-Objekt und jedes Default-Objekt ist einem Stack-Objekt zugeordnet[16].

2. **spezifische Integritätsbedingung**:
   Jedes Datenobjekt ist entweder lokales oder globales Datenobjekt eines Benutzers, niemals aber beides zugleich oder keines von beidem.

Anmerkung: Die Bezeichnungen der Entity- und Relationshiptypen des E/R-Diagramms aus Abbildung 2.8 weichen teilweise von denen in Abbildung 2.5 ab. In Abbildung 2.8 stimmen diese bereits mit den Bezeichnungen der formalen Spezifikation überein. In Abbildung 2.5 sind sie dagegen etwas länger und ausführlicher. Hier spiegelt sich ein allgemeines Problem der Systemanalyse wider. Zu Beginn der Analysetätigkeiten, versucht man ein System anhand informaler oder semiformaler Begriffe zu skizzieren. An die Einschränkungen der Begriffswahl, die einem beim Übergang zur formalen Spezifikation durch eine Spezifikationssprache oder ein Spezifikationssystem auferlegt werden, denkt man dabei natürlich nicht. So ist es nicht verwunderlich, wenn man später feststellt, daß verschiedene Begriffe, die auf informaler Ebene durchaus geeignet erschienen, sich als Bezeichner in einer formalen Spezifikation nicht eignen. Natürlich könnten, nach Fertigstellung der formalen Spezifikation, die Bezeichner der informalen und semiformalen Beschreibungen denen der formalen Ebene angepaßt werden. Dies würde allerdings einen erheblichen Arbeitsaufwand zur Folge haben und die Gedankengänge der informalen Ebene womöglich nachträglich verwischen. Dies ist insbesondere deshalb von Nachteil, weil zu Beginn der Systemanalyse Begriffe und Namen wichtige Informationsträger darstellen (es liegen zu diesem Zeitpunkt noch keinen weiteren Beschreibungen, ob semiformal oder informal, vor). Aus diesem Grund erfolgte in dieser Arbeit keine nachträgliche Anpassung der Begriffe. Auch unterscheiden sich die Bezeichner hier nur geringfügig, so daß keine Mißverständnisse auftreten sollten. Außerdem werden in Abbildung 2.9 die Namensbezüge zwischen den Abbildungen 2.8 und 2.5 angegeben.

2.3.3 Attribute der Entities

Die folgenden Diagramme 2.10 bis 2.16 ergänzen das E/R-Diagramm zum essentiellen Speicher von SUNRISE um weitere Informationen zu den Entitytypen. Ein Entitytyp faßt verschiedene Einzelinformationen des essentiellen Speichers (Attribute) zu einer Einheit zusammen. Bei den **Attributen** wird unterschieden zwischen **Attributbezeichnern, Attributsorten** und **Attributwerten**. Mit dem Attributbezeichner kann der Zugriff auf den Attributwert (der entsprechenden Sorte)

[16] Gemeinsam mit der Integritätsbedingung zum Grad des Relationshiptyps **Default-Objekt** (1:1) folgt, daß jedem Stack-Objekt genau ein Default-Objekt zugeordnet ist und umgekehrt.

Abbildung 2.8	**Abbildung 2.5**
Root	Umweltbezug
Hospital	Umweltbezug
Patient	Umweltbezug
Aufenthalt	Umweltbezug
Sunrise-Datenobjekt	Sunrise-Datenobjekt
Benutzer	Benutzer
Datenklasse	Datenklasse
Rootbezug	Umweltbezug
Hospitalbezug	Umweltbezug
Patientenbezug	Umweltbezug
Aufenthaltsbezug	Datenobjekt hat Umweltbezug
Lokal-fuer	lokales Datenobjekt von
Global-fuer	globales Datenobjekt von
Stack-Objekt-von	Stack-Objekt von
Aktuelles-Stack-Objekt-von	aktuelles Stack-Objekt von
Datenklassenbezug	Datenobjekt hat Datenklasse
Default-Objekt	Default-Objekt zu Stack
Entstanden-aus	ist entstanden aus

Abbildung 2.9: Namensbezüge zwischen den Abbildungen 2.8 und 2.5

erfolgen. Attributsorten tragen den Präfix **A-**. Dadurch werden sie von den ihnen zugrundeliegenden, Domainsorten unterschieden[17]. Als weitere Informationen enthalten die Diagramme die bereits erwähnten Zusätze **primary key** und **mandatory** zur Formulierung von schematischen Integritätsbedingungen. Es werden einige Attribute zusätzlich informal erläutert. Dies erfolgt vollständig für die Attribute des Entitytyps **Sunrise-Datenobjekt** (**S.-Datenobjekt**), danach werden allerdings nur solche erläutert, deren Bedeutung nicht unmittelbar aus dem Attributbezeichner, bzw. der Attributsorte, gefolgert werden kann.

Entitytyp Sunrise-Datenobjekt

Ein Sunrise-Datenobjekt (Abbildung 2.10) enthält Untersuchungsdaten einschließlich dazugehörender Informationen. Es wird entweder beim Import von Untersuchungsdaten kreiert oder entsteht durch Weiterbearbeitung eines anderen Sunrise-Datenobjekts.

- **DatenId**
  Ein Schlüsseleintrag zur Identifizierung.

[17] Auf den Unterschied zwischen Attributsorten und Domainsorten wird in den Abschnitten 2.5 und A.2.2 noch näher eingegangen.

- **Datenname**
  Der Name eines Sunrise-Datenobjekts.

- **Binaerdaten**
  Untersuchungsdaten werden als binäre Datenfelder importiert. Für resultierende Sunrise-Datenobjekte eines Bearbeitungsschritts ist die Binärdateninformation optional. Sie kann aufgrund der Geschichtsinformation aus der Binärdateninformation des ursprünglich importierten Sunrise-Datenobjekts rekonstruiert werden. (Beispiel einer Binärdateninformation: Ein CT[18]-Bild wird als eine 512^2-Matrix mit 8 oder 16 Bit großen Bildelementen dargestellt.)

- **Parameterliste**
  Nur zusammen mit der Parameterinformation können die Binärdaten interpretiert werden. (Beispiel: Die Binärdaten zu einem CT-Bild sind ohne Zusatzinformationen wie Schichtdicke, Tischposition usw. wertlos.)

- **History**
  Der Geschichtsteil eines Sunrise-Datenobjekts umfaßt eine vollständige Information über die Entstehungsgeschichte dieses Objekts. Dadurch ist die Rekonstruierbarkeit eines Sunrise-Datenobjekts aus einem Vorgängerobjekt (z. B. dem importierten Datenobjekt) gewährleistet. Auch die umfangreichen Binärdaten müssen aufgrund des Geschichtsteils nicht für jedes Datenobjekt explizit abgespeichert werden.

- **ROI**
  Regions Of Interest — ein optionaler Eintrag, der es ermöglicht, einzelne Bereiche des Binärdatenteils auszuzeichnen.

- **CreationInfo**
  Hier werden zusätzliche Information, wie z. B. das Entstehungsdatum, vermerkt.

Entitytyp Benutzer

Der Entitytyp Benutzer (Abbildung 2.11) verwaltet Informationen zu den Systembenutzern von SUNRISE.[19]

Entitytyp Datenklasse

Der Entitytyp Datenklasse (Abbildung 2.12) ist sehr wichtig für die Anwendung von Bearbeitungsoperationen auf Sunrise-Datenobjekte. Über den Datenklassenbezug eines Sunrise-Datenobjekts wird festgestellt, ob eine Operation anwendbar ist. Die unterschiedlichen vorgegebenen Datenklassen sind im Systemdokument [BSS93]

[18] Computertomograph

[19] In der aktuellen Implementierung erfolgt die Verwaltung von Benutzerinformationen teilweise über das UNIX-Datei- bzw. Betriebssystem.

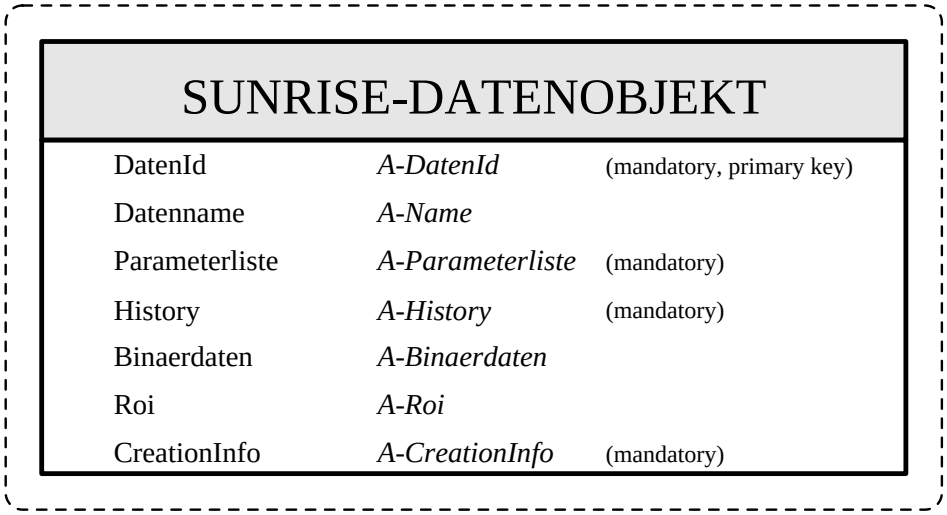

Abbildung 2.10: Attribute zum Entitytyp Sunrise-Datenobjekt

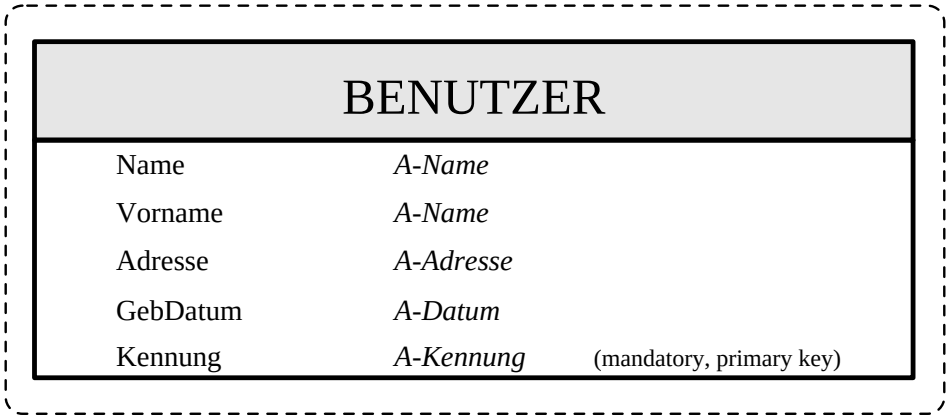

Abbildung 2.11: Attribute zum Entitytyp Benutzer

Abbildung 2.12: Attribute zum Entitytyp Datenklasse

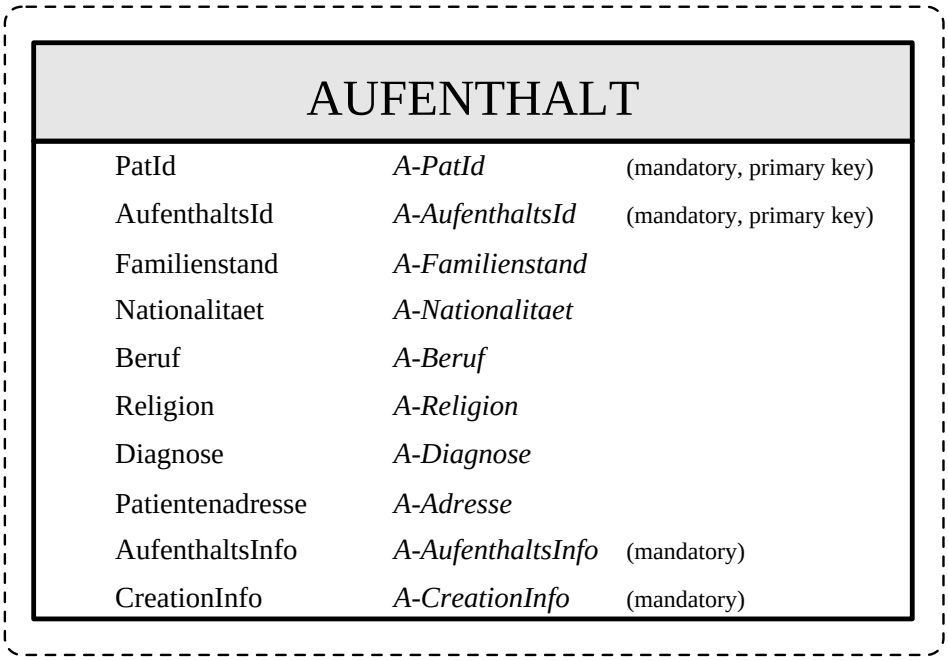

Abbildung 2.13: Attribute zum Entitytyp Aufenthalt

dokumentiert. Es können, im Zusammenhang mit der Entwicklung neuer Bearbeitungsoperationen, zur Laufzeit auch neue Datenklassen von den Entwicklern definiert werden. Das einzige Attribut einer Datenklasse enthält den Klassennamen.[20]

- **Klassenname**
  Mögliche Klassennamen sind: Spektrum, Image, Profile, Signal usw. Anhand dieser Datenklassen werden logisch unterschiedlich zu handhabende Sunrise-Datenobjekte vom SUNRISE-System unterschieden.

Entitytyp Aufenthalt

Der Entitytyp Aufenthalt (Abbildung 2.13) enthält die Informationen zu den Hospitalaufenthalten von Patienten. Bei jeder Neuaufnahme eines Patienten wird ein neuer Vertreter dieses Entitytyps in der Datenbank angelegt.

- **PatId**
  Hinweis auf den Patienten, dem dieser Aufenthalt zugeordnet ist.
- **AufenthaltsId**
  Durchnumerierung der Aufenthalte zu einem Patienten. Diese beiden ersten

[20] In der Implementierung werden Datenklassen als zusätzliches Attribut der S.-Datenobjekte vermerkt (siehe auch Anmerkung auf Seite 28).

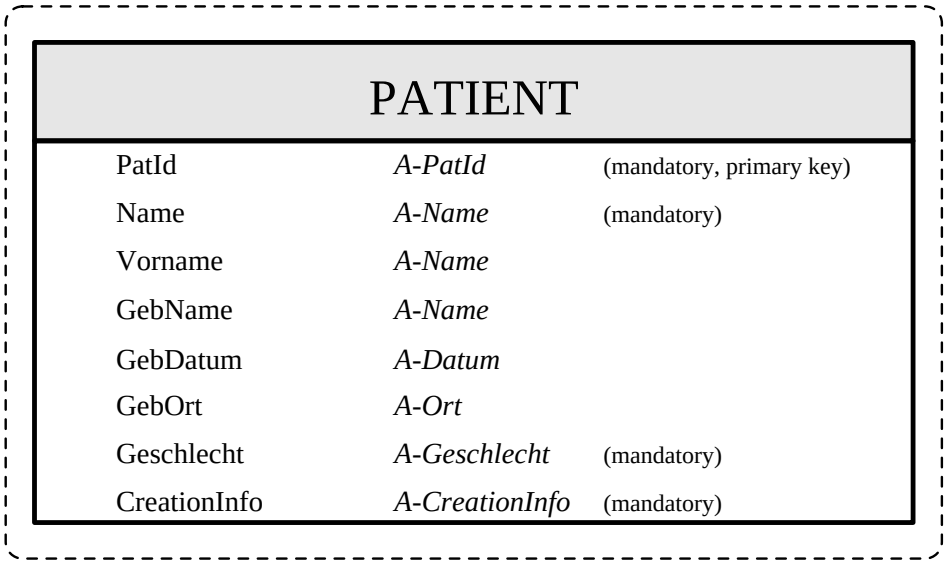

Abbildung 2.14: Attribute zum Entitytyp Patient

Attribute bilden gemeinsam einen eindeutigen Schlüssel für Träger des Typs Aufenthalt.

- Die Attribute **Familienstand, Nationalitaet, Beruf, Religion, Diagnose** und **Patientenadresse** werden hier angefügt und nicht etwa bei Patient, weil diese sich für denselben Patienten von einem Hospitalaufenthalt zum anderen verändern können.

- **AufenthaltsInfo**
  Informationen über den Aufenthaltszeitraum, die Stations- und Zimmernummer, den behandelnden Arzt usw.

- **CreationInfo**
  Siehe unter Sunrise-Datenobjekt.

Entitytyp Patient

Der Entitytyp Patient (Abbildung 2.14) speichert die Informationen zu den Patienten. Für jeden realen Patienten wird nur ein Vertreter des Entitytyps Patient in der Datenbank angelegt.

Entitytyp Hospital

Informationen zu verschiedenen Hospitälern (oder auch Abteilungen) werden durch den Entitytyp Hospital (Abbildung 2.15) verwaltet.

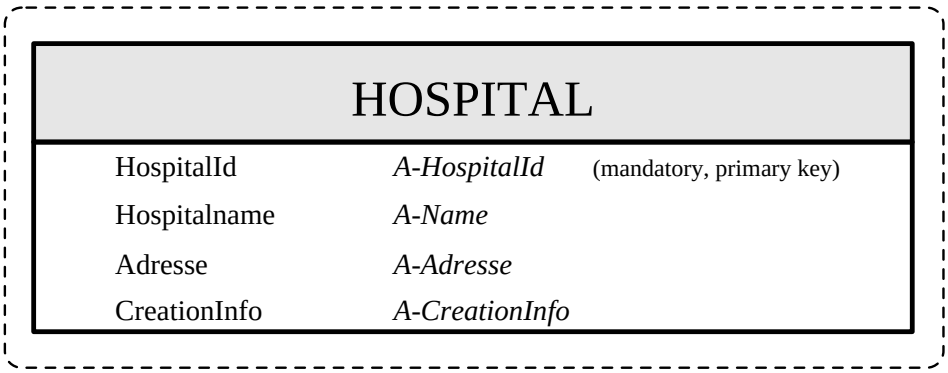

Abbildung 2.15: Attribute zum Entitytyp Hospital

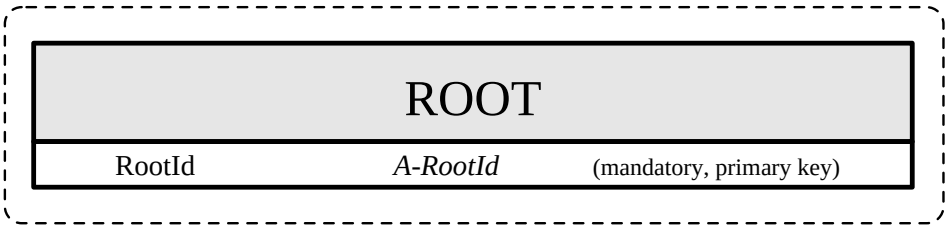

Abbildung 2.16: Attribute zum Entitytyp Root

Entitytyp Root

Der Entitytyp Root (Abbildung 2.16) verwaltet verschiedene Netzwerkadressen. Für die weitere Betrachtung ist er, wie auch der Entitytyp Hospital, nur von geringer Bedeutung.

Anmerkung: Die Wahl der Attributbezeichner erfolgte möglichst in Anlehnung an die Wortwahl in den Systemdokumenten zu SUNRISE.

2.3.4 Datenflußdiagramme zur essentiellen Aktivität WSAV

Die essentielle Aktivität WSAV (**Speichere lokales Datenobjekt global ab**) wird nun durch Datenflußdiagramme und zusätzlichen Prosatext erläutert.

WSAV

Zum besseren Verständnis des DFD-Diagramms in Abbildung 2.17 und der folgenden, informalen Erläuterungen wird dem Leser empfohlen, sich die Architektur des essentiellen Speichers zum betrachteten Ausschnitt aus Abbildung 2.8 in Erinnerung zu rufen.

Als Argumente werden der essentiellen Aktivität WSAV die **Benutzerkennung** des aufrufenden Benutzers und ein **Name** für das abzuspeichernde Datenobjekt

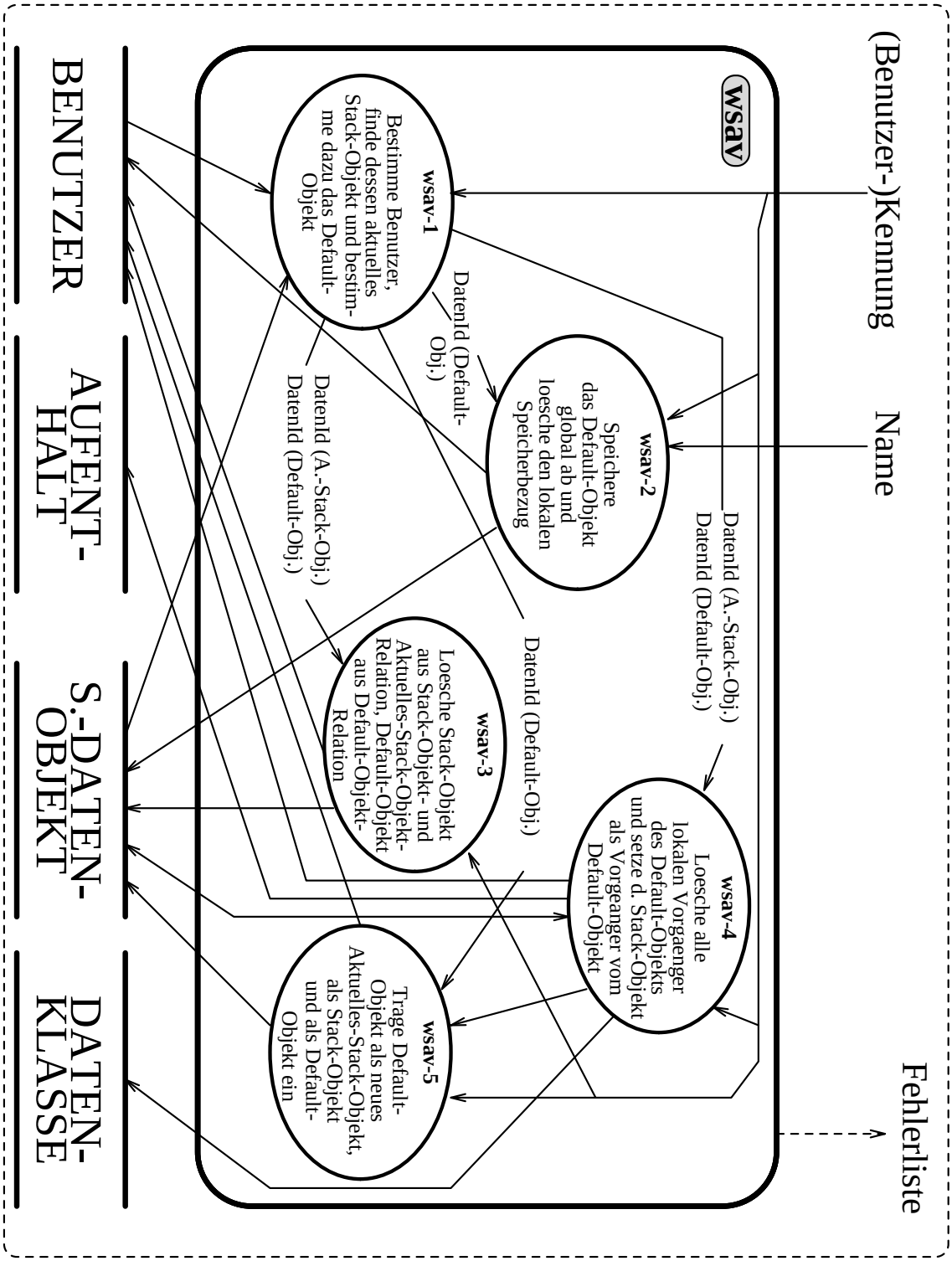

Abbildung 2.17: DFD-Diagramm zur essentiellen Aktivität WSAV

übergeben. In der Implementierung erfolgt die Übergabe dieser Argumente zeitlich versetzt. Hier wird von solchen zeitlichen Aspekten jedoch abstrahiert, und die beiden Argumente werden gleichzeitig an WSAV übergeben. Falls WSAV auf den Eingaben fehlerfrei ablaufen kann, wird als Ergebnis ein veränderter essentieller Speicher (Datenbank) und die leere Liste von Fehlermeldungen zurückgeliefert. Falls WSAV nicht fehlerfrei ablaufen kann, wird der unveränderte essentielle Speicher und eine Liste mit entsprechenden Fehlermeldungen zurückgeliefert. Die logisch fehlerhaften Eingaben[21] werden durch eine gesonderte Fehleranzeige-Operation (nicht eingezeichnet) aufgedeckt.

Falls keine logisch fehlerhafte Eingabe angezeigt wird, bestimmt die Unteraktivität WSAV-1 zum Benutzer, der durch die **Kennung** identifiziert wird, das aktuelle Stack-Objekt. Dieses definiert den aktuellen Arbeitsstack des Benutzers. Zu jedem Arbeitsstack existiert ein Default-Objekt. Dieses Objekt ist z. B. Default-Objekt eines weiteren Bearbeitungsschrittes, aber auch das Default-Objekt der Anwendung der Operation WSAV — also zur globalen Abspeicherung. WSAV-1 bestimmt ausgehend vom aktuellen Stack-Objekt das dazugehörige Default-Objekt, und leitet die Schlüsselinformationen (DatenId) beider Objekte an die Unteraktivitäten WSAV-3 und WSAV-4 weiter. An die Unteraktivitäten WSAV-2 und WSAV-5 übergibt sie lediglich die DatenId des Default-Objekts.

Die Unteraktivität WSAV-2 registriert die Veränderung des lokalen Speicherbezugs des Default-Objekts in einen globalen. Dazu bedient sie sich der Kennungsinformation aus der Argumentliste. Ferner vergibt WSAV-2 dem Default-Objekt den Namen aus der Argumentliste, überprüft dabei aber die Namenseindeutig bzgl. der Objekte, die ebenfalls aus dem Vorgängerobjekt des Default-Objekts abgleitet wurden und bereits im globalen Archiv abgelegt sind[22].

WSAV-3 löscht das Default-Objekt aus den Relationships Stack-Objekt-von[23] und Aktuelles-Stack-Objekt-von. Außerdem wird das Default-Objekt aus der Default-Objekt-Relationship entfernt.

WSAV-4 räumt den privaten Arbeitsstack des Benutzers bis zum Default-Objekt auf. Nur Nachfolgeobjekte des Default-Objekts bleiben erhalten, Vorgänger werden eliminiert. Weil die Vorgänger des Default-Objekts entfernt werden, muß das bisherige aktuelle Stack-Objekt nun als Vorgänger des Default-Objekts registriert werden. Alle eliminierten Datenobjekte müssen natürlich auch aus den Relationships, an denen sie teilnehmen, entfernt werden.

WSAV-5 trägt das Default-Objekt als neues aktuelles Stack-Objekt und damit auch als Stack-Objekt des Benutzers (identifiziert durch die Kennung) ein.

[21] Es wird davon ausgegegangen, daß die Eingaben syntaktisch korrekt sind, also z. B. keine Benutzerkennung eingegeben wird, die keinem aktuellen Benutzer entspricht. Die logisch fehlerhaften Argumentkonstellationen werden auf Seite 72 noch ausführlich erläutert.

[22] Dadurch wird im Zusammenhang mit der, hier nicht betrachteten, Aktivität **Wähle Datenobjekt zur Bearbeitung aus** eine benutzerfreundliche Datenobjektauswahl nach den Datennamen ermöglicht.

[23] Jedes Stack-Objekt definiert einen privaten Arbeitsstack. Es gibt für jeden Benutzer aber nur einen aktuellen Arbeitsstack und dieser wird durch das aktuelle Stack-Objekt festgelegt. Damit ist jedes aktuelle Stack-Objekt zugleich auch ein Stack-Objekt.

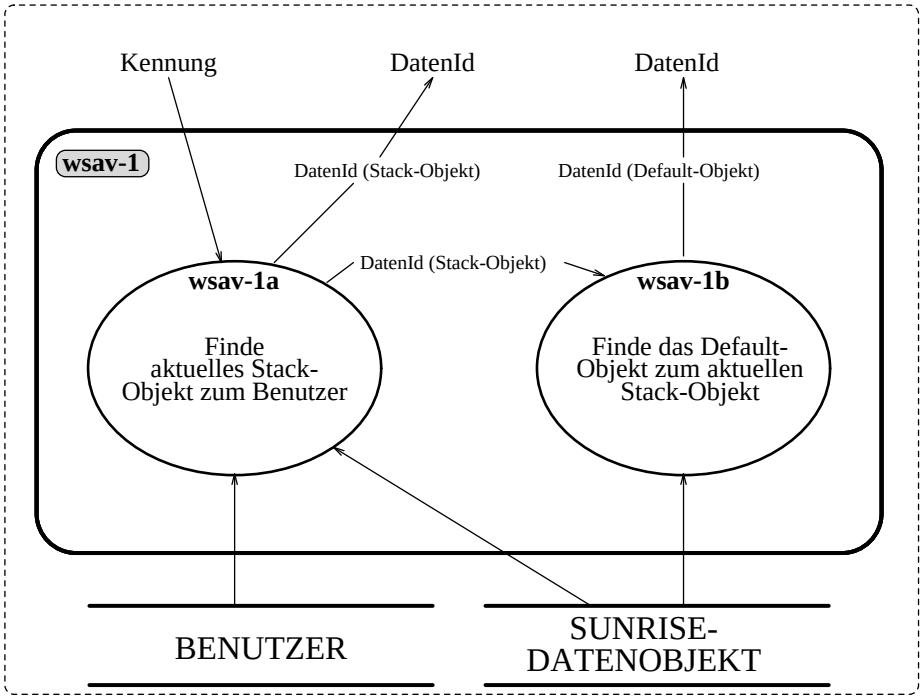

Abbildung 2.18: DFD-Diagramm zur Unteraktivität WSAV-1

Exemplarisch werden die Unteraktivitäten WSAV-1 und WSAV-4 näher erläutert. Für sie werden erneut DFD-Diagramme angegeben, die durch weiteren Prosatext ergänzt werden. Analog hätte dies auch für WSAV-2, WSAV-3 und WSAV-5 erfolgen können. Diese Vorgehensweise könnte bei sehr komplexen Aktivitäten auf noch tieferen Ebenen fortgesetzt werden, bis eine genügend verständliche Beschreibung der essentiellen Aktivität erreicht wird.

WSAV-1

Die Unteraktivität WSAV-1 (siehe Abbildung 2.18) zerfällt selbst wieder in die Unteraktivitäten WSAV-1a und WSAV-1b. WSAV-1a bestimmt das aktuelle Stack-Objekt zum Benutzer und gibt dessen DatenId sowohl an WSAV-1b als auch nach außen ab. Zur Bestimmung des aktuellen Stack-Objekts greift WSAV-1a lesend auf die Relationship Aktuelles-Stack-Objekt-von zu (siehe Abbildung 2.8). WSAV-1b bestimmt das Default-Objekt zu dem, durch WSAV-1a ermittelten, aktuellen Stack-Objekt durch einen lesenden Zugriff auf die Relationship Default-Objekt und gibt dieses nach außen ab.

WSAV-4

Auch WSAV-4 (siehe Abbildung 2.19) zerfällt in zwei Unteraktivitäten: WSAV-4a und WSAV-4b. WSAV-4a eliminiert alle lokalen Vorgänger des Default-Objekts. Die Vorgänger eines Datenobjekts werden über die Ist-Entstanden-aus-Relationship ver-

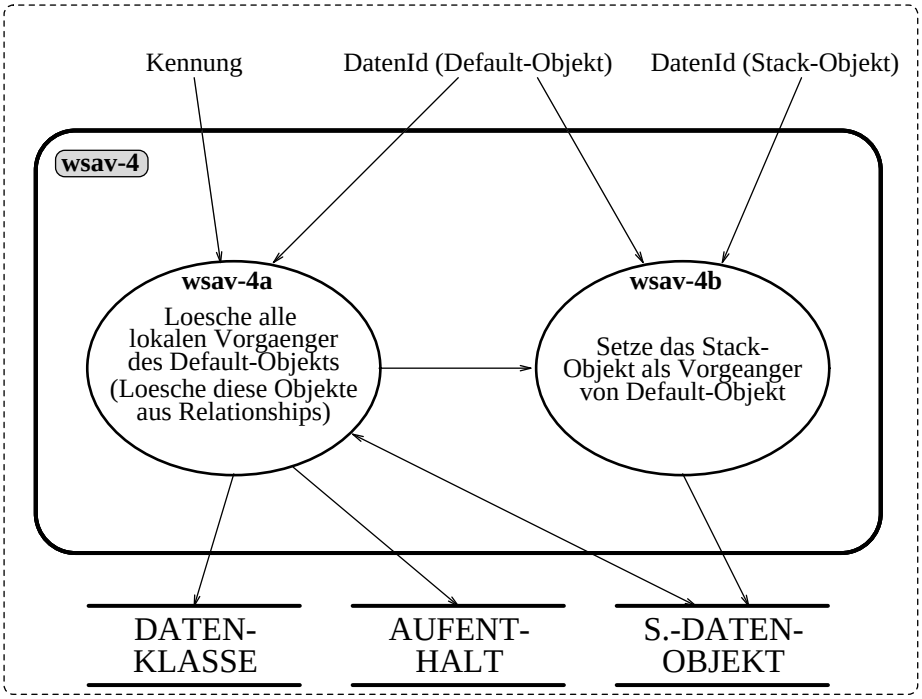

Abbildung 2.19: DFD-Diagramm zur Unteraktivität WSAV-4

waltet. Beim Eliminieren eines Datenobjektes muß darauf geachtet werden, daß dieses Datenobjekt auch aus den Relationships an denen es teilnimmt, eliminiert wird. Die betroffenen Relationshiptypen sind: Entstanden-aus, Lokal-fuer, Datenklassenbezug und Aufenthaltsbezug. WSAV-4b notiert anschließend das Stack-Objekt als direkten Vorgänger des Default-Objekts.

2.4 Eine Sicherheitsaussage

Auf den Begriff der Sicherheit — im Zusammenhang mit Systemen der Informationstechnik — wird in Kapitel 5 detaillierter eingegangen. Hier soll exemplarisch eine Sicherheitsaussage ausgewählt werden, die Bestandteil eines Sicherheitsmodells zu SUNRISE sein könnte. Ausgangspunkt zur Formulierung von Sicherheitsaussagen ist die Analyse der Bedrohung der Sicherheit eines Systems.

Zur weiteren Behandlung in dieser Arbeit wurde eine Aussage ausgewählt, deren Bezug zum wichtigen Datenintegritätsaspekt[24] einer medizinischen Datenbank deutlich erkennbar ist. Sie beschäftigt sich mit der Zuordnung von Untersuchungsdaten zu Patienten. Es ist leicht ersichtlich, daß eine falsche Zuordnung von Untersuchungsdaten zu einem Patienten fatale Folgen haben könnte. Deshalb muß eine medizinische Datenbank wie SUNRISE sicherstellen, daß die Zuordnung eines Da-

[24] Unter Integrität versteht man den *Schutz vor unbefugter Veränderung von Informationen*, siehe auch Kapitel 5, Seite 93

tenobjektes zu einem Patienten niemals bewußt oder unbewußt (z. B. durch die Anwendung der Operation WSAV) verfälscht werden kann. Die Sicherheitsaussage lautet somit:

- **Die Anwendung von WSAV verfälscht nicht die Zuordnung von Datenobjekten zu Patienten.**

Wie dem E/R-Diagramm von Abbildung 2.8 zu entnehmen ist, wird der Bezug von Datenobjekten zu Patienten durch die beiden Relationshiptypen Aufenthaltsbezug und Patientenbezug realisiert. Konkret gilt: Jedes Datenobjekt ist genau einem Aufenthalt zugeordnet — dies geht aus den Integritätsbedingungen (Grad 1:N) zum Relationshiptyp Aufenthaltsbezug hervor. Analog gehört jeder Aufenthalt zu genau einem Patienten. Aufbauend auf dieser Feststellung, kann die obige Sicherheitsaussage nun weiter präzisiert werden.

- **Die Anwendung der Operation WSAV verfälscht nicht die Zuordnung von Aufenthalten zu Patienten (Relationshiptyp Patientenbezug) und**
  nicht die Zuordnung von Datenobjekten zu Aufenthalten (Relationshiptyp Aufenthaltsbezug).

Was aber bedeutet es, eine Zuordnung (d. h. Relationship) nicht zu *verfälschen*? Dieser Begriff soll weiter präzisiert werden. Die Operation WSAV verfälscht eine Zuordnung von Datenobjekten zu Aufenthalten genau dann, wenn nach der Ausführung von WSAV ein Datenobjekt **d** einem Aufenthalt **a** zugeordnet ist, während vor der Ausführung ein Aufenthaltsbezug von **d** zu einem Aufenthalt **a'** bestand und ferner **a** ungleich **a'** ist. Analog kann definiert werden, wann WSAV die Zuordnung von Aufenthalten zu Patienten verfälscht. Kehrt man die Aussage nun um, und definiert, wann die Operation WSAV eine Zuordnung nicht verfälscht, so muß man darauf achtgeben, daß möglicherweise Datenobjekte bei Anwendung von WSAV gelöscht werden. Es wäre zu restriktiv zu definieren, daß WSAV eine Zuordnung von Datenobjekten zu Aufenthalten genau dann nicht verändert, wenn für alle Datenobjekte **d** gilt: Wenn **d** vor der Anwendung von WSAV genau einem Aufenthalt **a** zugeordnet ist, dann gilt dies auch nach der Anwendung von WSAV. Denn alle durch WSAV eliminierten Datenobjekte würden gegen diese Forderung verstoßen. Man muß also in der Aussage berücksichtigen, daß Datenobjekte durch WSAV gelöscht werden können. Für den Bezug von Aufenthalten zu Patienten stellt sich dieses Problem aber nicht, weil weder Aufenthalte noch Patienten durch WSAV gelöscht werden. Die endgültige informale Version der Sicherheitsaussage lautet damit:

- **Für alle Aufenthalte a und Patienten p gilt: wenn vor der Ausführung von WSAV a in der Relationship Patientenbezug zu p steht, dann gilt dies auch nach der Ausführung von WSAV,**
  und
  für alle Datenobjekte d und Aufenthalte a gilt: wenn vor der Ausführung von WSAV d in der Relationship Aufenthaltsbezug zu

a steht, dann gilt dies auch nach der Ausführung von WSAV oder Datenobjekt d wurde aus dem essentiellen Speicher entfernt.

Die hier informal beschriebene Sicherheitsaussage wird im folgenden Kapitel 3 formal spezifiziert. Damit eröffnet sich die Möglichkeit eines formalen Beweises der Gültigkeit dieser Aussage für die ebenfalls im folgenden Kapitel formal spezifizierte Operation WSAV.

2.5 Anmerkungen und Namenskonventionen

Die semiformale Beschreibung der logischen Essenz zu SUNRISE, die in diesem Kapitel vorgestellt wurde, ist Grundlage für die formale Spezifikation in Kapitel 3 bzw. in Anhang A. Einige wichtige Begriffe, die auch dort Verwendung finden, werden kurz zusammengefaßt. Die folgenden Begriffe sind analog verwendet worden im KORSO-HDMS-A-Fallbeispiel (siehe [Het93]). In [Aut93] finden sich zum Teil mathematisch rigorose Definitionen der hier nur informal erläuterten Begriffe.

- **Entities und Entitytypen**
  Es ist notwendig zwischen Entitytypen und Entities streng zu unterscheiden. Die Abbildung 2.15 beschreibt einen **Entitytyp Hospital**. Analog zum Begriff **Entitytyp** wird auch der Begriff **Entitysorte** verwendet. Die Instantiierung eines **Entitytyps** mit Attributwerten liefert eine **Entity** — einen Vertreter des Entitytyps.

- **Relationships und Relationshiptypen**
  Auch hier ist die Trennung beider Begriffe wichtig. In Abbildung 2.5 und 2.8 werden **Relationshiptypen** (**Relationshipsorten**) beschrieben. Erst eine Instantiierung dieser Typen durch konkrete Relationen liefert Vertreter dieses Typs — also **Relationships**.

- **Attributtypen, Attribute und Attributbezeichner**
  Auch bei **Attributen** wird zwischen Typen und Vertretern unterschieden. Anstelle von **Attributtyp** wird häufiger der Begriff **Attributsorte** verwendet. **Attributbezeichner** dienen als Selektoren für Zugriffe auf bestimmte **Attributwerte**. Beispiel: Abbildung 2.10 zeigt einen **Attributbezeichner Datenname**. Mit diesem kann für eine konkrete Entity des Typs **Sunrise-Datenobjekt** auf dessen **Attributwert** (oder einfach **Attribut**), z.B. "Röntgenbild-1", der **Attributsorte A-Name** zugegriffen werdem.

- **Domainsorten**
  Domainsorten bilden die Grundlage der **Attributsorten**. Eine bestimmte **Attributsorte** unterscheidet sich von der ihr zugrundeliegenden **Domainsorte** durch genau einen Träger — dem leeren Attributeintrag. Beispiel: Die **Attributsorte A-Datum** in Abbildung 2.14 erweitert eine **Domainsorte Datum** um einen **leeren Attributeintrag**.

- **Datenbank**
  Der **essentielle Speicher** des SUNRISE-Systems wird in der formalen Spezifikation auch **Datenbank** genannt. Diese Namenskonvention wird in Anlehnung an das KORSO-HDMS-A-Fallbeispiel (siehe [Het93] und [Huß93]), sowie an [Hec93], [Aut93] und [Ben93c] getroffen. Die Sorte **Db** (für **Datenbank**) entspricht also dem, in diesem Kapitel vorgestellten, **essentiellen Speicher**, bzw. einem Ausschnitt des essentiellen Speichers, von SUNRISE. Verwaltet werden in der **Datenbank** die essentiellen Informationen des SUNRISE-Systems. Die **Datenbank** wird aufgeteilt in eine **Entity-Komponente** und eine **Relationship-Komponente**.

- **Entity-Komponente**
  Entities werden in der **Entity-Komponente** der Datenbank verwaltet.

- **Relationship-Komponente**
  Relationships werden in der **Relationship-Komponente** der Datenbank verwaltet.

Kapitel 3

Formale Anforderungsspezifikation

In Kapitel 2 wurde das Ergebnis einer Systemanalyse zum SUNRISE-System vorgestellt. Die analysierte logische Essenz des Systems (genauer: des Ausschnitts *duales Speicherkonzept*) wurde dokumentiert mit Hilfe semiformaler Methoden — nämlich mit E/R- und DFD-Diagrammen. Bis hierher stimmt die Vorgehensweise mit konventionellen Methoden der Praxis überein. Der nun folgende Schritt weicht erstmals von der traditionellen Software-Entwicklung ab. Nachdem die logische Essenz des Systems ausgearbeitet ist, wird diese in einer formalen Spezifikationssprache beschrieben. Auch die Integritätsbedingungen (siehe Seite 36) und die exemplarische Sicherheitsaussage (siehe Seite 48) werden dabei berücksichtigt. Allerdings erfolgt keine eigenständige Spezifikation eines Sicherheitsmodells, wie z. B. in [ITS91] (siehe Kapitel 5) vorgeschlagen, sondern die exemplarische Sicherheitsaussage zur Operation WSAV (siehe Seite 48) wird in Form von Exportaxiomen der formalen Anforderungspezifikation hinzugefügt.

Zur Erstellung der formalen Spezifikation zum essentiellen Speicher von SUNRISE wird in dieser Arbeit das Transformationsschema [Aut93] angewendet. Nach diesem Schema, das in Anlehnung an [Het93] und [NW93] innerhalb des KORSO-Projekts entwickelt wurde, ist es möglich, E/R-Diagramme in OBSCURE-Spezifikationen zu transformieren. Auch die Erstellung der formalen Spezifikation zur essentiellen Aktivität WSAV hält sich streng an die Vorgaben des Kapitels 2. Allerdings wird diese Spezifikation nicht nach einem vorgegebenen Schema entwickelt, sondern **frei** erstellt.[1] Insgesamt wird also verdeutlicht, wie ein herkömmlicher, praxisnaher Software-Entwicklungsprozeß um eine formale Ebene ergänzt werden kann.

Ein weiterer Aspekt dieses Kapitels betrifft die Untersuchung der Praxisrelevanz der Spezifikationssprache sowie des Systems OBSCURE an einem weiteren Beispiel. Bisherige Fallstudien dazu sind: KORSO-HDMS-A (siehe [Hec93], [Aut93] und [Ben93c]), LEX (siehe [AFHL92]), UNIX (siehe [ABH92]), EXITUS-Fopra (siehe [WH89]) und Modellierung von Zugriffsrechten (siehe [Huf94] bzw. [Lev93]).

[1] [NW93] beschreibt auch die Möglichkeit einer Transformation von DFD-Diagrammen in formale (SPECTRUM-)Spezifikationen bzw. Spezifikationsfragmente.

3.1 Gründe für eine formale Spezifikation

Anhand einiger Stichpunkte werden Vorteile einer formalen Spezifikation von Systemanforderungen vorgestellt:

1. Während informale und semiformale Beschreibungen möglicherweise mehrdeutig und damit mißverständlich sind, ist eine formale Spezifikation mathematisch präzise, vorausgesetzt, der formalen Spezifikationssprache liegt eine formale Semantik zugrunde. Die Semantik der hier verwendeten Spezifikationssprache OBSCURE wird in [LL93] definiert.

2. Durch die formale und mathematisch präzise Beschreibung eröffnet sich die Möglichkeit zur formalen Verifikation von Aussagen zum spezifizierten System, sowie der Konsistenz der Spezifikation.

3. Führt man eine formale Software-Entwicklung nach der Wasserfallmethode durch, kann die Korrektheit der einzelnen Verfeinerungsschritte ebenfalls formal bewiesen werden. Die Aussagen, die auf einer abstrakten Ebene formuliert und verifiziert wurden (siehe Stichpunkt 2.), können dann auf die verfeinerten Entwicklungsebenen übertragen werden.

4. Verwendet man zur formalen Spezifikation eine algorithmische Spezifikationssprache (z. B. OBSCURE), so besteht die Möglichkeit eines Rapid-Prototypings. Die abstrakte Beschreibung kann dazu als ein Prototyp angesehen werden, der auf seine Eignung, die Systemziele tatsächlich zu realisieren (Validierung), in einer Interpretersitzung überprüft werden kann. Siehe hierzu auch Abschnitt 3.5.

5. Zur Erstellung einer formalen Spezifikation wird der Spezifizierer gezwungen, sich sehr gründlich mit der Beschreibung des geplanten Systems auseinanderzusetzen. Nachlässigkeiten, die sich vor allem bei informalen Beschreibungen leicht einschleichen, können frühzeitig aufgedeckt werden (spätestens beim Rapid-Prototyping oder beim Versuch einer formalen Verifikation).

3.2 Struktur der Spezifikation

Bevor die Spezifikation selbst vorgestellt wird, gibt dieser Abschnitt einen Überblick zum Aufbau der Gesamtspezifikation. Die Grobgliederung der Spezifikation ist in engem Zusammenhang zu den Ergebnissen der Systemanlyse in Kapitel 2 zu sehen. Dort wird unterschieden zwischen einem essentiellen Speicher und den essentiellen Aktivitäten bzw. der einen essentiellen Aktivität WSAV. Auch die folgende formale Spezifikation richtet sich nach dieser Aufteilung. Sie unterscheidet zwischen einer Datenmodellebene, deren wesentlicher Bestandteil die Spezifikation des essentiellen Speichers (der Datenbank) ist und einer Aktivitätenebene, in der die essentielle Aktivität WSAV spezifiziert wird. Die Spezifikationen der Integritätsbedingungen (OK-Prädikat) sowie der Sicherheitsaussage wurden ebenfalls mit in

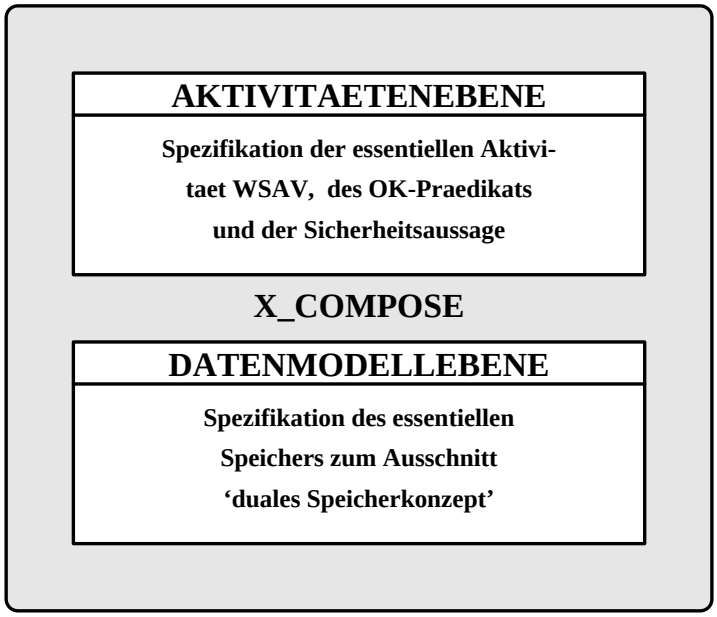

Abbildung 3.1: Struktur der Gesamtspezifikation

die Aktivitätenebene aufgenommen. Abbildung 3.1 beschreibt die angesprochene Gesamtstruktur. Die Aktivitätenebene und die Datenmodellebene werden durch das OBSCURE-Sprachkonstrukt **X_COMPOSE** horizontal miteinander verknüpft, d. h. die Aktivitätenebene importiert einen Teil der Operationen, die von der Datenmodellebene eingeführt und nach außen exportiert werden.

3.3 Spezifikation der Datenmodellebene

Die Spezifikation der Datenmodellebene erfolgt nach dem Transformationsschema von E/R-Diagrammen in OBSCURE-Spezifikationen [Aut93]. Dieses Schema ist innerhalb des KORSO-HDMS-A-Projektes in Anlehnung an die Arbeiten [Het93] und [NW93] entwickelt worden und wird in dieser Arbeit erstmals angewendet. Nach [Aut93] kann die Übersetzung eines E/R-Diagramms in eine OBSCURE-Spezifikation prinzipiell automatisch erfolgen, doch zur Zeit steht zu diesem Zweck kein System zur Verfügung. In dieser Arbeit erfolgte die Übersetzung mühsam von Hand.

Leider ist die nach [Aut93] schematisch entwickelte Spezifikation recht unübersichtlich und teilweise schwer nachzuvollziehen. Weil es aber zum Verständnis der weiteren Arbeit nicht unbedingt notwendig ist, diese Spezifikation in allen Einzelheiten zu überschauen, wird sie im Anhang A vorgestellt. Dort wird nicht die gesamte Datenmodellebene zu SUNRISE formal spezifiziert, sondern nur der, für uns interessante, Auschnitt **duales Speicherkonzept**. Argumente der Transformation sind das E/R-Diagramm in Abbildung 2.8 und die dazugehörenden Erläuterungen der Entitytypen durch die Abbildungen 2.10 bis 2.16. Auch wird das Schema nach [Aut93] nicht in aller Ausführlichkeit angewendet, sondern es werden einige, für die-

sen Fall uninteressante, Bestandteile ausgeklammert. Dadurch kann unnötige Arbeit und ein ebenso unnötiges Aufblähen der Spezifikation vermieden werden.

Wichtig für das Verständnis der Spezifikation der Aktivitätenebene sind lediglich einige wenige der insgesamt in der Datenmodellebene spezifizierten Sorten und Operationen. Diese sind in einem Modul SCHNITTSTELLE aufgelistet, der die Schnittstelle zwischen Datenmodellebene und der Aktivitätenebene beschreibt. Die Sorten und Operationen des Moduls SCHNITTSTELLE werden in dieser Importliste nur informal, d. h. durch zusätzlichen Kommentartext erläutert, ihre formale Spezifikation kann dem Anhang A entnommen werden.

Hinweis: Obwohl in Kapitel 1.5 das Literarische Spezifizieren bereits angesprochen wurde, soll hier erneut auf den Unterschied zwischen reinem OBSCURE-Spezifikationstext und zusätzlichem Kommentar hingewiesen werden. Der Beginn eines OBSCURE-Moduls wird durch einen Kommentartext '*Modul: <Modulname>*' und das Ende durch '*Ende Modul <Modulname>*' angezeigt. Zwischen diesen beiden Zeilen befindet sich reiner OBSCURE-Spezifikationstext, der durch weitere informale Kommentare und Hinweise angereichert wurde. Alle Kommentare und Hinweise sind durch die Schriftart *Italic* vom reinen OBSCURE-Spezifikationstext (und vom übrigen Text der Arbeit) optisch abgesetzt. Die Unterscheidung zwischen OBSCURE-Spezifikationstext und erläuternden Kommentaren sollte deshalb nicht schwerfallen.

Modul: SCHNITTSTELLE
Autor: cb *Datum: 23. Januar 94*
Inhalt: Sorten und Operationen der Datenmodellebene, die in der Spezifikation der Aktivitätenebene verwendet werden.

IMPORTS
SORTS

Datenbanksorte (Sorte des essentiellen Speichers). Eine Datenbank ist ein Paar bestehend aus einer Entity- und einer Relationshipkomonente.

Db

Entitykomponente der Datenbank. Eine Entity-Komponente einer Datenbank ist ein 7-Tupel über den Ansammlungssorten (Mengensorten) Benutzer-coll, Datenklasse-coll, Datenobjekt-coll, Aufenthalt-coll, Patient-coll, Hospital-coll und Root-coll

Entity-Db

Sorte einer Ansammlung (Menge) von Entities des Typs S.-Datenobjekt.

Datenobjekt-coll

Entitytypen.

Sunrise-Datenobjekt

Relationshiptypen. Ein Relationshiptyp wird spezifiziert als eine Menge (Monoliste) von Paaren über den Schlüsselsorten der beteiligten Entitytypen.

Aufenthaltsbezug Entstanden-aus Datenklassenbezug
Aktuelles-Stack-Objekt-von Default-Objekt

Attributsorten.

A-Name A-History A-DatenId

Domainsorten. Domainsorten bilden die Grundlage zur Spezifikation der Attributsorten. Durch Anreicherung einer Domainsorte um einen weiteren Träger, der den leeren Attributeintrag repräsentiert, wird die entsprechende Attributsorte spezifiziert.

Name PatId History DatenId Kennung

Sorte von Paaren über zwei Domainsorten. Diese Sorte ist die Schlüsselsorte des Entitytyps Aufenthalt.

PatId-x-AufenthaltsId

Sorten von Mengen über unterschiedl. Elementsorten (Set of ...).

SoPatId-x-AufenthaltsId S4oDatenId S3oDatenId S2oDatenId S1oDatenId
SoName

Anmerkung: Eigentlich sollen die Sorten S1oDatenId, ..., S4oDatenId — diese Entstehen im Zusammenhang mit der Anwendung des parametrisierten Standardmoduls MK_SPECIAL_BINARY_RELATIONSHIP (siehe auch Anmerkung auf Seite 129) — miteinander identifiziert werden. Alle vier Sorten beschreiben Mengen über der Elementsorte DatenId. Wegen des Sortenclashproblems (technisches Problem) ist die Zusammenfassung dieser Sorten zu einer einzigen jedoch nicht möglich. Das Sortenclashproblem wurde am Lehrstuhl von Prof. Dr.-Ing. Loeckx bereits ausführlichst diskutiert. Die erarbeitete Lösung — das Dummysharing — konnte aber hier nicht angewendet werden. Dies hätte nämlich ein wesentliches Umstrukturieren der gesamten Spezifikation erforderlich gemacht, so daß insbesondere die Anwendung des Transformationsschemas [Aut93] in der Strukturierung kaum wiederzuerkennen und damit die Gesamtspezifikation wesentlich unverständlicher geworden wäre. Insofern sind die vier unterschiedlichen Mengensorten S1oDatenId, ..., S4oDatenId, die eine identische Semantik haben sollen, das kleinere Übel.

OPNS

Prädikat zur Überprüfung der schematischen Integritätsbedingungen. Das Argument, eine Datenbank, wird auf die Einhaltung der schematischen Integritätsbedingungen überprüft.

OK-schematisch : Db → bool

Zugriff auf die Entitykomponente einer Datenbank. Aus der Argumentdatenbank wird die Entitykomponente extrahiert.

get-Entity-Db : Db → Entity-Db

Zugriff auf Relationships einer Datenbank. Jede dieser Operationen extrahiert aus der Argumentdatenbank eine Relationship eines bestimmten Typs.

get-Aufenthaltsbezug : Db → Aufenthaltsbezug
get-Entstanden-aus : Db → Entstanden-aus
get-Datenklassenbezug : Db → Datenklassenbezug
get-Aktuelles-Stack-Objekt-von : Db → Aktuelles-Stack-Objekt-von
get-Default-Objekt : Db → Default-Objekt

Zugriff auf die Ansammlung von S.-Datenobjekten in einer Datenbank. Angewendet auf eine Entitykomponente wird die darin enthaltene Ansammlung von S.-Datenobjekten bestimmt.

get-Datenobjekt-coll : Entity-Db → Datenobjekt-coll

Zugriff auf ein S.-Datenobjekt in einer Datenbank. In der Argumentdatenbank wird diejenige Entity bestimmt, die zu dem übergebenen Schlüssel korrespondiert.

get-Sunrise-Datenobjekt : DatenId Db → Sunrise-Datenobjekt

Anfrage, ob ein Relationshipeintrag in einer Datenbank vorliegt. Jede dieser Operationen überprüft, ob in einer speziellen Relationship ein Eintrag existiert, der zu dem (den) übergebenen Schlüssel(n) paßt.

is-in-snd-Default-Objekt : DatenId Db → bool
is-in-fst-Default-Objekt : DatenId Db → bool
is-in-Patientenbezug : PatId-x-AufenthaltsId PatId Db → bool
is-in-Aufenthaltsbezug : DatenId PatId-x-AufenthaltsId Db → bool
is-in-Stack-Objekt-von : DatenId Db → bool
is-in-Stack-Objekt-von : Kennung Db → bool
is-in-Lokal-fuer : DatenId Db → bool
is-in-Lokal-fuer : DatenId Kennung Db → bool
is-in-Global-fuer : DatenId Db → bool
is-in-Global-fuer : DatenId Kennung Db → bool

Setzen eines Relationshipeintrags in einer Datenbank. In der Argumentdatenbank wird durch diese Operationen ein spezieller Relationshipeintrag gemäß den übergebenen Schlüsseln vermerkt.

est-Default-Objekt : Db DatenId DatenId → Db

est-Entstanden-aus : Db DatenId DatenId → Db
est-Lokal-fuer : Db DatenId Kennung → Db
est-Global-fuer : Db DatenId Kennung → Db
est-Stack-Objekt-von : Db DatenId Kennung → Db
est-Aktuelles-Stack-Objekt-von : Db DatenId Kennung → Db

Entfernen eines Relationshipeintrags in einer Datenbank. In der Argumentdatenbank wird durch diese Operationen ein spezieller Relationshipeintrag gemäß den übergeben Schlüsseln gelöscht.

rel-Entstanden-aus : Db DatenId DatenId → Db
rel-Aufenthaltsbezug : Db DatenId PatId-x-AufenthaltsId → Db
rel-Datenklassenbezug : Db DatenId Name → Db
rel-Default-Objekt : Db DatenId DatenId → Db
rel-Stack-Objekt-von : Db DatenId Kennung → Db
rel-Aktuelles-Stack-Objekt-von : Db DatenId Kennung → Db
rel-Lokal-fuer : Db DatenId Kennung → Db

Entfernen eines S.-Datenobjekts aus einer Datenbank. Aus der Argumentdatenbank wird das übergebene S.-Datenobjekt entfernt. (Es ist an dieser Stelle nicht klar, warum das Transformationsschema [Aut93] bzw. [Het93] keinen Schlüssel als Argument vorsieht.)

del-Sunrise-Datenobjekt : Sunrise-Datenobjekt Db → Db

Überschreiben eines S.-Datenobjekts in einer Datenbank. In der Argumentdatenbank wird das S.-Datenobjekt durch das Argumentdatenobjekt überschrieben, das den selben Schlüssel wie das Argumentdatenobjekt aufweist.

update-Sunrise-Datenobjekt : Sunrise-Datenobjekt Db → Db

Anfrage, ob ein S.-Datenobjekt in einer Datenbank enthalten ist. Die Argumentdatenbank wird danach untersucht, ob sie ein S.-Datenobjek entält, das mit dem Argumentobjekt übereinstimmt. (Auch hier hätte die Anfrage nur über einen Schlüssel erfolgen können.)

is-in-Db : Sunrise-Datenobjekt Db → bool

Berechnung der Schlüsselattributwerte der Entities, die zu einem S.-Datenobjekt in einer bestimmten Relationship stehen.

reachable_from : DatenId Aufenthaltsbezug → SoPatId-x-AufenthaltsId
reachable_from : DatenId Entstanden-aus → S4oDatenId
reachable_from : DatenId Datenklassenbezug → SoName

Berechnung der Schlüsselattributwerte der Entities, die auf ein S.-Datenobjekt durch eine bestimmte Relationship abgebildet werden (Umkehrung der 'reachable-from'-Operationen).

reachable : DatenId Entstanden-aus → S3oDatenId

reachable : Kennung Aktuelles-Stack-Objekt-von → S1oDatenId
reachable : DatenId Default-Objekt → S2oDatenId

Zugriff ein (das erste) Element einer Menge (Monoliste). Für die unterschiedlichen Mengensorten (Monolistensorten) S1oDatenId bis S4oDatenId muß je eine Operation zur Verfügung gestellt werden. Weil in dieser Spezifikation Monolisten (siehe auch Abschnitt A.6.3) anstelle von Mengen verwendet werden, kann der Zugriff mit je einer Operation head erfolgen, die das erste Element einer gegebenen Monoliste bestimmt (Hinweis: die Elemente einer Menge sind a priori nicht geordnet).

head : S4oDatenId → DatenId
head : S3oDatenId → DatenId
head : S2oDatenId → DatenId
head : S1oDatenId → DatenId
head : Datenobjekt-coll → Sunrise-Datenobjekt
head : SoName → Name
head : SoPatId-x-AufenthaltsId → PatId-x-AufenthaltsId

Restlistenbestimmung. Das erste Element der Liste wird abgeschnitten und die Restliste zurückgeliefert.

tail : S4oDatenId → S4oDatenId
tail : S3oDatenId → S3oDatenId
tail : Datenobjekt-coll → Datenobjekt-coll

Leere Listen.

e-Datenobjekt-coll : → Datenobjekt-coll
e-S3oDatenId : → S3oDatenId

Zugriff auf Attributwerte eines S.-Datenobjekts. Jede dieser Operationen extrahiert den Attributwert eines bestimmten Attributs aus dem Argumentdatenobjekt.

History : Sunrise-Datenobjekt → A-History
DatenId : Sunrise-Datenobjekt → A-DatenId
Datenname : Sunrise-Datenobjekt → A-Name

Setzen von Attributwerten bei einem S.-Datenobjekt. Diese Operation überschreibt den Attributwert zum Attributbezeichner Datenname im Argumentdatenobjekt durch den Argumentnamen.

set-Datenname : A-Name Sunrise-Datenobjekt → Sunrise-Datenobjekt

Liftingkonstruktor zur Domainsorte Name. Dieser Konstruktor erhebt einen Träger der Domainsorte Name zu einem Träger der Attributsorte A-Name.

$\tilde{\ }$ _ : Name → A-Name

Selektoren zu den Liftingkonstruktoren (Umkehrung von $\tilde{\ }$).

$\tilde{\ }\tilde{\ }$ _ : A-DatenId → DatenId

~~ _ : A-History → History

Gleichheitsprädikate über unterschiedlichen Sorten.

_ = _ : Db Db → bool
_ = _ : Datenobjekt-coll Datenobjekt-coll → bool
_ = _ : A-Name A-Name → bool
_ = _ : S3oDatenId S3oDatenId → bool
_ = _ : DatenId DatenId → bool

Ende Modul SCHNITTSTELLE

3.4 Spezifikation der Akivitätenebene

Es wird nun die Spezifikation der Aktivitätenebene vorgestellt. Deren Struktur wird durch die Abbildung 3.2 beschrieben. Der Modul SCHNITTSTELLE wurde bereits erläutert. Er besteht lediglich aus einer Importliste, die sämtliche Importsignaturen der übrigen Module zusammenfaßt, d. h. er beschreibt genau die Schnittstelle zur Datenmodellebene. In SONDERMODUL werden einige Hilfssorten spezifi-

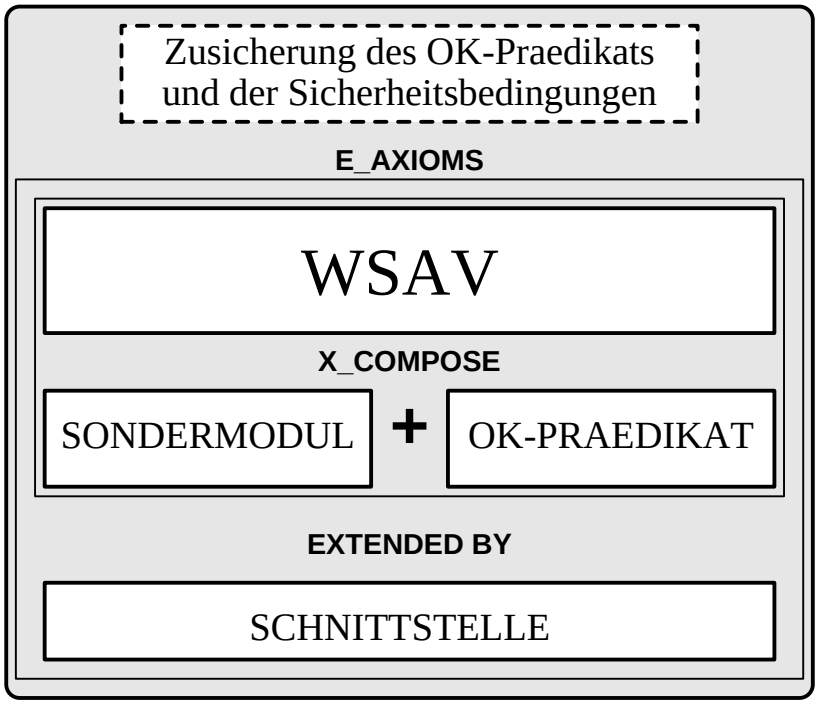

Abbildung 3.2: Struktur der Spezifikation zur Aktivitätenebene

ziert und in OK-PRAEDIKAT werden die schematischen Integritätsbedingungen, die bereits auf Datenmodellebene spezifiziert wurden, ergänzt um die spezifischen

Integritätsbedingungen. Das in OK-PRAEDIKAT spezifizierte Prädikat *OK* faßt die schematischen und spezifischen Integritätsbedingungen zusammen.

Im Modul WSAV wird die Aktivität WSAV modelliert. Dabei dienen die Datenflußdiagramme der Abbildungen 2.17 bis 2.19 aus Kapitel 2 als Grundlage. Sie geben die zu spezifizierenden Operationen und deren Stelligkeiten vor. Desweiteren beschreiben sie die Aufgliederung von WSAV in Unteraktivitäten und deren Kommunikation. Die rekursiven Algorithmen zu den einzelnen Unteraktivitäten — diese sind bisher nur informal beschrieben — müssen allerdings vom Spezifizierer hinzugefügt werden. Eine Operation WSAV-*fehler* ermittelt (logisch) fehlerhafte Argumentkonstellationen und produziert entsprechende Fehlermeldungen.

Anmerkung: Es ist leicht einzusehen, daß ähnlich zur Transformation von E/R-Diagrammen in OBSCURE-Spezifikationen auch Datenflußdiagramme transformiert werden können. Allerdings ist hierbei nur die Generierung von OBSCURE-Spezifikationsfragmenten möglich und nicht eine vollständige Spezifikation, wie bei den E/R-Diagrammen. Das Wesentliche, was diesen Fragmenten fehlen wird, sind die rekursiven Algorithmen, die nicht automatisch aus den Datenflußdiagrammen abgeleitet werden können. In [NW93] ist eine solche Transformation für Datenflußdiagramme beschrieben, allerdings für die Zielsprache SPECTRUM.

Zuletzt wird die Spezifikation der Aktivitätenebene ergänzt um die Zusicherungen, daß WSAV die Integritätsbedingungen berücksichtigt und die Sicherheitsaussage(n) erfüllt. Dazu werden Exportaxiome formuliert und der Spezifikation der Aktivitätenebene hinzugefügt.

In den folgenden Unterabschnitten werden die einzelnen Module vorgestellt. Die Konventionen zur Form, die besonders hier von Bedeutung sind, wurden in Abschnitt 1.5 vorgestellt. Es soll erneut darauf hingewiesen werden, daß in dieser Arbeit die Spezifikationsprache OBSCURE weder näher erläutert noch detailliert vorgestellt wird. Dazu dienen die Dokumente [FHM+91] und [LL93].

3.4.1 Spezifikation von Hilfssorten

Die Operation WSAV (siehe Abbildung 2.17) erhält als Argumente einen Träger einer Sorte *Db* (Datenbank), einen der Sorte *Kennung* (identifiziert den aktuellen Benutzer) und einen der Sorte *Name* (ein neuer Datenobjektname). Sie liefert eine Liste von Fehlermeldungen — im Normalfall die leere — und eine möglicherweise veränderte Datenbank zurück. Das heißt, sie bildet ab in das Kreuzprodukt der Trägermengen über den Sorten *Db* und *Fehlerliste*. Die Sorte *Fehlerliste* und die Kreuzproduktsorte *Db_x_Fehlerliste* werden nicht von der Datenmodellebene bereitgestellt, sondern im Modul SONDERMODUL spezifiziert. Auch die Zielsorte der Unteraktivität WSAV-1 — nämlich die Kreuzproduktsorte *DatenId_x_DatenId* — wird in diesem Modul eingeführt.
Zur Spezifikation der Kreuzproduktsorten kann man den parametrisierten OBSCURE-Standardmodul[2] TUPEL-2 einsetzen. Die Spezifikation der Sorte *Fehlerliste* erfolgt in zwei Schritten. In einem Modul FEHLERMELDUNG werden die möglichen Feh-

[2]Standardmodule werden in Abschnitt A.6 erläutert und vorgestellt.

lermeldungen der Operation WSAV spezifiziert als Konstanten einer Sorte Fehler. Auf die Sorte *Fehler* wird dann der Standardmodul LIST angewendet und so die Sorte *Fehlerliste* eingeführt.

Modul: SONDERMODUL
Autor: cb *Datum: 17. Januar 94*
Inhalt: Spezifikation der Sorten Db_x_Fehlerliste, Fehlerliste, Fehlermeldung *und* DatenId_x_DatenId, *sowie einiger Operationen.*

```
(
INCLUDE TUPEL-2 ( SORTS Db Fehlerliste )
                [ SORTS Db_x_Fehlerliste
                  OPNS {_/_}, fst, snd, set_fst, set_snd ]
X_COMPOSE
INCLUDE LIST ( SORTS Fehlermeldung)
             [ SORTS Fehlerliste
               OPNS e-fehlerliste, list ]

X_COMPOSE
INCLUDE FEHLERMELDUNG
)

PLUS

INCLUDE TUPEL-2 ( SORTS DatenId DatenId)
                [ SORTS DatenId_x_DatenId
                  OPNS {_/_}, fst, snd, set_fst, set_snd ]
```

Ende Modul SONDERMODUL

WSAV kann nicht auf allen Argumentkonstellationen fehlerfrei ablaufen. In den folgenden drei Fällen liegt ein (logischer) Eingabefehler vor: Im ersten Fall existiert zum aktuellen Benutzer kein offener Arbeitsstack, im zweiten ist das Default-Objekt des aktuellen Arbeitsstacks bereits ein globales Datenobjekt, es muß also nicht abgespeichert werden und im dritten ist der an WSAV übergebene Datenname (lokal) nicht eindeutig. Die drei entsprechenden Fehlermeldungen werden spezifiziert als Konstanten einer Sorte *Fehlermeldung*.

Modul: FEHLERMELDUNG
Autor: cb *Datum: 20. Januar 94*
Inhalt: Spezifikation von Fehlermeldungen als Konstanten der Sorte Fehlermeldung.

```
CREATE
 SORTS  Fehlermeldung
```

OPNS

kein-offener-stack-vorhanden	: →	Fehlermeldung
nichts-abzuspeichern	: →	Fehlermeldung
datenname-nicht-eindeutig	: →	Fehlermeldung
_ = _	: Fehlermeldung Fehlermeldung →	bool

SEMANTICS

CONSTRS

kein-offener-stack-vorhanden	: →	Fehlermeldung
nichts-abzuspeichern	: →	Fehlermeldung
datenname-nicht-eindeutig	: →	Fehlermeldung

ENDCREATE

Ende Modul FEHLERMELDUNG

3.4.2 Spezifikation der Integritätsbedingungen

Die schematischen Integritätsbedingungen werden durch die Spezifikation der Datenmodellebene bereitgestellt. Das Prädikat *OK-schematisch* (siehe Modul IB-SCHEMATISCH auf Seite 142) faßt diese zusammen. Angewendet auf eine Datenbank überprüft *OK-schematisch*, ob die übergebene Datenbank die schematischen Integritätsbedingungen erfüllt oder nicht. Dieses Prädikat wird hier importiert und ergänzt um ein Prädikat *OK-spezifisch*, das die beiden spezifischen Integritätsbedingungen beinhaltet (siehe auch Seite 38). Die spezifischen Integritätsbedingungen konnten im Diagramm 2.8 nicht graphisch dargestellt werden und sind deshalb auch nicht in der Spezifikation der Datenmodellebene enthalten. Konjunktiv verknüpft bilden *OK-schematisch* und *OK-spezifisch* das *OK-Prädikat*. Angewendet auf eine Datenbank bestimmt das *OK-Prädikat*, ob die schematischen und spezifischen Integritätsbedingungen von der Datenbank erfüllt werden.

Modul: OK-Prädikat
Autor: cb *Datum: 20. Januar 94*
Inhalt: Spezifikation der spezifischen Integritätsbedingungen und Verknüpfung der spezifischen und schematischen Integritätsbedingungen zum Prädikat OK.

IMPORTS

SORTS

Db Entity-Db Datenobjekt-coll Sunrise-Datenobjekt A-DatenId A-History DatenId History

OPNS

OK-schematisch	: Db	→ bool
get-Entity-Db	: Db	→ Entity-Db
get-Datenobjekt-coll	: Entity-Db	→ Datenobjekt-coll
e-Datenobjekt-coll	:	→ Datenobjekt-coll
head	: Datenobjekt-coll	→ Sunrise-Datenobjekt
tail	: Datenobjekt-coll	→ Datenobjekt-coll

DatenId	: Sunrise-Datenobjekt	→ A-DatenId
History	: Sunrise-Datenobjekt	→ A-History
~~ _	: A-DatenId	→ DatenId
~~ _	: A-History	→ History
is-in-Stack-Objekt-von	: DatenId Db	→ bool
is-in-fst-Default-Objekt	: DatenId Db	→ bool
is-in-snd-Default-Objekt	: DatenId Db	→ bool
is-in-Global-fuer	: DatenId Db	→ bool
is-in-Lokal-fuer	: DatenId Db	→ bool
_ = _	: Datenobjekt-coll Datenobjekt-coll	→ bool

CREATE

OPNS

OK	: Db	→ bool
OK-spezifisch	: Db	→ bool
OK-spez-1	: Datenobjekt-coll Db	→ bool
OK-spez-2	: Datenobjekt-coll Db	→ bool

SEMANTICS

VARS

db : Db
dc : Datenobjekt-coll
di : DatenId

PROGRAMS

Das OK-Prädikat setzt sich konjunktiv verknüpft aus den Prädikaten OK-schematisch und OK-spezifisch zusammen.

OK(db) ← OK-schematisch(db) and OK-spezifisch(db);

Das spezifische OK-Prädikat enthält die beiden spezifischen Integritätsbedingungen, die im E/R-Diagramm 2.8 nicht dargestellt wurden, weil die gewählte E/R-Modellierungstechnik dazu keine geeigneten Beschreibungsmittel bereitstellt. Diese beiden Integritätsbedingungen überprüfen je eine Eigenschaft für alle Entities des Typs S.-Datenobjekt, *die in einer Datenbank (Träger der Sorte Db) enthalten sind. Die Ansammlung von S.-Datenobjekten in der übergebenen Datenbank wird der Variablen dc zugewiesen.*

OK-spezifisch(db) ←
LET dc : get-Datenobjekt-coll(get-Entity-Db(db))
IN OK-spez-1(dc,db) and OK-spez-2(dc,db)
TEL ;

Die Operation ok-spez-1 *überprüft für eine Menge von S.-Datenobjekten, ob für diese in der aktuellen Datenbank folgende Integritätsbedingung gilt: Zu einem S.-Datenobjekt, das ein Stack-Objekt ist, gibt es ein Default-Objekt, und ein Default-Objekt steht immer in Beziehung zu einem Stack-Objekt. Dazu wird ein rekursiver Algorithmus angeben.*

```
OK-spez-1(dc,db) ←
    IF dc = e-Datenobjekt-coll THEN true
    ELSE
     LET di : ~~ DatenId(head(dc))
     IN   ( (is-in-Stack-Objekt-von(di,db) and
               is-in-snd-Default-Objekt(di,db))
               or
               ((not is-in-Stack-Objekt-von(di,db)) and
               (not is-in-snd-Default-Objekt(di,db)))
          )
          and OK-spez-1(tail(dc),db)
     TEL
    FI ;
```

Die Operation ok-spez-2 *überprüft für eine Menge von S.-Datenobjekten, ob für diese in der aktuellen Datenbank folgende Integritätsbedingung gilt: Jedes S.-Datenobjekt ist entweder lokales S.-Datenobjekt eines Benutzers oder globales, aber niemals ist es beides zugleich.*

```
OK-spez-2(dc,db) ←
    IF dc = e-Datenobjekt-coll THEN true
    ELSE
     LET di : ~~ DatenId(head(dc))
     IN   ( (is-in-Global-fuer(di,db) and (not is-in-Lokal-fuer(di,db)))
               or
               ((not is-in-Global-fuer(di,db)) and is-in-Lokal-fuer(di,db))
          )
          and OK-spez-2(tail(dc),db)
     TEL
    FI ;
ENDCREATE
```

Bis auf das OK-Prädikat können sämtliche Operationen und Sorten (mit Ausnahme von Db und bool) vergessen werden.

```
FORGET_ALL_BUT
OPNS OK **
```

Ende Modul OK-Prädikat

3.4.3 Spezifikation der essentiellen Aktivität WSAV

Die essentielle Aktivität WSAV zerfällt gemäß Abbildung 2.17 in die Unteraktivitäten WSAV-1 bis WSAV-5. WSAV-1 und WSAV-4 wurden weiter zerlegt in WSAV-1a und WSAV-1b, sowie WSAV-4a und WSAV-4b (siehe Abbildungen 2.18 und 2.19). Auch die Stelligkeiten und die Kommunikation der Unteraktiväten, werden durch

die Datenflußdiagramme festgelegt. Die folgende Spezifikation berücksichtigt diese Vorgaben. Die fehlerhaften Argumentkonstellationen werden durch eine Operation WSAV-*fehler* ermittelt.

Modul: WSAV
Autor: cb *Datum: 22. Februar 94*
Inhalt: Spezifikation der essentiellen Aktivität WSAV*, sowie ihrer Unteraktivitäten und einer Fehleranzeigeoperation.*

IMPORTS

SORTS

Db Fehlerliste Fehlermeldung Db_x_Fehlerliste Kennung
DatenId DatenId_x_DatenId A-Name Name Sunrise-Datenobjekt
PatId-x-AufenthaltsId S1oDatenId S2oDatenId S3oDatenId S4oDatenId
SoName SoPatId-x-AufenthaltsId
Aktuelles-Stack-Objekt-von Default-Objekt Entstanden-aus
Datenklassenbezug Aufenthaltsbezug

OPNS

Importiert aus SONDERMODUL:

e-fehlerliste	:	→ Fehlerliste
list	: Fehlermeldung	→ Fehlerliste
_ = _	: Fehlerliste Fehlerliste	→ bool
kein-offener-stack-vorhanden	:	→ Fehlermeldung
nichts-abzuspeichern	:	→ Fehlermeldung
datenname-nicht-eindeutig	:	→ Fehlermeldung
{_/_}	: Db Fehlerliste	→ Db_x_Fehlerliste
fst	: DatenId_x_DatenId	→ DatenId
snd	: DatenId_x_DatenId	→ DatenId
{_/_}	: DatenId DatenId	→ DatenId_x_DatenId

Importiert aus DATENMODELLEBENE:

reachable	: Kennung Aktuelles-Stack-Objekt-von	→ S1oDatenId
head	: S1oDatenId	→ DatenId
reachable	: DatenId Default-Objekt	→ S2oDatenId
head	: S2oDatenId	→ DatenId
reachable	: DatenId Entstanden-aus	→ S3oDatenId
head	: S3oDatenId	→ DatenId
tail	: S3oDatenId	→ S3oDatenId
e-S3oDatenId	:	→ S3oDatenId
_ = _	: S3oDatenId S3oDatenId	→ bool
reachable_from	: DatenId Entstanden-aus	→ S4oDatenId
head	: S4oDatenId	→ DatenId
tail	: S4oDatenId	→ S4oDatenId

reachable_from : DatenId Datenklassenbezug → SoName
head : SoName → Name
reachable_from : DatenId Aufenthaltsbezug → SoPatId-x-AufenthaltsId
head : SoPatId-x-AufenthaltsId → PatId-x-AufenthaltsId
Datenname : Sunrise-Datenobjekt → A-Name
set-Datenname : A-Name Sunrise-Datenobjekt → Sunrise-Datenobjekt
_ = _ : A-Name A-Name → bool
~ _ : Name → A-Name
is-in-Stack-Objekt-von : Kennung Db → bool
is-in-Global-fuer : DatenId Kennung Db → bool
is-in-Lokal-fuer : DatenId Kennung Db → bool
get-Aktuelles-Stack-Objekt-von : Db → Aktuelles-Stack-Objekt-von
get-Default-Objekt : Db → Default-Objekt
get-Entstanden-aus : Db → Entstanden-aus
get-Datenklassenbezug : Db → Datenklassenbezug
get-Aufenthaltsbezug : Db → Aufenthaltsbezug
est-Global-fuer : Db DatenId Kennung → Db
est-Lokal-fuer : Db DatenId Kennung → Db
est-Entstanden-aus : Db DatenId DatenId → Db
est-Default-Objekt : Db DatenId DatenId → Db
est-Aktuelles-Stack-Objekt-von : Db DatenId Kennung → Db
est-Stack-Objekt-von : Db DatenId Kennung → Db
rel-Lokal-fuer : Db DatenId Kennung → Db
rel-Aktuelles-Stack-Objekt-von : Db DatenId Kennung → Db
rel-Stack-Objekt-von : Db DatenId Kennung → Db
rel-Entstanden-aus : Db DatenId DatenId → Db
rel-Datenklassenbezug : Db DatenId Name → Db
rel-Aufenthaltsbezug : Db DatenId PatId-x-AufenthaltsId → Db
rel-Default-Objekt : Db DatenId DatenId → Db
get-Sunrise-Datenobjekt : DatenId Db → Sunrise-Datenobjekt
update-Sunrise-Datenobjekt : Sunrise-Datenobjekt Db → Db
del-Sunrise-Datenobjekt : Sunrise-Datenobjekt Db → Db

CREATE
OPNS

wsav : Db Kennung Name → Db_x_Fehlerliste

Die Unteraktivitäten:

wsav-1 : Db Kennung → DatenId_x_DatenId
wsav-1a : Db Kennung → DatenId
wsav-1b : Db DatenId → DatenId
wsav-2 : Db Kennung DatenId Name → Db
wsav-3 : Db Kennung DatenId DatenId → Db
wsav-4 : Db Kennung DatenId DatenId → Db

wsav-4b	: Db DatenId DatenId	→ Db
wsav-4a	: Db Kennung DatenId	→ Db
wsav-5	: Db Kennung DatenId	→ Db

Anzeige fehlerhafter Argumentkonstellationen:

wsav-fehler	: Db Kennung Name	→ Fehlerliste

Hilfsoperationen zu wsav-fehler*:*

wsav-1-fehler	: Db Kennung	→ bool
wsav-2-fehler	: Db Kennung	→ bool
wsav-genereller-fehler	: Db Kennung Name	→ bool

Zusätzliche Hilfsoperationen:

datenklasse-zu	: Db DatenId	→ Name
aufenthalt-zu	: Db DatenId	→ PatId-x-AufenthaltsId
vorgaenger-von	: Db DatenId	→ DatenId
name-eindeutig	: Db Name S3oDatenId	→ bool

SEMANTICS

VARS

db, db1, db1a, db1b, db1c, db1d, db2, db3, db4 : Db
di1,di2 : DatenId
fehlerliste : Fehlerliste
k : Kennung
n : Name
x : DatenId_x_DatenId
y : S3oDatenId
sd : Sunrise-Datenobjekt

PROGRAMS

Die Spezifikation der Operation WSAV *erfolgt nach den Vorgaben von Abbildung 2.17. Zuerst überprüft* WSAV*, ob sie auf ihren Argumenten fehlerfrei ablaufen kann. Dazu bedient sie sich der Operation* WSAV*-fehler, die eine Fehlerliste zurückliefert. Falls dies die leere Fehlerliste ist (*e-fehlerliste*), wendet* WSAV *ihre Unteraktivitäten* WSAV*-1 bis* WSAV*-5 gemäß Abbildung 2.17 an. Zurückgeliefert wird von* WSAV *dann ein Paar, bestehend aus einer veränderten Datenbank und der leeren Fehlerliste. Falls* WSAV*-fehler jedoch logische Eingabefehler anzeigt, liefert* WSAV *die unveränderte Datenbank und eine Fehlerliste zurück, die Hinweise auf die Art der logischen Fehler enthält.*

Anmerkung: Leider kann eine parallele Ausführung von Unteraktivitäten nicht ohne weiteres spezifiziert werden. Deshalb muß für die Unteraktivitäten WSAV*-2,* WSAV*-3 und* WSAV*-4, für die im DFD-Diagramm der Abbildung 2.17 keine Ausführungsreihenfolge festgelegt wurde, hier eine solche explizit angegeben werden.*

```
wsav(db,k,n) ←
 LET fehlerliste : wsav-fehler(db,k,n) IN
  IF fehlerliste = e-fehlerliste
  THEN
   LET x : wsav-1(db,k) IN
    LET di1 : fst(x), di2 : snd(x) IN
     LET db1 : wsav-2(db,k,di2,n) IN
     LET db2 : wsav-3(db1,k,di1,di2) IN
     LET db3 : wsav-4(db2,k,di1,di2) IN
     LET db4 : wsav-5(db3,k,di2) IN
      {db4/e-fehlerliste}
     TEL
     TEL
     TEL
     TEL
    TEL
   TEL
  ELSE {db/fehlerliste}
  FI
 TEL ;
```

WSAV-*1 zerfällt, wie in Abbildung 2.18 dargestellt, in zwei weitere Unteraktivitäten* WSAV-*1a und* WSAV-*1b.* WSAV-*1a bestimmt die DatenId des S.-Datenobjekts, das den aktuellen Arbeitsstack des, durch die Kennung k identifizierten, Benutzers definiert. Diese DatenId wird weitergeleitet an* WSAV-*1b. Dort wird dann das Default-Objekt dieses Arbeitsstacks ermittelt. Zurückgeliefert wird von* WSAV *ein Paar, das die Ergebnisse beider Unteraktivitäten zusammenfaßt.*

```
wsav-1(db,k) ←
 LET di1 : wsav-1a(db,k) IN
  LET di2 : wsav-1b(db,di1) IN
   {di1/di2}
  TEL
 TEL ;
```

Um die DatenId desjenigen S.-Datenobjekts zu ermitteln, das den aktuellen Arbeitstack des, durch k identifizierten, Benutzers definiert, erfolgt in WSAV-*1a ein lesender Zugriff auf die Relationship Aktuelles-Stack-Objekt-von. Die Operation get-Aktuelles-Stack-Objekt extrahiert die Relationship Aktuelles-Stack-Objekt aus einer Datenbank. Danach bestimmt die Operation reachable die Menge (Monoliste) der Schlüssel aller S.-Datenobjekte, die in dieser Relationship zu dem Benutzer stehen, der, durch die Kennung k identifiziert wird. Wegen der schematischen Integritätsbedingungen enthält diese Menge genau einen Schlüssel, auf den mit der Operation head zugegriffen wird.*

```
wsav-1a(db,k) ←
```

head(reachable (k,get-Aktuelles-Stack-Objekt-von(db)));

WSAV-*1b ermittelt in einem lesenden Zugriff auf die Relationship Default-Objekt das Default-Objekt des Arbeitstacks, der durch das Stack-Objekt zu di1 definiert wird. Der Zugriff erfolgt analog zu* WSAV-*1a.*

wsav-1b(db,di1) ←
head(reachable (di1,get-Default-Objekt(db)));

WSAV-*2 speichert das, über die DatenId di1 referenzierte, S.-Datenobjekt global ab (est-Global-fuer) und löscht den lokalen Speicherbezug (rel-Lokal-fuer). Vor dem globalen Abspeichern wird dem S.-Datenobjekt der Datenname n übergeben (set-Datenname und update-Sunrise-Datenobjekt).*

```
wsav-2(db,k,di1,n) ←
 LET sd : set-Datenname(˜n,get-Sunrise-Datenobjekt(di1,db)) IN
  LET db : update-Sunrise-Datenobjekt(sd,db) IN
   est-Global-fuer(rel-Lokal-fuer(db,di1,k),di1,k)
  TEL
 TEL ;
```

WSAV-*3 löscht einerseits das, durch di1 referenzierte, S.-Datenobjekt aus den Relationships Stack-Objekt-von und Aktuelles-Stack-Objekt-von, andererseits das durch di2 referenzierte S.-Datenobjekt aus der Relationship Default-Objekt.*

```
wsav-3(db,k,di1,di2) ←
 LET db1 : rel-Stack-Objekt-von(db,di1,k) IN
  LET db2 : rel-Aktuelles-Stack-Objekt-von(db1,di1,k) IN
   rel-Default-Objekt(db2,di2,di1)
  TEL
 TEL ;
```

WSAV-*4 zerfällt gemäß Abbildung 2.17 in die Unteraktivitäten* WSAV-*4a und* WSAV-*4b.* WSAV-*4b wird auf die resultierende Datenbank von* WSAV-*4a angewendet.*

wsav-4(db,k,di1,di2) ← wsav-4b(wsav-4a(db,k,di2),di1,di2);

WSAV-*4a löscht rekursiv alle lokalen S.-Datenobjekte, die Vorgänger des, durch di2 referenzierten, S.-Datenobjekts sind. Das Löschen eines S.-Datenobjektes muß natürlich einhergehen mit dem Löschen aller Einträge in Relationships, an denen dieses S.-Datenobjekt beteiligt ist. Dies sind in diesem Fall die Relationships Entstanden-aus, Lokal-fuer, Datenklassenbezug und Aufenthaltsbezug. Die Hilfsoperation vorgaenger-von bestimmt zunächst das Vorgängerobjekt (in der Relationship Entstanden-aus) eines S.-Datenobjekts. Dann wird überprüft, ob dieses auch ein lokales S.-Datenobjekt ist, also, ob das unterste Element des Stacks (das Stack-Objekt) noch nicht erreicht wurde.*

Falls dies der Fall ist, wird das lokale S.-Datenobjekt aus den vier Relationships Entstanden-aus, Lokal-fuer, Datenklassenbezug und Aufenthaltsbezug eliminiert. (Beachte: In dem Relationship Entstanden-aus wird nur die Information gelöscht, daß das S.-Datenobjekt zu di2 aus dem zu di1 entstanden ist. Die Information zu dem Vorgänger des S.-Datenobjekts zu di1 — die für den rekursiven Aufruf benötigt wird — ist zu diesem Zeitpunkt noch enthalten.) Danach erfolgt der rekursive Aufruf von WSAV-*4a, und erst ganz zum Schluß werden die aufgesammelten lokalen S.-Datenobjekte aus der Datenbank entfernt.*

```
wsav-4a(db,k,di2) ←
 LET di1 : vorgaenger-von(db,di2) IN
  IF is-in-Lokal-fuer(di1,k,db)
  THEN
   LET db1a : rel-Entstanden-aus(db,di2,di1) IN
   LET db1b : rel-Lokal-fuer(db1a,di1,k) IN
   LET db1c : rel-Datenklassenbezug(db1b,di1,datenklasse-zu(db1b,di1)) IN
   LET db1d : rel-Aufenthaltsbezug(db1c,di1,aufenthalt-zu(db1c,di1)) IN
    LET db2 : wsav-4a(db1d,k,di1) IN
     del-Sunrise-Datenobjekt(get-Sunrise-Datenobjekt(di1,db2),db2)
    TEL
   TEL
   TEL
   TEL
   TEL
  ELSE rel-Entstanden-aus(db,di2,di1)
  FI
TEL;
```

WSAV-*4b setzt einen Relationshipeintrag zum Typ Entstanden-aus zwischen zwei S.-Datenobjekten, die durch ihre DatenIds identifiziert werden.*

```
wsav-4b(db,di1,di2) ← est-Entstanden-aus(db,di2,di1);
```

WSAV-*5 setzt Relationshipeinträge zu den Typen Stack-Objekt-von und Aktuelles-Stack-Objekt-von zwischen dem, durch di1 referenzierten, S.-Datenobjekt und dem, durch k referenzierten, Benutzer und setzt das S.-Datenobjekt zu di1 in Default-Objekt-Beziehung zu sich selbst.*

```
wsav-5(db,k,di1) ←
 LET db1 : est-Stack-Objekt-von(db,di1,k) IN
  LET db2 : est-Aktuelles-Stack-Objekt-von(db1,di1,k) IN
   est-Default-Objekt(db2,di1,di1)
  TEL
 TEL ;
```

WSAV-*fehler überprüft für eine beliebige Argumentkonstellation der Aktivität* WSAV*, ob diese ein fehlerfreies Ablaufen von* WSAV *garantiert. Drei Fehlermöglichkeiten sind hier zu unterscheiden. Die ersten beiden stehen in Zusammenhang zu der Aktivität* WSAV *selbst. Die Unteraktivitäten* WSAV-*1 und* WSAV-*2 können auf verschiedenen Argumentkonstellationen nicht fehlerfrei ablaufen. Diese Argumentkonstellationen werden durch die Operationen* WSAV-*1-fehler und* WSAV-*2-fehler angezeigt. Eine weitere Fehlermöglichkeit steht in Zusammenhang mit einer anderen Systemaktivität (Auswahl eines S.-Datenobjekts aus dem globalen Archiv nach dessen Datennamen zur Weiterbearbeitung), die ebenfalls nur dann fehlerfrei auf der Datenbank arbeiten kann, falls* WSAV *verschiedene Argumentkonstallation von der Bearbeitung ausschließt.*

```
wsav-fehler(db,k,n) ←
 IF wsav-1-fehler(db,k)
 THEN list(kein-offener-stack-vorhanden)
 ELSE
  IF wsav-2-fehler(db,k)
  THEN list(nichts-abzuspeichern)
  ELSE
   IF wsav-genereller-fehler(db,k,n)
   THEN list(datenname-nicht-eindeutig)
   ELSE e-fehlerliste
   FI
  FI
 FI ;
```

Die erste Fehlermöglichkeit besteht darin, daß zu dem Benutzer, der durch die, an WSAV *und damit auch an* WSAV-*1 übergebene, Kennung k identifiziert wird, kein aktueller Arbeitstack existiert. Praktisch bedeutet das: Der Benutzer mit der Kennung* k *hat keine Arbeitssitzung offen; es gibt also keinen Arbeitsstack und deshalb auch kein aktuelles Stack-Objekt zum Benutzer.* WSAV-*1 kann in diesem Fall nicht fehlerfrei ablaufen.*

```
wsav-1-fehler(db,k) ← not (is-in-Stack-Objekt-von(k,db));
```

Eine weitere Fehlermöglichkeit besteht darin, daß zu dem durch k *bestimmten Benutzer zwar ein aktueller Arbeitstack existiert (das S.-Datenobjekt, das diesen definiert, wird durch* WSAV-*1a ermittelt), dessen Default-Objekt (wird durch* WSAV-*1b ermittelt) aber bereits ein globales Datenobjekt ist, d. h. es gibt nichts abzuspeichern. In diesem Fall würde die Unteraktivität* WSAV-*2 nicht fehlerfrei ablaufen können.*

```
wsav-2-fehler(db,k) ←
 LET di1 : wsav-1a(db,k) IN
  LET di2 : wsav-1b(db,di1) IN
   is-in-Global-fuer(di2,k,db)
```

```
   TEL
  TEL ;
```

Wie bereits angesprochen, ermittelt das dritte Fehlerprädikat solche Argumentkonstellationen, die zu einem Fehler für andere Systemaktivitäten führen können, die aber bereits hier abgeprüft werden müssen. Konkret betroffen ist eine Aktivität zur Auswahl von S.-Datenobjekten in der Datenbank nach deren Datennamen. Diese setzt die Vergabe eindeutiger Namen (zumindest für Teilbereiche in der Datenbank) voraus. Datennamen werden aber durch WSAV *vergeben, also muß auch in* WSAV *darauf geachtet werden, daß (für den jeweiligen Teilbereich) eindeutige Namen vergeben werden. Der angesprochene Teilbereich umfaßt alle Geschwister eines S.-Datenobjekts, also diejenigen Objekte, die aus dem gleichen direkten Vorfahrobjekt hervorgehen. Der Name* n *wird durch die Operation name-eindeutig auf seine Eindeutigkeit bezüglich der Liste der DatenIds y von Geschwistern des S.-Datenobjekts zu di1 überprüft.*

```
wsav-genereller-fehler(db,k,n) ←
 LET di1 : wsav-1a(db,k) IN
  LET di2 : wsav-1b(db,di1) IN
   LET y : reachable(vorgaenger-von(db,di2),get-Entstanden-aus(db)) IN
    not (name-eindeutig(db,n,y))
   TEL
  TEL
 TEL ;
```

Die Hilfsoperation datenklasse-zu bestimmt den Klassennamen der Datenklasse des, durch di1 *referenzierten, S.-Datenobjekts.*

```
datenklasse-zu(db,di1) ← head(reachable_from(di1,get-Datenklassenbezug(db)));
```

Die Hilfsoperation aufenthalt-zu bestimmt den Schlüssel des Aufenthalts, dem das, durch di1 referenzierte, S.-Datenobjekt zugeordnet ist.

```
aufenthalt-zu(db,di1) ← head(reachable_from(di1,get-Aufenthaltsbezug(db)));
```

Die Hilfsoperation vorgaenger-von bestimmt die DatenId des S.-Datenobjekts, aus dem das, durch di1 referenzierte, entstanden ist.

```
vorgaenger-von(db,di1) ← head(reachable_from(di1,get-Entstanden-aus(db)));
```

Die Hilfsoperation name-eindeutig überprüft, ob ein Datenname n für eines der S.-Datenobjekte, die durch die Liste von DatenIds y bestimmt werden, schon vergeben wurde.

```
name-eindeutig(db,n,y) ←
 IF y = e-S3oDatenId
 THEN true
 ELSE
   (not (Datenname(get-Sunrise-Datenobjekt(head(y),db)) = ˜ n ))
   and
```

name-eindeutig(db,n,tail(y))
FI ;

ENDCREATE

Ende Modul WSAV

3.4.4 Zusammensetzen der Aktivitätenebene

Die einzelnen Module der Aktivitätenebene können, wie in Abbildung 3.2 beschrieben, zusammengefügt werden. Dies erfolgt im Modul AKTIVITAETENEBENE. Durch Exportaxiome wird ferner die Einhaltung der Integritätsbedingungen und die Gültigkeit der Sicherheitsaussage für WSAV gefordert. Die informal beschriebene Sicherheitsaussage kann recht einfach in Spezifikationstext übersetzt werden. Auch die Forderung nach der Einhaltung der Integritätsbedingungen kann leicht und verständlich formalisiert werden. Durch eine formale Verifikation kann prinzipiell der Nachweis erbracht werden, daß diese Exportaxiome in dem spezifizierten Modell tatsächlich gelten, daß also WSAV die formulierten Eigenschaften auch wirklich besitzt.

Modul: AKTIVITAETENEBENE
Autor: cb *Datum: 24. Februar 94*
Inhalt: Zusammensetzen der Aktivitätenebene und Zusicherung, daß WSAV *die Integritätsbedingungen und die Sicherheitsaussagen berücksichtigt. Das Zusammensetzen der einzelnen Module der Aktivitätenebene erfolgt gemäß der Abbildung 3.2.*

INCLUDE SCHNITTSTELLE
EXTENDED BY
INCLUDE WSAV ## **PLUS INCLUDE** 'andere Aktivitäten'
X_COMPOSE
(INCLUDE SONDERMODUL **PLUS INCLUDE** OK-PRAEDIKAT)
ENDEXTENDED

In Exportaxiomen wird zum einen die Berücksichtigung der Integritätsbedigungen für WSAV *gefordert und zum anderen die Gültigkeit der beiden Sicherheitsaussagen, die in Abschnitt 2.4 erläutert wurden. Analog kann die Einhaltung der Integritätsbedingungen und der Sicherheitsaussagen auch für andere, hier nicht betrachtete, essentielle Systemaktivitäten gefordert werden.*

E_AXIOMS
VARS db : Db
k : Kennung
n : Name
di : DatenId
ai : PatId-x-AufenthaltsId
pi : PatId;

Integritätsbedingungen: Falls WSAV *auf eine Datenbank angewendet wird, die die Integritätsbedingungen erfüllt, so werden diese auch von der resultierenden Datenbank erfüllt. Wenn alle anderen Systemaktivitäten dies ebenfalls gewährleisten und die initiale Datenbank des* SUNRISE*-Systems die Integritätsbedingungen einhält, so wird hiermit sichergestellt, daß niemals eine Datenbank mit dem System erzeugt werden kann, welche die Integritätsbedingungen nicht erfüllt.*

```
OK(db) == true  =>  OK(fst(wsav(db,k,n))) == true;
```

1. Sicherheitsaussage: Die erste Sicherheitsaussage fordert, daß die Aktivität WSAV *niemals den Aufenthaltsbezug eines nicht zu löschenden S.-Datenobjekts verändert. Sie ist allquantifiziert über den Sorten DatenId (identifiziert S.-Datenobjekt-Entities), PatId_x_AufenthaltsId (identifiziert Aufenthalts-Entities), Db, Kennung und Name.*

```
(is-in-Aufenthaltsbezug(di,ai,db) == true
  => is-in-Aufenthaltsbezug(di,ai,fst(wsav(db,k,n))) == true) |
is-in-Db(get-Sunrise-Datenobjekt(di,db),fst(wsav(db,k,n))) == false;
```

2. Sicherheitsaussage: Die zweite Sicherheitsaussage fordert, daß durch WSAV *niemals ein Patientenbezug eines Aufenthalts verändert werden kann. Diese Sicherheitsaussage ist allquantifiziert über den Sorten PatId_x_AufenthaltsId, PatId (identifiziert Patienten-Entities), Db, Kennung und Name. Die erste und zweite Sicherheitsbedingung stellen gemeinsam sicher, daß die indirekte Zuordnung eines nicht zu löschenden S.-Datenobjekts zu Patienten-Entities durch* WSAV *nicht verändert werden kann.*

```
is-in-Patientenbezug(ai,pi,db) == true
  => is-in-Patientenbezug(ai,pi,fst(wsav(db,k,n))) == true;
```

Anmerkung: In den Axiomen der beiden Sicherheitsaussagen wurde deshalb eine (schwächere) Implikationsbeziehung (=>) gewählt, weil eine mögliche weitere Sicherheitsaussage explizit formulieren sollte, daß die Aktivität WSAV *keine neuen Entities einführt, bzw. anlegt. Gememeinsam mit den Integritätsbedingungen wird dann die Wahl eine Aquivalenzbeziehung (<=>) anstelle der Implikationsbeziehung (=>) überflüssig. Prinzipiell hätte man aber hier auch direkt eine Aquivalenzbeziehung (<=>) formulieren können.*

ENDAXIOMS

Am Ende können alle Operationen, bis auf WSAV *"vergessen" werden. Nur die essentielle Aktivität* WSAV *selbst, würde zur Spezifikation des Gesamtsystems weiter benötigt. Zur Durchführung eines Rapid-Prototypings muß hier auf das "Vergessen" verzichtet werden: Weil keine anderen Systemaktivitäten spezifiziert wurden, ist es sonst nicht möglich eine Ausgangsdatenbank aufzubauen, auf die* WSAV *angewendet werden kann. Anmerkung: '##' ist das Kommentarzeichen für* OBSCURE*.*

FORGET_ALL_BUT

OPNS
wsav **

Ende Modul AKTIVITAETENEBENE

3.5 Rapid-Prototyping

Die vorgestellte formale Anforderungsspezifikation in der algorithmischen Spezifikationssprache OBSCURE kann man auch als eine abstrakte Beschreibung eines System-Prototypen zum Ausschnitt *duales Speicherkonzept* auffassen. Ist der Import der Spezifikation leer, d. h. alle in der Spezifikation auftretenden Datentypen sind vollständig spezifiziert, so kann mit dem Interpreter (siehe [Sto91]) des OBSCURE-Systems ein Rapid-Prototyping durchgeführt werden. Die beschriebenen essentiellen Aktivitäten (hier leider nur WSAV) können dabei auf abstrakter Ebene durchgespielt werden, um dadurch die Anforderungspezifikation hinsichtlich ihrer Eignung, die Systemziele zu realisieren (Validierung), zu untersuchen. Besonders nützlich kann ein Rapid-Prototyping sein, wenn es gemeinsam mit den Auftraggebern des zu erstellenden Systems erfolgt. Möglicherweise können dadurch bereits in einer frühen Phase der Systementwicklung Mißverständnisse und Unstimmigkeiten bezüglich der Systemanforderungen aufgedeckt und beseitigt werden. Gerade die Behebung von Fehlern oder Mißverständnissen hinsichtlich der Systemanforderungen ist meist dann mit großem Aufwand verbunden, wenn diese erst spät erkannt werden.

Ein kommentierter Mitschnitt der Interpretersitzung mit dem Interpreter des OBSCURE-Systems findet sich im Anhang B. Dort wird auch versucht, die abstrakten und schwer verständlichen[3] Terme der Interpretersitzung zu erläutern. An dieser Stelle soll das Ergebnis eines Aufrufs der Operation WSAV mit graphischen Mitteln beschrieben werden. Diese Beschreibung bezieht sich auf den, auf den Seiten 175 bis 179 vorgestellten, Ausschnitt der Interpretersitzung.

3.5.1 Erreichbare Datenbankzustände

In einer kompletten Anforderungsspezifikation des SUNRISE-Systems wären alle essentielle Aktivitäten formalisiert worden. Es würden dann z. B. Aktivitäten zur Initialisierung der Datenbank, zum Anlegen von Aufenthalten, Patienten, Benutzern, zum Import von S.-Datenobjekten und zur Bearbeitung von S.-Datenobjekten existieren. Mit diesen Systemaktivitäten könnte man, ausgehend vom Initialzustand der Datenbank, mögliche Datenbankzustände aufbauen. Diejenigen Datenbankzustände, die durch auschließliche Anwendung essentieller Aktivitäten aus dem Initi-

[3] Die Terme der Interpretersitzung sind deshalb so wenig selbsterklärend, weil die *Domainsorten* der *Attributsorten* nicht ausspezifiziert wurden, sondern aus einer Umbenennung der ganzen Zahlen hervorgehen. Siehe hierzu auch Abschnitt A.2.1

alzustand ableitbar sind, werden im folgenden **erreichbare Datenbankzustände** genannt.

3.5.2 Allgemeine Datenbankzustände

Bei der Transformation von E/R-Diagrammen in OBSCURE-Spezifikationen, werden eine große Anzahl von Operationen definiert, mit Hilfe derer ebenfalls Datenbankzustände definiert werden können. Im Gegenstatz zu den essentiellen Aktivitäten — diese greifen letztendlich auf die hier betrachteten Operationen zurück —, können mit den Operationen der Datenmodellebene beliebige Datenbanken (Träger der Sorte Db) beschrieben werden, also auch solche, die die Integritätsbedingungen verletzen. Datenbankzustände, die direkt durch Operationen der Datenmodellebene definiert werden, heißen im folgenden **allgemeine Datenbankzustände**. Es ist leicht einzusehen, daß die erreichbaren Datenbankzustände eine **Teilmenge** der allgemeinen Datenbankzustände bilden, weil alle essentiellen Aktivitäten ihrerseits Datenbankzustände mit Hilfe der Operationen der Datenmodellebene modellieren.

3.5.3 Integre Datenbankzustände

Als **integre Datenbankzustände** werden diejenigen allgemeinen Datenbankzustände bezeichnet, die die Integritätsbedingungen erfüllen. Mit Hilfe des OK-Prädikats kann für jede beliebige Datenbank überprüft werden, ob sie die Integritätsbedingungen erfüllt[4]. Weil für alle essentiellen Aktivitäten die Invarianz bzgl. der Integritätsbedingungen gefordert wird, gilt: Die erreichbaren Datenbankzustände bilden eine **Teilmenge** der integren Datenbankzustände. Ferner gilt trivialerweise, daß die integren Datenbankzustände eine **Teilmenge** der allgemeinen Datenbankzustände bilden.

3.5.4 Anmerkungen zu erreichbaren und integren Datenbankzuständen

Es erscheint zwar auf den ersten Blick sinnvoll, sich beim Rapid-Prototyping (oder auch bei einem späteren Testen der Implementierung) nur auf erreichbare Datenbankzustände zu konzentrieren, weil in der späteren Praxis nur diese Datenbankzustände tatsächlich auftreten werden, doch spricht folgendes Argument dagegen: Möglicherweise führt eine Änderung an einer einzigen essentiellen Aktivität dazu, daß eine ganze Anzahl bisher nicht erreichbarer Datenbankzustände — die also nicht betrachtet wurden — erreichbar werden. Konzentriert man sich beim Rapid-Prototyping (oder Testen) von vorneherein auf integre Datenbankzustände, hat man dieses Problem nicht.

[4]Die Integritätsbedingungen sollen statisch auf der Datenbank abprüfbar sein. Es sind also insbesondere nur solche Integritätsbedingungen zugelassen, die Turing-entscheidbar sind.

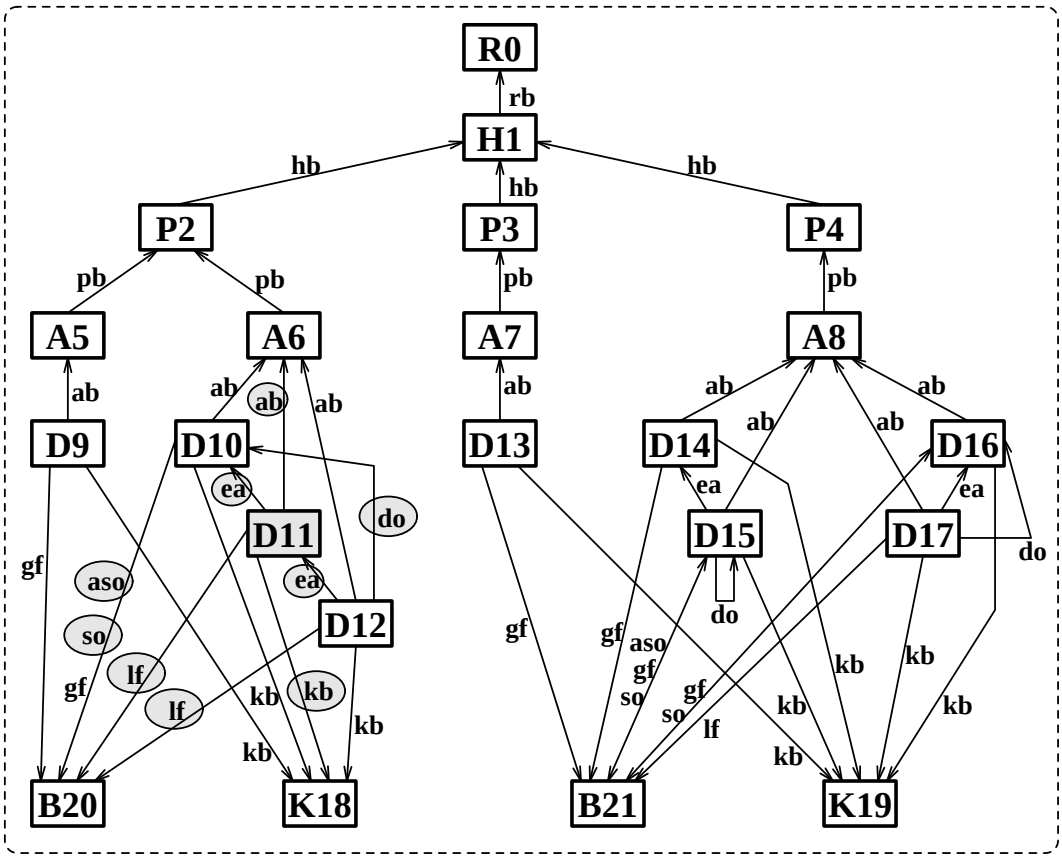

Abbildung 3.3: Ausgangsdatenbank einer Operationsanwendung von WSAV

3.5.5 Ein Beispiel: Ausgangsdatenbank zur Anwendung von WSAV

Weil nur die essentielle Aktivität WSAV spezifiziert wurde, kann ein Ausgangszustand der Datenbank nicht durch Anwendung anderer essentieller Aktivitäten aus dem Initialzustand aufgebaut werden, sondern nur mithilfe der Operationen der Datenmodellebene. Aus diesem Grund muß die Spezifikation der Datenmodellebene auch eine Anzahl geeigneter Operationen exportieren. Dies ist auch ein Grund dafür, warum so viele Sorten und Operationen im Modul DATENMODELLEBENE (siehe Seite 149) nicht vergessen werden und im Modul AKTIVITATENEBENE (siehe Seite 74) auf die FORGET_ALL_BUT-Konstruktion verzichtet wird. In dem folgenden Rapid-Prototyping wird die Operation WSAV also auf einen **allgemeinen Datenbankzustand** angewendet. Solange die anderen Systemaktivitäten nicht formal spezifiziert sind, kann auf keinen Fall entschieden werden, ob dieser Ausgangszustand der Datenbank auch **erreichbar** ist. Allerdings kann durch Anwendung des OK-Prädikats überprüft werden, ob dieser Datenbankzustand die Integritätsbedingungen erfüllt, also **integer** ist. Nach den Anmerkungen aus Abschnitt 3.5.4 erfüllt das im folgenden beschriebene Rapid-Prototyping auch dann seinen Sinn, wenn nicht

sichergestellt werden kann, ob es sich beim Ausgangszustand um einen erreichbaren Datenbankzustand handelt: Das Rapid-Prototyping kann dazu beitragen, Fehler bezüglich der Anforderungen an die Aktivität WSAV aufzuspüren.

Abbildung 3.3 beschreibt den Ausgangszustand der Datenbank mit graphischen Mitteln. Diese Ausgangsdatenbank entspricht dem in Kapitel 2, durch Abbildung 2.6, bereits grob erläuterten Zustand. Ausnahme: Der in Abbildung 2.6 vorgesehene Benutzer B22 wird hier nicht berücksichtigt. Der Konstruktorterm zu diesem Ausgangszustand ist im Anhang B ab Seite 175 abgebildet.

Je ein Rechteck repräsentiert eine Entity. Die Entities wurden insgesamt durchnumeriert, was die Zahlen hinter den Großbuchstaben erklärt. Die Großbuchstaben selbst, geben einen Hinweis zum Typ einer Entity: **R** steht für *Root*, **H** für *Hospital*, **P** für *Patient*, **A** für *Aufenthalt*, **D** für *Sunrise-Datenobjekt*, **K** für *Datenklasse* und **B** für *Benutzer*. Zwischen den Entities bestehen unterschiedliche Relationships. Diese werden im Bild durch benannte Verbindungslinien zwischen den Rechtecken repräsentiert. Der Namenszusatz einer Linie (dies können auch mehrere sein) gibt einen Hinweis zu bestehenden Relationships: **rb** steht für *Rootbezug*, **hb** für *Hospitalbezug*, **pb** für *Patientenbezug*, **ab** für *Aufenthaltsbezug*, **gf** für *Global-für*, **lf** für *Lokal-für*, **so** für *Stack-Objekt-von*, **aso** für *Aktuelles-Stack-Objekt-von*, **do** für *Default-Objekt*, **ea** für *Entstanden-aus* und **kb** für *Datenklassenbezug*.

Das globale Archiv dieses Datenbankzustands umfaßt die S.-Datenobjekte **D9**, **D10**, **D13**, **D14**, **D15** und **D16**. Von Benutzer **B20** wurde das S.-Datenobjekt **D10** aktuell weiterbearbeitet zu **D11** und dieses dann zu **D12**. **D10** definiert also einen Arbeitsstack im lokalen Archiv von **B20**, d. h. **D10** ist *Stack-Objekt-von* und *Aktuelles-Stack-Objekt-von* von B20. Das lokale S.-Datenobjekt **D12** ist das momentane *Default-Objekt* des Arbeitsstacks. Die S.-Datenobjekte **D10**, **D11** und **D12** haben in diesem Beispiel die gleiche Datenklasse — dies muß aber nicht so sein. Eine Erklärung der restlichen Beziehungen bleibt dem Leser überlassen.

3.5.6 Ein Beispiel: Resultatdatenbank einer Anwendung von WSAV

Auf den vorgestellten Datenbankzustand wird nun die Operation WSAV angewendet. Übergibt man dieser als weitere Argumente die Kennung des Benutzers **B20** und einen neuen Datenobjektnamen **n**, dann verändert diese ganz bestimmte Bestandteile der aktuellen Datenbank. Diejenigen, die gelöscht werden, sind in Abbildung 3.3 grau unterlegt. Abbildung 3.4 zeigt den resultierenden Datenbankzustand der Anwendung der Operation WSAV. Die grau unterlegten Teile beschreiben diesmal diejenigen Bestandteile, die, im Vergleich zur Ausgangssituation, neu aufgenommen wurden.

Es ist zu erkennen, daß das S.-Datenobjekt **D12** nun in der Relationship *Global-fuer* (**gf**) zum Benutzer **B20** steht, also global abgespeichert wurde. Gleichzeitig ist **D12** nun das definierende Objekt des aktuellen Arbeitsstacks von **B20** (Relationshiptypen *Stack-Objekt-von* (**so**) und *Aktuelles-Stack-Objekt-von* (**aso**)) und bildet zudem das Default-Objekt (Relationshiptyp *Default-Objekt* (**do**)) des Stacks für weitere Anwendungen. S.-Datenobjekt **D11** wurde als Zwischenresultat der Bearbeitung von Objekt **D10** zu **D12** ganz aus dem essentiellen Speicher entfernt. Dabei

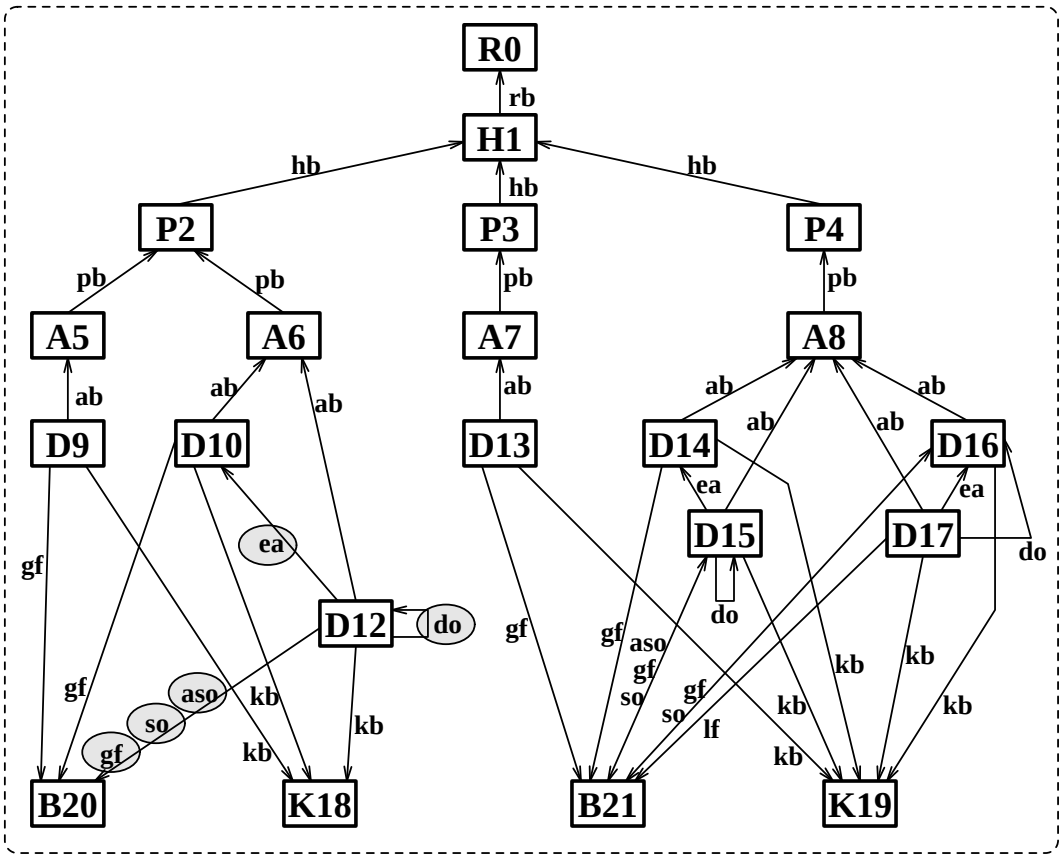

Abbildung 3.4: Resultatdatenbank einer Operationsanwendung von WSAV

wurden auch die Relationshipeinträge, an denen **D11** beteiligt war, eliminiert. Die Vorgänger-Information zu **D12** (Relationsiptyp *Entsanden-aus* (**ea**)) zeigt nun auf S.-Datenobjekt **D10**.

Man kann anhand dieser Anwendung nun analysieren, ob WSAV die gewünschten Eigenschaften aufweist. Die Auftraggeber eines Softwaresystems können sich bereits jetzt davon überzeugen, ob Ihre grundlegenden Erwartungen an das System richtig verstanden wurden. Werden in diesem frühen Stadium Mißverständnisse aufgedeckt, können die Änderungen mit geringem Aufwand erfolgen und, was besonders hervorzuheben ist, sie können *solide* durchgeführt werden, während die nachträglichen Änderungen bei fertig entwickelten Softwaresystemen oft als *Notlösungen* angesehen werden müssen.

Kapitel 4

Zusammenhang zur aktuellen Implementierung von Sunrise

Das Ziel dieses Kapitels ist es, die Beziehung der vorgestellten Anforderungsspezifikation zur aktuellen Implementierung des SUNRISE-Systems zu beschreiben. Dadurch, daß ein großer Bogen von der Anforderungsspezifikation bis zur Implementierung gespannt wird, werden einzelne Entwurfsentscheidungen, die der aktuellen Implementierung zugrundeliegen, angedeutet. Dieses Kapitel ist dabei in zweierlei Hinsicht interessant:

- Der Zusammenhang der Anforderungsspezifikation zur aktuellen Implementierung ist für SUNRISE-Laien nicht leicht ersichtlich. Nur dem System-Spezialisten wird es möglich sein, einzelne Komponenten der Anforderungsspezifikation auf deren Realisierung in der aktuellen Implementierung abzubilden. Dieses Kapitel versucht, diese Abbildung für die wichtigsten Komponenten des Ausschnitts *duales Speicherkonzept* informal zu beschreiben.
- Eine mögliche Anschlußarbeit zur formalen Spezifikation eines Systementwurfs zum Aussschnitt *duales Speicherkonzept* könnte sich zur Einarbeitung in das Thema an diesem Kapitel orientieren.

Zunächst wird der Begriff *Systementwurf* näher erläutert. Anschließend wird die aktuelle Implementierung informal beschrieben. Dazu wird das Beispiel zur Operationsanwendung WSAV aus Abschnitt 3.5 bzw. 2.3.1 erneut aufgegriffen. Es wird beschrieben, wie diese Operationsanwendung in der aktuellen Implementierung realisiert ist. Anschließend wird versucht, eine präzise Abbildung der Entity- und Relationshiptypen auf die Komponenten der Implementierung anzugeben. Es wird hier also keine informale Entwurfspezifikation vorgestellt, sondern eine solche wird durch die Abbildung der Anforderungsspezifikation auf die Implementierung nur angedeutet.

4.1 Begriff *Systementwurf*

In der Einleitung dieser Arbeit (Abschnitt 1.1) wurde das Wasserfallmodell vorgestellt. Ausgangspunkt der formalen Entwicklung von Software ist dabei eine Anforderungsspezifikation. Diese beschreibt, wie in Kapitel 3 ausführlich erläutert wurde, die logischen Anforderungen an das zu erstellende System und abstrahiert von Details, die aufgrund einer nicht perfekten Implementierungstechnologie gewählt werden. Resultat eines Vorgehens nach dem Wasserfallmodell ist eine Implementierung des Systems in einer (oder mehreren) Programmiersprache(n). Alle Entwicklungsebenen, die zwischen diesen beiden Extrema - Anforderungsspezifikation und Implementierung - liegen, werden als **Entwurfsebenen** bezeichnet. Eine Entwurfsspezifikation zeichnet sich dadurch aus, daß sie auf der einen Seite Detailentscheidungen enthält, die nicht der logischen Essenz eines Systems zuzuordnen sind, auf der anderen Seite aber von der Implementierung abstrahiert, indem sie nicht alle Details berücksicht, die durch den Programmcode beschrieben werden. Der Entwicklungsprozeß nach dem Wasserfallmodell sieht vor, sich ausgehend von der Anforderungsspezifikation über eine (oder mehrere) Entwurfsebene(n) der letztendlichen Implementierung zu nähern. Bei diesem Top-Down-Vorgehen werden auf jeder Entwurfsebene neue Entwurfsentscheidungen getroffen und formal beschrieben. Zunächst handelt es sich dabei um grobe Entscheidungen, die in den unteren Entwurfsebenen immer feiner werden.

Anmerkung: Leider standen keine Systemdokumente zu SUNRISE zur Verfügung, die eine reine Beschreibung einer Anforderungs- oder Entwurfsebene beinhalten. Im Sinne eines mehrstufigen Entwurfsprozesses vermischen die verfügbaren Dokumente grobe Entscheidungen höherer Entwurfsebenen oder der Anforderungsebene mit sehr feinen Details der unteren Ebenen oder der Implementierung.

4.2 Informale Erläuterungen zur aktuellen Implementierung

In diesem Abschnitt wird versucht, die Implementierung des SUNRISE-Systems anhand der exemplarischen Operationsanwendung von WSAV in Abschnitt 3.5 (beim Rapid-Prototyping) informal zu erläutern.

Ausgangssituation der folgenden Betrachtung ist: Das System wurde auf einer geeigneten Hardware (z. B. SUN-Workstation, UNIX-Betriebssystem) installiert. Das System befindet sich in einem Zustand, der demjenigen aus Abbildung 3.3 (Seite 78) entspricht. Das heißt, es kennt Objekte, die der Root **R0**, dem Hospital **H1**, den Patienten **P1** bis **P3**, den Aufenthalten **A5** bis **A8**, den Datenobjekten **D9** bis **D17**, den Benutzern **B20** und **B21**, sowie den Datenklassen **K18** und **K19** entsprechen. Außerdem kennt dieser Zustand genau die Beziehungsinformationen, die in Abbildung 3.3 dargestellt sind. Aktuell sind zwei Arbeitssitzungen geöffnet — eine durch den realen Benutzer zum Objekt **B21**, die andere durch denjenigen zum Objekt **B20**.

Von der Frage, ob dieser Zustand einen erreichbaren Datenbankzustand wider-

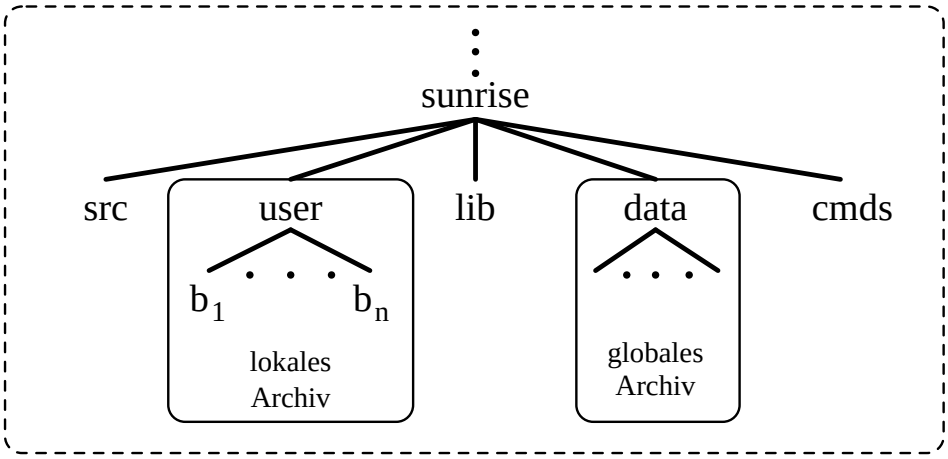

Abbildung 4.1: SUNRISE-Baumstruktur: Hauptebene

spiegelt oder nicht, wird hier weiter abstrahiert. Um diese Frage zu beantworten, hätte z. B. das aktuelle SUNRISE-System initialisiert werden können, um dann den Versuch zu starten, den hier beschriebenen Zustand durch Anwendung essentieller Systemaktivitäten zu erreichen. Aus Aufwandsgründen erfolgte dies nicht.

Die folgenden Ausführungen konzentrieren sich nicht so sehr darauf, wie das System über Fenster und Menüs mit dem Benutzer kommuniziert — dies sagt wenig über die Implementierung aus —, sondern welche Beobachtungen bei der Operationsanwendung im UNIX-Dateibaum und in einzelnen UNIX-Dateien gemacht werden können. Spätestens hier muß erwähnt werden, daß eine der wichtigsten Entwurfsentscheidungen, die der aktuellen Implementierung zugrunde liegen, die massive Einbindung der hierarchischen Struktur des **UNIX-Dateibaums** zur Verwaltung von Beziehungsinformationen (Relationshiptypen) betrifft. Der Entscheidung, Relationshiptypen über eine Baumstruktur zu verwalten, gilt im folgenden das Hauptinteresse.

4.2.1 Ein Beispiel: Ausgangszustand einer Anwendung von WSAV

Der reale Benutzer zu **B20** könnte (in der angedeuteten Ausgangssituation) in einer UNIX-Shell folgende Beobachtungen machen: Wechselt er zum Verzeichnis **.../sunrise** (der Pfad dorthin wurde bei der Installierung festgelegt), kann er sich von der in Abbildung 4.1 dargestellten Unterbaumstruktur überzeugen. Hier weniger interressant sind die Unterbäume **src**, **lib** und **cmds**. Unter **cmds** befinden sich die installierten SUNRISE-Module, die jedem Benutzer zur Verfügung stehen, unter **lib** die Bibliotheken und globalen Konfigurationsdateien und unter **src** die Headerdateien und der sonstige Quellcode. Nähere Angaben enthalten die Systemdokumente [BSS93] und [Sta93]. Wichtig für diese Betrachtung sind die Unterbäume zu **user** und **data**. Während der Baum unter **data** das **globale Archiv** enthält, befinden sich im Baum unter **user**, neben weiteren benutzerspezifischen Komponenten, die **lokalen Arbeitsarchive** der einzelnen Benutzer. Diese physikalische Trennung

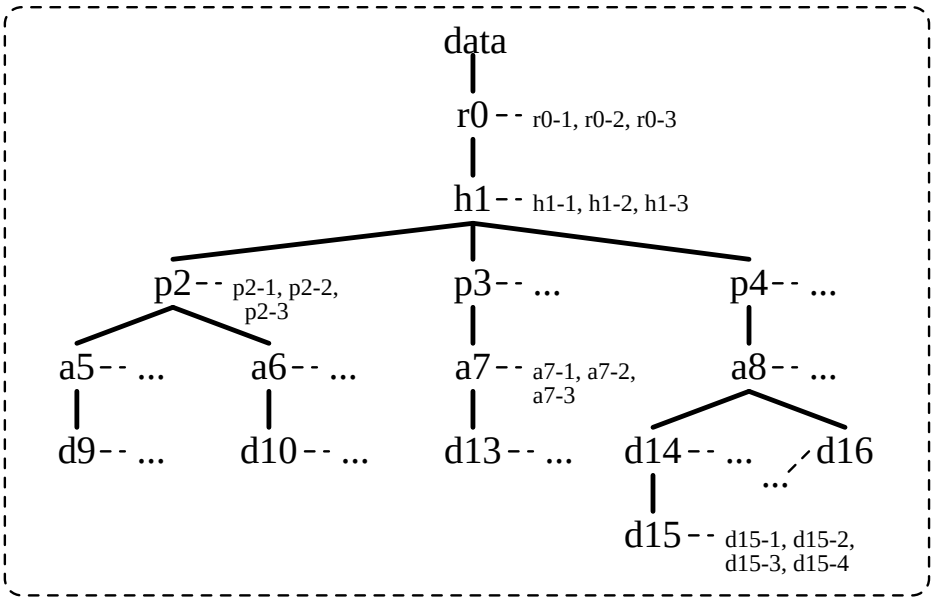

Abbildung 4.2: SUNRISE-Baumstruktur: globales Archiv (Ausgangszustand)

der lokalen Archive vom globalen Archiv erklärt auch die Wahl des Begriffs **duales Speicherkonzept**.

Wechselt unser Benutzer nun in das Verzeichnis **data**, so findet er die Unterstruktur von Abbildung 4.2 vor. Das Unterverzeichnis **r0** enthält Dateien, deren Einträge den Attributinformationen zur Entity **R0** des Typs Root aus Abbildung 3.3 entsprechen. Die Zusammengehörigkeit dieser Dateien wird durch die Dateinamen angezeigt. Von der konkreten Anzahl der Dateien wird hier abstrahiert. Wichtig ist lediglich, daß die Attributinformationen durch mehrere Dateien in einem ausgezeichneten Verzeichnis verwaltet werden. Das Verzeichnis **r0** und die angesprochenen Informationsdateien realisieren gemeinsam die Entity **R0** der Anforderungsspezifikation.

Als Unterverzeichnis von **r0** gibt es ein Verzeichnis **h1**. Die Dateien in diesem Verzeichnis enthalten wiederum die Attributinformationen, die denen zur Hospital-Entity **H1** aus Abbildung 3.3 entsprechen. Der Hospitalbezug von **H1** zu **R0** wird durch die **Ist-Unterverzeichnis-von**-Beziehung im UNIX-Dateibaum zwischen den Verzeichnissen **h1** und **r0** verwaltet. Das erläuterte Prinzip gilt auch für die weiteren Ebenen. Die Patienten-Entities (kurz: Patienten), die der Hospital-Entity **H1** zugeordnet werden, sind durch Unterverzeichnisse von Verzeichnis **h1** im Dateibaum realisiert. Zu Patient **P2** existieren bereits die beiden Aufenthaltseinträge **A5** und **A6** (die Verzeichnisse **a5** und **a6** sind Unterverzeichnisse von **p2**). In diesem Beispiel wurden zu allen Aufenthalten auch Untersuchungsdaten ermittelt. Das S.-Datenobjekt **D9** (Verzeichnis **d9**) gehört zum Aufenthalt **A5** (Verzeichnis **a5**) und **D10** (Verzeichnis **d10**) zu **A6** (Verzeichnis **a6**). Auch bei den S.-Datenobjekten werden die Attributinformationen durch Dateien (3 oder 4) im entsprechenden Di-

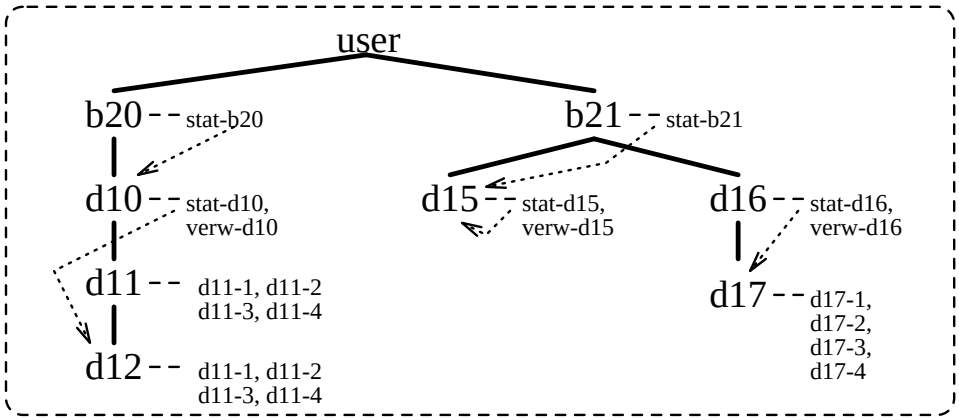

Abbildung 4.3: SUNRISE-Baumstruktur: lokale Arbeitsarchive (Ausgangszustand)

rectory verwaltet. S.-Datenobjekte, die aus einem anderen durch Anwendung einer Bearbeitungsoperation entstanden sind, werden wiederum als Unterverzeichnisse im Dateibaum angelegt. In diesem Beispiel wurde das S.-Datenobjekt **D14** weiterbearbeitet zu **D15**, **d15** ist also Unterverzeichnis von **d14**.

Anmerkung: Die aktuelle Implementierung unterscheidet zwischen *prozessierten* S.-Datenobjekten (diese gehen aus einer Operationsanwendung hervor, haben aber die gleiche Datenklasse wie das Vorgängerobjekt) und *subordinaten* S.-Datenobjekten (bei Operationsanwendung verändert sich die Datenklasse). *Prozessierte* S.-Datenobjekte werden auf der gleichen Ebene wie ihr Vorgängerobjekt verwaltet, während nur *subordinate* S.-Datenobjekte, wie hier beschrieben, als Unterverzeichnisse angelegt werden. Von dieser Differenzierung wird hier abstrahiert. Formale Entwurfsspezifikationen, die sich dieser Arbeit anschließen könnten, müßten diese Differenzierung allerdings berücksichtigen. Hier wird dagegen versucht, die Anforderungsspezifikation in groben Zusammenhang zur Implementierung zu setzen, alle Feinheiten können dabei aus Umfangsgründen nicht berücksichtigt werden.

Als nächstes wird der Dateibaum unter **user** erläutert. Sieht man von hier nicht betrachteten benutzerspezifischen Komponenten ab, so hat im Ausgangszustand der Dateibaum unter **.../sunrise/user** die in Abbildung 4.3 beschriebene Struktur. Die Unterverzeichnisse **b20** und **b21** enthalten die lokalen Arbeitsarchive der Benutzer zu den Entities **B20** und **B21** aus Abbildung 3.3. Die Verzeichnisse unter diesen Benutzerverzeichnissen (hier sind dies: **d10** unter **b20** und **d15**, **d16** unter **b21**) definieren je einen Arbeitsstack. Der Benutzer **B20** hat also in seiner laufenden Arbeitssitzung das S.-Datenobjekt **D10** zur Bearbeitung ausgewählt. Dabei wurde das Verzeichnis **d10**, sowie die Dateien **stat-b20**, **stat-d10** und **verw-d10** angelegt. Die Statusdatei **stat-b20** enthält einen Verweis auf den aktuellen Arbeitsstack **d10**, **stat-d10** auf das Default-Objekt dieses Stacks, dies ist **d12**, und **verw-d10** stellt den Zusammenhang zwischen dem lokalen und globalen Archiv her, indem es auf das Verzeichnis **d10** im globalen Archiv (Abbildung 4.2) verweist. Das S.-Datenobjekt **D10** wurde weiterbearbeitet zu **D11**, also wurde ein Unterverzeichnis **d11** ange-

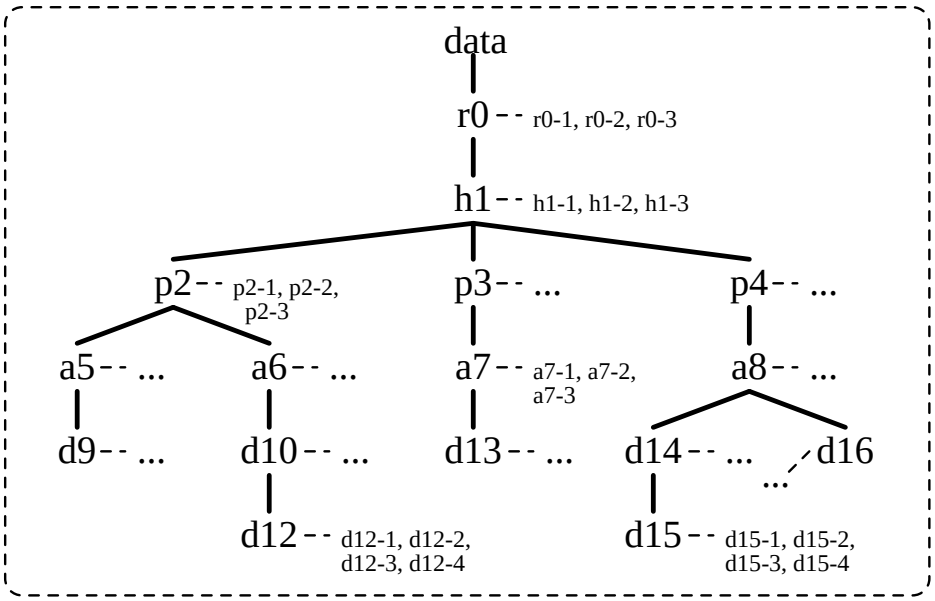

Abbildung 4.4: SUNRISE-Baumstruktur: globales Archiv (Resultatzustand)

legt, daß die Informationsdateien **d11-1** bis **d11-4** enthält. Das Unterverzeichnis **d12** enthält die Informationsdateien zu einem weiter abgeleiteten S.-Datenobjekt **D12**. Dieses ist das aktuelle Default-Objekt des Stacks. Der Benutzer **B21** hat zwei Arbeitsstacks geöffnet, dabei wurden die beiden Unterverzeichnisse **d15** und **d16** angelegt. Der Verweis in Datei **stat-b21** zeigt den aktuellen Arbeitsstack des Benutzers an, indem es auf eines der definierenden Stackverzeichnisse[1] verweist (hier auf **d15**). Die Dateien **stat-d15** und **stat-d16** enthalten Verweise zu den jeweiligen Default-Objekten der beiden Stacks. Wie schon zuvor bei **d10**, stellen die Dateien **verw-d15** und **verw-d16** den Bezug zum globalen Archiv her. S.-Datenobjekt **D17** ist, wie schon **D11** und **D12**, ein lokales, prozessiertes Objekt. Weil es aus **D16** entstanden ist, wurde unter **d16** das Verzeichnis **d17** angelegt und dort wurden die Informationsdateien **d17-1** bis **d17-4** zu **D17** abgespeichert.

4.2.2 Ein Beispiel: Resultatzustand einer Anwendung von WSAV

Analog zum Rapid-Prototying in Abschnitt 3.5 wird nun die Operation WSAV auf den Ausgangszustand angewendet. Die sich dadurch ergebenden Veränderungen werden anschließend untersucht.

Zur Anwendung von WSAV aktiviert der Benutzer **B20** mit der Maus im SUNRISE-Hauptfenster (dieses enthält aktuell eine Visualisierung von Datenobjekt **D12** — dem Default-Objekt des aktuellen Arbeitsstacks) den Menüpunkt **wsav** im Menüfenster **file**. Es erscheint ein Eingabefenster, indem der Benutzer zur Eingabe

[1] Die definierenden Stackverzeichnisse sind die Verzeichnisse im lokalen Archiv, die unterhalb der Benutzerverzeichnisse angesiedelt sind.

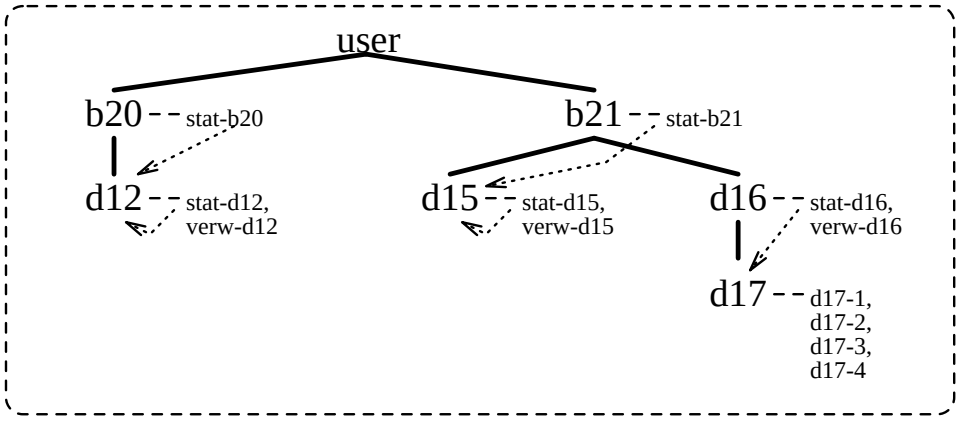

Abbildung 4.5: SUNRISE-Baumstruktur: lokale Abeitsarchive (Resultatzustand)

eines Datennamens aufgefordert wird. Die ebenfalls benötigte Kennungsinformation zum ausführenden Benutzer bezieht WSAV über das UNIX-Betriebssystem. Es erfolgt dann die Anwendung von WSAV.

Der resultierende Zustand kann nun analog zum Ausgangszustand analysiert werden. Zunächst wird das globale Archiv betrachtet. Im Unterbaum zu **data** könnte sich der Benutzer **B20** von der in Abbildung 4.4 dargestellten Verzeichnisstruktur überzeugen. Neu angelegt wurde ein Verzeichnis **d12**, das die Informationsdateien **d12-1** bis **d12-4** enthält. **d12** wurde als Unterverzeichnis von **d10** angelegt, weil das Objekt **D12** aus dem Objekt **D10** abgeleitet wurde. Diese Information konnte WSAV im Ausgangszustand über die indirekte Unterverzeichnisbeziehung von **d12** zu **d10** ermitteln (siehe Abbildung 4.3). Dies sind bereits alle Veränderungen, die sich für das globale Archiv ergeben. Anzumerken ist lediglich, daß die Benutzerinformation zum S.-Datenobjekt **D12** nicht verloren ist. Ein zusätzlicher Eintrag in einer der Informationsdateien zeigt an, durch welchen Benutzer das Objekt im globalen Archiv angelegt wurde.

Ein wenig umfangreicher sind die Veränderungen im lokalen Archiv des Benutzers zu **B20**. Der resultierende Zustand der Operationsanwendung wird durch Abbildung 4.5 beschrieben. Das Objekt **D12** ist nun das neue definierende und zugleich auch Default-Objekt des aktuellen Arbeitsstacks des Benutzers zu **B20**. Die Verzeichnisse **d10** und **d11** wurden zuvor komplett aus dem lokalen Archiv entfernt und Verzeichnis **d12** wurde direkt unter Benutzerverzeichnis **b20** angelegt. Die Datei **stat-d12** enthält den Hinweis auf das Default-Objekt, und die Datei **verw-d12** stellt die Verbindung zum globalen Archiv her, denn dort wurde Objekt **D12**, wie bereits erläutert als globales Objekt angelegt. Obwohl das Objekt **D11** ganz eliminiert wurde, kann es, aufgrund der (totalen) Geschichtsinformation in einer der Informationsdateien zu Objekt **D12** im globalen Archiv, aus einem Vorgängerobjekt von **D12** wieder hergestellt werden. Das Objekt **D10** existiert nach wie vor im globalen Archiv.

4.3 Abbildung der formalen Spezifikation auf Bestandteile der Implementierung

Während im vorigen Abschnitt versucht wurde, wichtige Aspekte der Implementierung zum Ausschnitt **duales Speicherkonzept** an einem Beispiel zu erläutern, soll nun die Abbildung der Entity- und Relationshiptypen der Anforderungsspezifikation auf die angesprochenen Komponenten der Implementierung informal, aber möglichst präzise beschrieben werden. Zuvor werden aber noch einige Vereinbarungen getroffen:

Ein Verzeichnis **d1** ist Vorgängerverzeichnis von Verzeichnis **d2** im Dateibaum D genau dann, wenn es einen Pfad über die Unterverzeichnisstruktur in D von **d1** zu **d2** gibt.

Der Begriff Unterverzeichnis wird gemäß seiner herkömmlichen Bedeutung verwendet: Verzeichnis **d2** ist Unterverzeichnis von Verzeichnis **d1** im Dateibaum D, falls **d2** Kind von **d1** im Baum D ist.

Im folgenden wird davon ausgegangen, daß sowohl im lokalen Archiv, wie auch im globalen, alle Verzeichnisse unterschiedliche Namen (Bezeichner) tragen.

Es werden nun die verschiedenen Entity- und Relationshiptypen einzeln diskutiert.

- **Entitytyp Benutzer**
  Die Informationen zu den Entities des Typs Benutzer werden in der aktuellen Implementierung über die Benutzerverwaltung des UNIX-Filesystem realisiert. Ferner enthält das lokale Archiv (und andere hier nicht betrachtete Teile des gesamten SUNRISE-Teilbaums) Unterverzeichnisse, deren Namen den Kennungsinformationen der Benutzer entsprechen (Pfade **.../sunrise/user/b**$_i$).

- **Entitytyp Root**
  Die Entities des Typs Root werden durch Unterverzeichnisse im globalen Archiv (Pfade **.../sunrise/data/r**$_i$) verwaltet. Die Attributinformationen zu einer Entity werden durch strukturierte Einträge in den Dateien eines solchen Unterverzeichnisses gespeichert. Also: Das Verzeichnis **r** (Pfad **.../sunrise/data/r**) realisiert gemeinsam mit den in diesem abgelegten Dateien eine Entity **R**.

- **Entitytyp Hospital**
  Durch Unterverzeichnisse der Root-Ebene werden im lokalen Archiv die Entities des Typs Hospital realisiert. Auch hier enthalten mehrere Informationsdateien die Attributinformationen. Also: Das Verzeichnis **h** (Pfad **.../sunrise/data/r/h**, wobei r ein beliebiges Rootverzeichnis ist) realisiert gemeinsam mit den in diesem abgelegten Dateien eine Entity **H**.

- **Entitytyp Patient**
  Die Realisierung dieses Entitytyps erfolgt analog zum Entitytyp Root (bzw. Hospital), allerdings unterhalb der Hospital-Ebene.

- **Entitytyp Aufenthalt**
  Dieser Entitytyp wird ebenfalls analog zum Entitytyp Root (bzw. Hospital) realisiert, allerdings unterhalb der Patient-Ebene.

- **Entitytyp S.-Datenobjekt.**
  Entities des Typs S.-Datenobjekte werden ebenfalls durch je ein Verzeichnis mit drei (oder vier) Informationsdateien realisiert. Wie schon zuvor werden die Attributinformationen durch strukturierte Einträge in diesen Informationsdateien gespeichert. Im globalen Archiv befinden sich die S.-Datenobjekt-Verzeichnisse unterhalb der Aufenthalt-Ebene. Im lokalen Archiv werden sie unterhalb der privaten Verzeichnisse der einzelnen Benutzer verwaltet (nur die erste Unterebene enthält dabei keine vollständigen Objekte, sondern Verweise auf die entsprechenden Objekte im globalen Archiv).

- **Entitytyp Datenklasse**
  Der Entitytyp Datenklasse wird in der Implementierung analog zu den Attributen des Entitytyps S.-Datenobjekt behandelt (siehe auch Anmerkung auf Seite 28). Als zusätzlicher Eintrag in einer der Informationsdateien eines S.-Datenobjekts wird dessen Datenklasseninformation vermerkt.

- **Relationshiptyp Rootbezug**
  Über die Beziehung *ist Unterverzeichnis von* der Verzeichnisse der Hospitalebene zu denen der Rootebene, werden die Rootbeziehungen der Hospitalobjekte realisiert:
  Hospital-Objekt **H** steht in Rootbezug zu Root-Objekt **R** genau dann, wenn das Verzeichnis **h** Unterverzeichnis von Verzeichnis **r** im globalen Archiv (Dateibaum) ist.

- **Relationshiptyp Hospitalbezug**
  Ebenfalls über die Beziehung *ist Unterverzeichnis von* erfolgt die Realiserung der Hospitalbeziehungen der Patientenobjekte:
  Patient-Objekt **P** steht in Hospitalbezug zu Hospital-Objekt **H** genau dann, wenn das Verzeichnis **p** Unterverzeichnis von Verzeichnis **h** im globalen Archiv (Dateibaum) ist.

- **Relationshiptyp Patientenbezug**
  Analog zu Hospitalbezug und Rootbezug.

- **Relationshiptyp Aufenthaltsbezug**
  Die Aufenthaltsbezüge der Datenobjekte werden in der Implementierung über die Beziehung *ist Vorgängerverzeichnis von* verwaltet. Im Gegensatz zu Root-, Hospital- und Patientenbezug werden also nicht nur die direkten Vorgängerverzeichnisse im Dateibaum betrachtet, sondern ausgehend einem S.-Datenobjektverzeichnis soweit in der Verzeichnishierarchie zurückgegangen, bis man auf ein Aufenthaltsverzeichnis stößt. Auf diese Weise kann der Aufenthaltsbezug zumindest für alle Objekte im globalen Archiv bestimmt werden.

Für die Objekte in den lokalen Archiven besteht über die Verweisdateien (z. B. **verw-d10** in Abbildung 4.3) der definierenden Stackverzeichnisse (z. B. Verzeichnis **d10** in Abbildung 4.3) eine Anbindung an das globale Archiv. Ein Beispiel: Für Datenobjekt **D12** in Abbildung 4.3 wird der Aufenthaltsbezug über die Verzeichnisse **d12**, **d11** und **d10** im lokalen Archiv, dann über den Verweis in Datei **verw-d10** auf das Verzeichnis **d10** hin zum Aufenthaltsverzeichnis **a6** im globalen Archiv (siehe Abbildung 4.2) hergestellt.
Also: S.-Datenobjekt **D** steht in Aufenthaltsbezug zu Aufenthalt-Objekt **A** genau dann, wenn das Verzeichnis **a** Vorgängerverzeichnis von Verzeichnis **d** im globalen Archiv ist oder wenn ein Verzeichnis **d'** im lokalen Archiv Vorgängerverzeichnis von **d** ist und das, über die Verweisdatei **verw-d'** identifizierte, Verzeichnis **d"** im globalen Dateibaum als Vorgängerverzeichnis das Verzeichnis **a** aufweist.

- **Relationshiptyp Entstanden-aus**
  Dieser Relationshiptyp wird über die direkte Beziehung *ist Unterverzeichnis von* zwischen zwei Datenobjektverzeichnissen realisiert. Ausnahme: Über die Verweisdateien (z. B. **verw-d10** in Abbildung 4.3) in den lokalen Archiven wird wieder die Anbindung an die Objekte im globalen Archiv realisiert.
  S.-Datenobjekt **D1** ist entstanden aus S.-Datenobjekt **D2** genau dann, wenn das Verzeichnis **d2** Vorgängerverzeichnis von Verzeichnis **d1** im globalen Archiv ist oder wenn ein lokales Verzeichnis **d'** Vorgängerverzeichnis von **d1** im lokalen Archiv ist und der Bezug zu Verzeichnis **d2** des globalen Archivs über eine Verweisdatei **verw-d'** hergestellt wird.

- **Relationshiptyp Datenklassenbezug**
  Wie bereits beim Entitytyp Datenklasse erläutert, werden die Datenklasseninformationen der S.-Datenobjekte durch einen zusätzlichen Eintrag in einer der Informationsdateien eines jeden S.-Datenobjekts angezeigt.
  S.-Datenobjekt **D** hat den Datenklassenbezug **K** genau dann, wenn in einer der Informationsdateien **d-1** bis **d-4** ein Eintrag existiert, der die Datenklasse **K** identifiziert.

- **Relationshiptyp Stack-Objekt-von**
  Die Verzeichnisse auf der ersten Unterebene eines Benutzerverzeichnisses im lokalen Archiv verweisen (über ihre Verweisdateien) auf die Stack-Objekte des jeweiligen Benutzers. Bei Eröffnung eines neuen Arbeitstacks wird im lokalen Archiv des jeweiligen Benutzers ein neues Unterverzeichnis und eine Verweisdatei mit entsprechendem Hinweis auf das ausgewählte Objekt im globalen Archiv angelegt. (Beachte: Alle Stack-Objekte sind globale Objekte.)
  Ein S.-Datenobjekt **D** ist Stack-Objekt von Benutzer-Objekt **B** genau dann, wenn auf das Verzeichnis **d** des globalen Archivs, durch eine Verweisdatei in einem der Unterverzeichnisse des lokalen Benutzerverzeichnisses **b** verwiesen wird.

- **Aktuelles-Stack-Objekt-von**
  Durch eine Verweisdatei **stat-b**$_i$ wird im lokalen Benutzerverzeichnis b$_i$ ein Hinweis auf das definierende Stackverzeichnis des aktuellen Arbeitsstacks angelegt. Ein Beispiel: In Abbildung 4.3 wird durch die Datei **stat-b20** das Verzeichnis **d10** als definierendes Stackverzeichnis des aktuellen Arbeitsstacks des Benutzers zu **b20** ausgewiesen.
  Ein S.-Datenobjekt **D** ist aktuelles Stack-Objekt von Benutzer-Objekt **B** genau dann, wenn auf das lokale Verzeichnis **d**, durch eine Verweisdatei des lokalen Benutzerverzeichnisses **b**, verwiesen wird.

- **Relationshiptyp Default-Objekt**
  Eine weitere Verweisdatei in jedem definierenden Stackverzeichnis, zeichnet das Verzeichnis des aktuellen Default-Objekts dieses Stacks aus.
  Ein S.-Datenobjekt **D1** ist Default-Objekt eines S.-Datenobjekts **D2** (Stack-Objekt) genau dann, wenn im lokalen Verzeichnis **d2** eine Verweisdatei existiert, die auf das lokale Verzeichnis **d1** verweist.

- **Relationshiptyp Lokal-fuer**
  Die lokalen S.-Datenobjekte eines Benutzers befinden sich im lokalen Archiv unter dessen Benutzerverzeichnis. Über die Beziehung *ist Vorgängerverzeichnis von* wird für die S.-Datenobjekte im globalen Archiv das entsprechende Benutzerverzeichnis ermittelt. Ausnahme: Die definierenden Stackverzeichnisse (diese enthalten sowieso nur eine Verweisdatei mit Verweis auf das globale Archiv) repräsentieren keine lokalen, sondern globale Objekte.
  Ein S.-Datenobjekt **D** ist lokales Objekt von Benutzer-Objekt **B** genau dann, wenn **D** kein Stack-Objekt von **B** ist, und das lokale Benutzerverzeichnis **b** Vorgängerverzeichnis von Verzeichnis **d** ist.

- **Relationshiptyp Global-fuer**
  Dieser Relationshiptyp wird über einen zusätzlichen Dateieintrag in einer der Informationsdateien eines globalen S.-Datenobjekts realisiert. Dieser Eintrag verweist auf den Benutzer, dem das Objekt zugeordnet ist.
  S.-Datenobjekt **D** ist globales Objekt von Benutzer-Objekt **B** genau dann, wenn das Verzeichnis **d** im globalen Archiv existiert, und die Informationsdateien einen Eintrag enthalten, der das Benutzer-Objekt **B** identifiziert.

Anmerkung: Der Einbezug des UNIX-Dateisystems in die Implementierung des Systems bringt folgende Vorteile mit sich: Die Auslagerung von Unterbäumen des globalen (oder auch eines lokalen Archivs) auf externe Datenträger kann relativ mühelos erfolgen. Beim Wiedereinfügen muß allerdings darauf geachtet werden, daß dies an der richtigen Stelle geschieht. Außerdem können neben den Zugriffsrechten, die ohnehin durch die Aufteilung in lokale und einen globalen Speicherbereich verwaltet werden, zusätzlich individuelle Zugriffsrechte für jeden Benutzer über das UNIX-Betriebssystem vergeben werden.

Kapitel 5

Sicherheits-Gütezertifikate

Die Sicherheit von Systemen der Informationstechnik (IT) ist in den vergangenen Jahren zunehmend zum Diskussionsstoff in der Öffentlichkeit geworden. Es ist zu erwarten, daß sich die Forderungen nach *sicheren Systemen* in den kommenden Jahren weiter verstärken und damit die zertifizierte und nachweisbare Sicherheit eines Softwaresystems ein wichtiger Wettbewerbsfaktor werden wird. Was ist aber in diesem Zusammenhang unter *Sicherheit* zu verstehen und wie kann sie beurteilt werden? Erste Anstrengungen, diese Fragen zu beantworten, wurden ab 1967 in den USA unternommen. Allerdings beschränkten sich diese Überlegungen zunächst auf den militärischen Bereich. Als Ergebnis wurde 1983 das Dokument *Trusted Computer System Evaluation Criteria* (TCSEC) [TCS85] — auch *Orange Book* genannt — herausgegeben, in dem zum erstenmal wünschenswerte Eigenschaften sicherer Systeme präzisiert wurden.

In Europa entwickelten unterschiedliche Länder, aufbauend auf [TCS85], eigene Kriterienkataloge zur Bewertung der Sicherheit von IT-Systemen. Als Beispiele sind zu nennen: das *CESG Memorandum Nr.3* [CES89] und das *Grüne Buch* [DTI89] in Großbritannien, das *Blau-weiß-rote Buch* [SCS89] in Frankreich und [ZSI89] der Zentralstelle für Sicherheit in der Informationstechnik (ZSI) — jetzt Bundesamt für Sicherheit in der Informationstechnik (BSI) — in der Bundesrepublik Deutschland.

Die Bundesrepublik Deutschland, Frankreich, Großbritannien und die Niederlande beschlossen, die nationalen Kataloge weiterzuentwickeln zu einem gemeinsamen Kriterienkatalog. Ergebnis dieser Anstrengungen sind die *Kriterien für die Bewertung der Sicherheit von Systemen der Informationstechnik* (ITSEC) [ITS91] (siehe auch [ZSI90]), herausgegeben von der Europäischen Gemeinschaft. Dieser Katalog gibt Richtlinien zur Evaluierung von IT-Systemen vor, die als Grundlage für die Bewertung eines IT-Systems durch unabhängige Prüfstellen dienen sollen.

Ein erfolgreich evaluiertes IT-System erhält dann ein Gütezertifikat, das über nationale Grenzen hinweg anerkannt wird.

5.1 Überblick zu den IT-Sicherheitskriterien ITSEC

Grundsätzlich wird in [ITS91] zwischen **IT-System** (eindeutig definierte Einsatzumgebung) und **IT-Produkt** (nur allgemeine Annahmen über die Einsatzumgebung sind vorhanden) unterschieden. Während zur Bedrohung der Sicherheit von IT-Systemen konkrete Aussagen getroffen werden können, sind für IT-Produkte dazu nur allgemeine Annahmen möglich. Dieser Überblick unterscheidet nicht zwischen IT-Systemen und IT-Produkten, sondern verwendet — quasi als Oberbegriff — **IT-System**. Der Begriff Sicherheit zerfällt in [ITS91] in folgende drei Unterbegriffe:

- **Vertraulichkeit** : Schutz vor unbefugter Preisgabe von Informationen.
- **Integrität** : Schutz vor unbefugter Veränderung von Informationen.
- **Verfügbarkeit** : Schutz vor unbefugter Vorenthaltung von Informationen.

Zur Evaluierung eines IT-Systems müssen vom Antragsteller — dies ist in der Regel der Systemhersteller — die **Sicherheitsvorgaben** eingereicht werden. Diese umfassen Angaben zu den **erwarteten Bedrohungen**, den **Sicherheitszielen** und den **sicherheitsspezifischen Funktionen**. Die sicherheitsspezifischen Funktionen sind diejenigen, die den erwarteten Bedrohungen entgegenwirken sollen, um die Sicherheitsziele zu realisieren. Zur Formulierung der Sicherheitsziele wird aus Gründen der Übersichtlichkeit empfohlen, eine Einteilung nach den folgenden **generischen Oberbegriffen** vorzunehmen:

- **Identifizierung**
- **Zugriffskontrolle**
- **Beweissicherung**
- **Protokollauswertung**
- **Wiederaufbereitung**
- **Unverfälschtheit**
- **Zuverlässigkeit der Dienstleistung**
- **Übertragungssicherheit**

Weil viele Systeme ähnlichen Bedrohungen ausgesetzt sind, und sich damit auch ähnliche Sicherheitsziele ergeben, gibt es 10 **vordefinierte Funktionalitätsklassen**, auf die der Antragsteller bei der Formulierung der Systemziele (teilweise oder ganz) verweisen darf. Auch der Verweis auf andere IT-Sicherheitspolitik-Dokumente ist erlaubt.
Desweiteren enthalten die Sicherheitsvorgaben die **postulierte Mindeststärke** der Sicherheitsmechanismen bzw. der sicherheitsspezifischen Funktionen (**Wirksamkeitsaspekt** der Vertrauenswürdigkeit eines IT-Systems). Zwischen folgenden Stärken wird unterschieden:

- **niedrig** : Schutz vor unbeabsichtigtem Verstoßen gegen die Sicherheitsziele, der aber von sachkundigen Angreifern überwunden werden kann.
- **mittel** : Schutz gegenüber Angreifern mit beschränkten Gelegenheiten oder Betriebsmitteln.
- **hoch** : Schutz, der nur von äußerst sachkundigen Angreifern (mit sehr guten Gelegenheiten und Betriebsmitteln) überwunden werden kann.

Von der Prüfstelle muß überprüft werden, ob die postulierte Einstufung der Sicherheitsmechanismen auch tatsächlich gerechtfertigt ist, d. h. die Wirksamkeit der Mechanismen muß gemäß den Anforderungen der angestrebten Mindeststärke überprüft werden.

Neben dem Wirksamkeitsaspekt spielt der **Korrektheitsaspekt** der Vertrauenswürdigkeit eines IT-Systems eine wichtige Rolle für die Evaluation. Hierzu wird untersucht, ob die angegebenen sicherheitsspezifischen Funktionen auch korrekt sind, d. h. das tun, was von ihnen verlangt wird. Dazu werden in [ITS91] **7 Evaluationsstufen E0-E6** (auch **Korrektheitsstufen** oder **Qualitätstufen** genannt) definiert, von denen der Antragsteller eine als **angestrebte Evaluationsstufe** für sein IT-System auswählt. Im wesentlichen legen diese Stufen unterschiedliche Anforderungen an die Entwicklungs- und Systembeschreibungsdokumente fest, auf deren Basis die Korrektheit der sicherheitsspezifischen Funktionen analysiert werden soll. Konkret werden Anforderungen zu folgenden Aspekten definiert: Qualität der Sicherheitsanforderungen, Qualität des Architekturentwurfs, Qualtität des Feinentwurfs, Qualität der Implementierung, Qualität der Konfigurationskontrolle, Qualität der verwendeten Programmiersprachen und Compiler, Qualität der Sicherheit beim Entwickler, Qualität der Betriebsdokumentation, Qualität der Auslieferung und Konfiguration, Qualität des Anlaufs und des Betriebs.
Die 7 Evaluationstufen werden kurz skizziert:

- **E0** : unzureichende Vertrauenswürdigkeit
- **E1** : getestet
- **E2** : methodisch getestet
- **E3** : methodisch getestet und teilanalysiert
- **E4** : informell analysiert
- **E5** : semi-formal analysiert
- **E6** : formal analysiert (**E7** : formal verifiziert)

Die folgende Abbildung 5.1 zeigt die Anforderungen, aufgegliedert nach den oben aufgelisteten Aspekten, an die vom Antragsteller zu erbringenden Unterlagen und

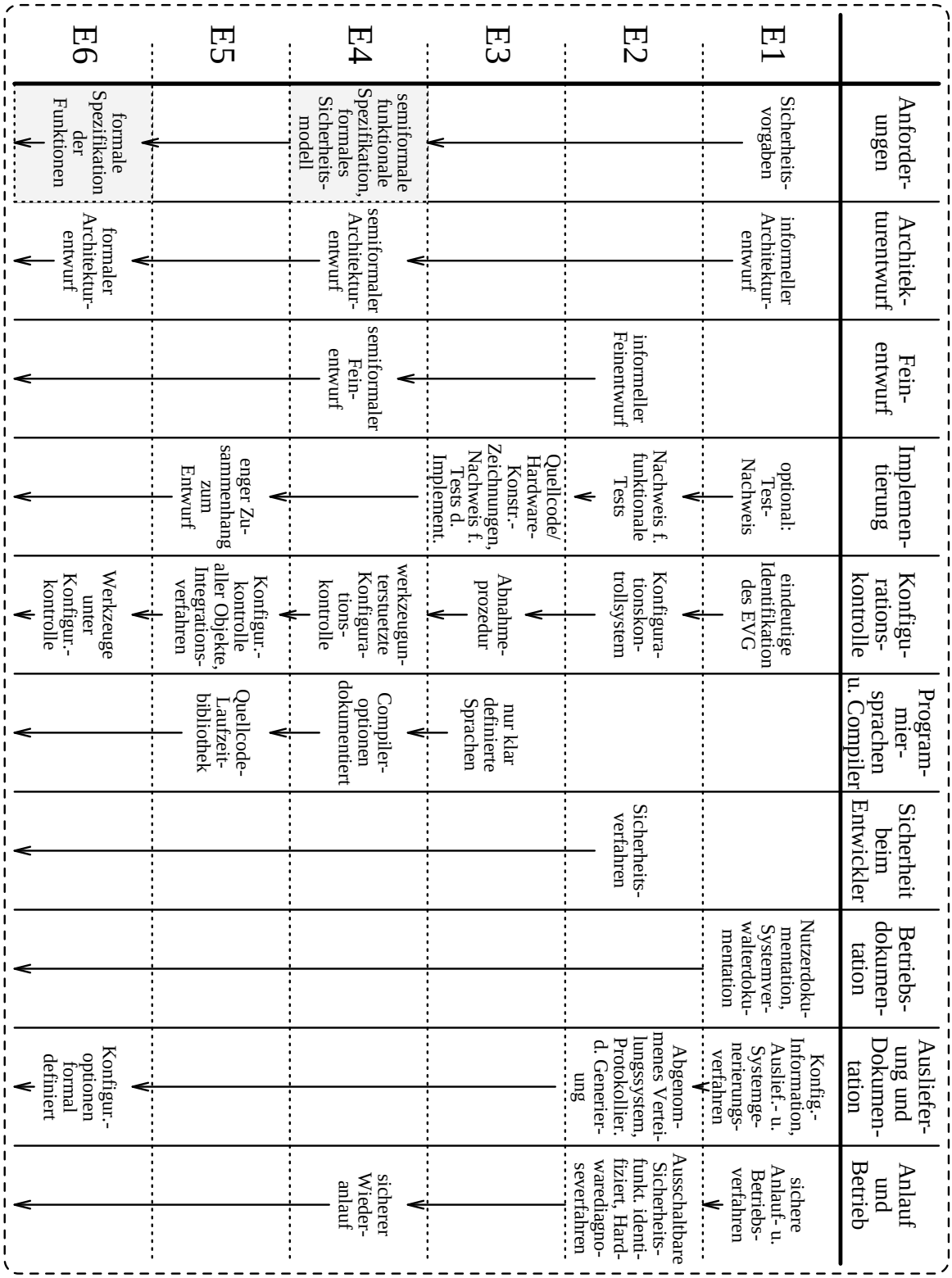

Abbildung 5.1: Übersicht zu den Evaluationsstufen

Dokumente (Quelle: [ITS91]). Die Stufe E0 ist nicht berücksichtigt. Sie wird dann vergeben, wenn ein Evaluationsgegenstand (EVG) nicht den Anforderungen der angestrebten Evaluationsstufe genügt, also bei der Evaluation durchfällt. Die vertikalen Pfeile in der Abbildung 5.1 sollen verdeutlichen, daß sich die Anforderungen einer Stufe natürlich auf die nachfolgenden höheren Stufen übertragen.
Aus der Abbildung ist deutlich zu entnehmen, daß ab der Evaluationsstufe E4 semiformale und formale Beschreibungsdokumente zum EVG unerläßlich sind. Beispiele für semiformale Beschreibungen (zur Systemaktivität WSAV und zu einem Ausschnitt des essentiellen Speichers des Systems) stellt diese Arbeit in Kapitel 2 vor, während in Kapitel 3 gezeigt wird, wie aus diesen semiformalen Beschreibungen eine formale (Anforderungs-) Spezifikation abgeleitet werden kann. Darin enthalten ist auch eine typische Aussage eines Sicherheitsmodells, die auf Seite 48 informal und auf Seite 75 formal beschrieben ist. Im folgenden Abschnitt wird diskutiert, wie die, innerhalb dieser Diplomarbeit geleisteten Arbeiten, in den hier skizzierten Rahmen zum Erhalt eines Gütezertifikats einzuordnen sind. Anmerkung: Dieser Überblick zu [ITS91] bezieht sich auf die Version 1.2 vom 28. Juni 1991. Als weitere Quelle diente [SB92]. In [ZSI90] wird die weitere Evaluationstufe E7 (formal verifiziert) beschrieben. Für diese Stufe gelten die gleichen strengen Anforderungen wie schon für E6. Zusätzlich müssen Unterlagen eingereicht werden, die u. a. formale Beweise zur Einhaltung der Sicherheitsanforderungen und der Korrektheit der Verfeinerungsschritte des Entwicklungsprozesses enthalten.

5.2 Einordnung der vorliegenden Arbeit

Im vorangegangenen Abschnitt wurde erläutert, welche umfassenden Dokumente und Maßnahmen vom Antragsteller erbracht werden müssen, um ein IT-System bezüglich einer hohen Wirksamkeits- und Korrektheitsstufe erfolgreich zu evaluieren. Diese Arbeit bezieht sich nur auf einen Teil der notwendigen Maßnahmen (semiformale und formale Beschreibungen eines Sicherheitsmodells bzw. essentieller Aktivitäten, siehe auch grau unterlegte Kästchen der Abbildung 5.1) und konzentriert sich zudem nur auf einen sehr kleinen Ausschnitt des Systems.

Diese Arbeit kann nicht ohne weiteres zum Erhalt eines Gütezertifikats für das SUNRISE-System verwendet werden. Sie versteht sich vielmehr als ein Beispiel zur semiformalen und formalen Spezifikation von Systemanforderungen. Außerdem zeigt sie an einem Beispiel, wie semiformale Bechreibungen als Grundlage formaler Spezifikationen verwendet und somit die in der Praxis entwickelten Software-Entwicklungsmethoden um einen formalen Aspekt ergänzt werden können. Nach dem, in dieser Arbeit durchgeführten, Vorgehen zur Analyse und Beschreibung von Systemanforderungen (informale Analyse – semiformale Beschreibung – formale Spezifikation), können prinzipiell auch die in Abbildung 5.1 geforderten Dokumente zum Architekturentwurf und Feinentwurf erstellt werden.
Die in Abschnitt 2.4 informal beschriebene und in Abschnitt 3.4.4 formal spezifizierte Sicherheitsaussage bezieht sich auf den Sicherheitsaspekt der Datenintegrität (siehe Seite 93). Den Sicherheitszielen (ebenfalls Seite 93) kann sie unter dem generischen

Oberbegriff Unverfälschtheit zugeordnet werden. Medizinischen Datenbanksystemen kommt den Sicherheitszielen, die unter diesen Oberbegriff fallen, eine besondere Bedeutung zu (siehe auch [CKL93]).
Zu weiteren Qualitätsaspekten der Abbildung 5.1 besteht kein Bezug.

Anmerkung: Es muß darauf hingewiesen werden, daß der Versuch, ein bereits konventionell entwickeltes System nachträglich um formale Beschreibungen bzw. eine formale Verifikation zu ergänzen, um dadurch eine Gütezertifizierung einer hohen Qualitätsstufe zu erwirken, natürlich wesentlich mehr Probleme aufwirft, als wenn von vorneherein formale Methoden bei der Systementwicklung angewendet werden. Eine Möglichkeit bestände natürlich darin, daß inzwischen gut analysierte SUNRISE-System unter Einbezug formaler Methoden komplett neuzuentwickeln. Diese Vorgehensweise wäre mit Sicherheit erfolgversprechender, als eine nachträgliche Anreicherung um formale Beschreibungen, die sich ständig an der aktuellen Implementierung orientieren müßte. Weil die Problemstellung mittlerweile recht präzise vorliegt, könnte bei einer Nachentwicklung womöglich auch ein strenges Top-Down-Vorgehen eingehalten werden. Vermutlich ist das SUNRISE-System — soll es einer vollstanndigen formalen Nachentwicklung unterzogen werden — zum Einstieg in das Gebiet der formalen Software-Entwicklung jedoch zu komplex.

Kapitel 6

Zusammenfassung und Anmerkungen

In diesem Kapitel wird zunächst die Diplomarbeit kurz zusammengefaßt. Anschließend werden weitere Anmerkungen zu den Themen *Systemanalyse*, *Obscure*, *Sunrise*, *Verifikation* und *Vorschlag für ein weiteres Vorgehen* aufgeführt.

6.1 Zusammenfassung der Arbeit

Kapitel 2 dieser Diplomarbeit beschäftigte sich mit der Ausarbeitung der logischen Essenz des SUNRISE-Systems. Als Grundlage diente die Methode der *strukturierten Systemanalyse* nach [MP88]. Anschließend erfolgte die Anwendung dieser Methode auf das SUNRISE-System. Dazu stand nur wenig Zeit zur Verfügung, so daß ein Gesamtblick auf das System nur ansatzweise möglich war und schnell eine Konzentration auf den Ausschnitt *duales Speicherkonzept* erfolgen mußte. Die Ergebnisse dieser Tätigkeit wurden in Kapitel 2 anhand von E/R- und DFD-Diagrammen dokumentiert.

Parallel zu den Systemanalyse-Tätigkeiten erfolgte eine Einarbeitung in das Thema *Sicherheits-Gütezertifikate*. Verschiedene Dokumente wurden dazu durchgearbeitet ([ITS91], [ZSI90], die DIN-Normen DIN 66285, DIN V VDE 0801/01.90, DIN ISO 9000, DIN IEC 65A und [SB92]) und daraufhin untersucht, ob diese bereits eine Einstufung von formal entwickelter Software vorsehen. Insbesondere der nationale bzw. europäische Kriterienkatalog [ITS91] fordert zur Vergabe von Sicherheits-Gütezertifikaten hoher Qualitätsstufen die Anwendung formaler Methoden. Einzelne Aspekte dieses Kriterienkatalogs wurden in Kapitel 5 skizziert und der Bezug zur vorliegenden Arbeit wurde verdeutlicht.

Im Anschluß an die erste Phase erfolgte die formale Spezifikation der Ergebnisse der Ausarbeitung der logischen Essenz in der algorithmischen Spezifikationssprache OBSCURE. Dazu baute diese Arbeit auf den Ergebnissen und Erfahrungen der Fallstudie KORSO-HDMS-A auf. Als Grundlage dienten insbesondere die Dokumente [NW93], [Het93], [Hec93], [Aut93] und [Ben93c]. Das in [Aut93] vorgestellte Schema zur Transformation von E/R-Diagrammen in OBSCURE-Spezifikationen wurde in

dieser Arbeit exemplarisch angewendet (allerdings *"von Hand"*). Dadurch konnte verdeutlicht werden, daß ein harmonischer Übergang von herkömmlicher Software-Entwicklung, mit informalen und semiformalen Beschreibungsmitteln, hin zur formalen Software-Entwicklung möglich ist. Es wurde darauf hingewiesen, daß dieser Übergang, zumindest teilweise, automatisch erfolgen kann. Die formale Spezifikation der logischen Essenz wurde in Kapitel 3 und im Anhang A präsentiert und erläutert.

Nach Fertigstellung der formalen Spezifikation wurde ein Rapid-Prototyping mit dem OBSCURE-Interpreter durchgeführt; ein Auszug hierzu wurde in Kapitel B vorgestellt. In Abschnitt 3.5 wurde dagegen versucht, diesen etwas schwer verständlichen Auszug durch graphische Darstellungen verständlicher zu beschreiben. Anhand des Rapid-Prototypings konnte nachträglich ein Fehler in der, zu diesem Zeitpunkt gegebenen, formalen Spezifikation aufgedeckt (es handelt sich um einen typischen Fehler: Nichtbeachtung eines Spezialfalls) und korrigiert werden. Zudem konnte das Rapid-Prototyping dazu beitragen, das Vertrauen in die Spezifikation des Ausschnitts *duales Speicherkonzept* zu vertiefen.

Die weitere Tätigkeit bestand darin, die Beziehungen des formal spezifizierten Ausschnitts der logischen Essenz zur derzeitigen Implementierung aufzuzeigen. Dazu wurde die aktuelle Implementierung zum Ausschnitt *duales Speicherkonzept* informal beschrieben und das Beispiel, das zuvor schon beim Rapid-Prototyping und zur Erläuterung des dualen Speicherkonzepts betrachtet wurde, auf die aktuelle Implementierung abgebildet. Aufbauend auf dieser Abbildung wurde versucht, einen möglichst präzisen Zusammenhang zwischen den Entity- und Relationshiptypen der Anforderungsspezifikation zu Komponenten der Implementierung zu beschreiben. Die Ergebnisse dieser Arbeitsphase wurden in Kapitel 4 dokumentiert.

Die letzte Arbeitsphase konzentrierte sich, neben dem Niederschreiben der Arbeit, auf die Diskussion einiger Fragen. Hierzu zählte u. a. die Beurteilung der Praxisrelevanz des OBSCURE-Systems und der Sprache OBSCURE, sowie die Diskussion des Nutzens der durchgeführten Fallstudie für das SUNRISE-System. Auf diese Themen gehen die folgenden Abschnitte ein.

6.2 Anmerkungen zur Systemanalyse

Um eine Anforderungspezifikation zum SUNRISE-System zu erstellen, mußte zuvor eine Systemanalyse durchgeführt werden. Hierzu existierten am Lehrstuhl von Prof. Dr.-Ing. Loeckx bisher keine Erfahrungen, so daß die Beschäftigung mit dieser ersten Phase sich schwieriger gestaltete als zunächst angenommen. Obwohl zu diesem Thema auf einschlägige Literatur zurückgegriffen werden konnte (z. B. [MP88] und [You89]) und das IBMT[1] alle verfügbaren Dokumente bereitstellte, sowie fruchtbare Diskussionen mit den Systementwicklern möglich waren, besteht doch eine leichte Unsicherheit bezüglich der Qualität der Ergebnisse dieser ersten Phase. Die wichtigste Ursache hierfür ist der enge Zeitraum in dem die Systemanalyse durchgeführt

[1] Fraunhofer-Institut Biomedizinische Technik, St. Ingbert.

wurde. Es standen insgesamt nur wenige Wochen für diese Phase zur Verfügung, an deren Ende ein akzeptabler Ausgangspunkt für die weitere Arbeit unbedingt erreicht sein mußte. Die Analysetätigkeit konnte deshalb nur bedingt mit dem Blick auf das gesamte SUNRISE-System erfolgen und mußte sich sehr schnell auf den Ausschnitt *duales Speicherkonzept* konzentrieren.

Auch auf das in Abschnitt 2.2.6 bereits näher erläuterte Problem, daß die logische Essenz des Systems nicht alleine aus den informalen Systembeschreibungen abgeleitet werden konnte, sondern auch die aktuelle Implementierung zu Rate gezogen werden mußte, soll nochmals hingewiesen werden.

Ebenfalls als problematisch hat sich die Beschäftigung als einzelne Person mit der Systemanalyse erwiesen. Weil dies nicht nur eine komplexe, sondern hinsichtlich der Beschreibung der Ergebnisse auch sehr kreative Tätigkeit ist, sind Diskussionen zwischen mehreren Personen, die mit der Systemanalyse beschäftigt sind, sehr zu empfehlen. Glücklicherweise konnte hierzu auf die beiden Betreuer dieser Arbeit, Roland Brill und Ramses A. Heckler zurückgegriffen werden, die, obgleich nicht selbst mit Analysetätigkeiten beschäftigt, sich in die Problematik hineinversetzten und als notwendige Diskussionspartner fungierten.

Triviales Fazit: Eine Systemanalyse sollte möglichst nicht von einer Person allein und nicht unter zu großem Zeitdruck durchgeführt werden.

6.3 Anmerkungen zu OBSCURE

Das OBSCURE-System wurde nicht für den praktischen Einsatz in größerem Rahmen entwickelt. So ist es auch nicht verwunderlich, wenn die Praxisrelevanz von OBSCURE nach den Erfahrungen dieser Arbeit als nicht ganz befriedigend eingestuft werden muß. Im folgenden werden die Schwächen und Kritikpunkte, die in dieser Arbeit deutlich wurden, unter den Gesichtspunkten OBSCURE-System, Spezifikationssprache OBSCURE und Anbindung an andere Systeme diskutiert.

- **OBSCURE-System**
  - **OBSCURE-Moduldatenbank**
    Die Leistungsgrenze der OBSCURE-Moduldatenbank wurde durch diese Spezifikation erreicht. Ohne die FORGET-Konstruktion im Modul DATENMODELLEBENE (siehe Seite 149) kann die Signatur dieses Moduls nicht mehr in der Moduldatenbank des OBSCURE-Systems abgelegt werden.

    Ein Abweichen vom Schema [Aut93] zur Transformation von E/R-Diagrammen in OBSCURE-Spezifikationen war alleine schon aus diesem Grund notwendig. Hätte diese Arbeit sich nicht im wesentlichen auf die zur Spezifikation der Operation WSAV speziell benötigten Sorten und Operationen konzentriert, so wäre die Leistungsgrenze der OBSCURE-Datenbank deutlich überschritten worden. Auch konnte die Modularisierung nicht immer analog zu [Aut93] erfolgen. Beispiel ist der in [Aut93]

vorgeschlagene Modul DATENBANKEBENE. Ein Ablegen dieses Moduls in der OBSCURE-Datenbank war nicht möglich, so daß auf diesen Modul verzichtet werden mußte.

- **OBSCURE-Interpreter**
  Die Operation WSAV der Anforderungspezifikation ist algorithmisch nicht besonders anspruchsvoll. Der OBSCURE-Interpreter schaffte die Auswertung deshalb auch recht mühelos. Trotzdem ist zu bezweifeln — dies haben insbesondere die Interpretersitzungen zur Fallstudie [ABH92] angedeutet, daß der Interpreter komplexere Operationen des SUNRISE-Systems, wie z. B. die Anwendung einer Macro-Bearbeitungsoperation[2] auf ein Datenobjekt, auswerten kann. Dieses Problem kann allerdings durch Verwendung des Übersetzers von OBSCURE nach ML umgangen werden. Es kann also auch ein Rapid-Prototyping mit dem Interpreter zu ML durchgeführt werden.

  Außerdem ist zu überlegen, ob nicht der Interpreter durch weitere Optionen ergänzt werden könnte, um ein besseres Nachvollziehen von Interpretersitzungen zu ermöglichen. Eine Möglichkeit bestände darin, eine intuitive Namensvergabe für nichtssagende Kontruktorterme[3] zu ermöglichen. Anstelle der Konstruktorterme könnte der Interpreter diese Namen ausgeben (sofern welche definiert sind) und nur auf besonderen Wunsch des Anwenders hin, die Namen durch die ihnen zugeordneten Konstruktorterme ersetzen. Eine solche Option deutet diese Arbeit auf Seite 167 an, indem sie intuitivere Namen für die unleserlichen Konstruktorterme der Attributsorten vergibt.

- **Spezifikationssprache** OBSCURE

  - **Abstraktheit und Losheit**[4]
    Es stellt sich die Frage, inwiefern die algorithmische Spezifikationssprache OBSCURE zur Spezifikation der logischen Essenz eines Systems unter dem Aspekt der erforderlichen Abstraktheit geeignet ist.
    Aufbauend auf den Ergebnissen dieser Arbeit kann diese Frage nur eingeschränkt positiv beantwortet werden: Die vorliegende algorithmische Spezifikation beinhaltet zwangsläufig viele Details, die nicht zur logischen Essenz eines Systems gehören. Dabei soll eine logische Essenz eigentlich von sämtlichen Implementierungsdetails abstrahieren und sich nur auf den

[2] Eine Macro-Bearbeitungsoperation ist eine benutzerspezifische Datenbearbeitungsoperation, die hintereinander mehrere Bearbeitungsoperationen auf ein Datenobjekt anwendet.

[3] In diesem Fallbeispiel sind die wenig aussagekräftigen Konstruktorterme Folge vieler unterschiedlicher Instanzen paramatrisierter Module (siehe z. B. Modul ATTRIBUTSORTEN auf Seite 116). Die bei der Instanziierung stattfindende Namensumbenennung, wird beim Auswerten von Termen in der Interpretersitzung wieder rückgangig gemacht.

[4] Beim losen Spezifizieren schränkt man sich nicht von vorneherein auf nur ein Modell ein. Allerdings verliert beim losen Spezifizieren die Möglichkeit zur Durchführung eines Rapid-Prototypings. Nähere Erläuterungen zu diesem Begriff werden z. B. in der Sprachbeschreibung zur Spezifikationssprache SPECTRUM vorgestellt (siehe [BFG+91]).

Aspekt '*Was tut ein System*' konzentrieren. In extremer Form läßt sich diese Forderung jedoch — unabhängig von der verwendeten Spezifikationssprache — kaum erfüllen. Denn eine Beschreibung eines essentiellen Speichers, auf der Basis zu Entitytypen klassifizierter Einzelinformationen, enthält in strengem Sinne bereits Details, die für eine nachfolgende Implementierung nicht verbindlich sein sollen (siehe Seite 28). Diese Details würden sich auch in einer axiomatischen Spezifikation wiederfinden, die auf E/R-Diagrammen aufbaut. Allerdings werden durch den Einsatz der algorithmischen Spezifikationsmethode weitere Detailentscheidungen notwendig (hier z. B. die Wahl konkreter Algorithmen in den Operationen und Hilfsoperationen im Modul WSAV, sowie die Festlegung auf eine Ausführungsreihenfolge von Unteraktivitäten, die prinzipiell parallel ausgeführt werden können), die beim losen axiomatischen Spezifizieren teilweise vermieden werden könnten. Im Hinblick auf spätere Entwurfsphasen stellt sich also besonders für algorithmische Spezifikationen das Problem, welche Aspekte der Spezifikation als essentielle Systemanforderungen aufgefaßt (z. B. Stelligkeit und Ein-/Ausgabeverhalten essentieller Operationen) und welche als Beschreibungsdetails angesehen werden sollen.
(Ein Beispiel für eine axomatische Spezifikation der logischen Essenz eines Systems enthält [SNM+93].)

- **höhere Ordnung**[5]
  Die Notwendigkeit oder der Wunsch zur Verwendung von Funktionen (bzw. Operationen) höherer Ordnung ist in dieser Fallstudie nicht aufgetreten. Diese Arbeit beschränkt sich aber auf einen kleinen Ausschnitt der logischen Essenz eines Systems. Insbesondere, das in [Aut93] auf Seite 20 ff. und in [Hec93] auf Seite 52 angesprochenene Problem der Schlüsselgenerierung wird durch den hier betrachteten Ausschnitt nicht berührt. Ebenfalls nicht behandelt wird in dieser Arbeit die Spezifikation eines Kommunikationsmodels der essentiellen Aktivitäten des Systems. [NW93] stellt u. a. ein solches Kommunikationsmodell vor und verwendet dazu Funktionen höherer Ordnung. Die Notwendigkeit und Vorteile der Verwendung höherer Ordnung ist sowohl im Hinblick auf eine Schlüsselgenerierung, als auch für ein Kommunikationsmodell essentieller Aktivitäten, nicht erschöpfend diskutiert.

[5] Eine Funktion (Operation) höherer Ordnung, ist eine Funktion deren Argumente oder Resultate selbst wieder Funktionen sein können. Nähere Erläuterungen zu diesem Begriff werden z. B. in der Sprachbeschreibung zur Spezifikationssprache SPECTRUM vorgestellt (siehe [BFG+91]).

- **Polymorphie**[6]
  Durch die Verwendung polymorpher Operationen hätte die vorliegende Spezifikation an mehreren Stellen kürzer und übersichtlicher formuliert werden können. Das folgende Beispiel soll dies stellvertretend belegen:
  Im Vergleich zu [Het93] — dort wird Polymorphie verwendet — erfolgt in dieser Spezifikation eine recht umständliche Anreicherung der Domainsorten um je eine Konstante, die den leeren Attributeintrag repräsentiert: Im Modul ATTRIBUTSORTEN muß der parametrisierte Modul ATTR auf sämtliche Domainsorten angewendet, d. h. mit sämtlichen Domainsortensorten instantiiert werden. In [Het93] wird diese lästige Instantiierung durch die Einführung polymomorpher Operationen vermieden. Die Verwendung von Polymorphie ist also wünschenswert, weil dies zu knapperen und übersichtlicheren Spezifikationen beitragen kann.

- **Sortenclashproblem**
  Das am Lehrstuhl von Prof. Dr.-Ing. Loeckx bereits ausführlich diskutierte Sortenclashproblem stellt sich auch in dieser Arbeit (siehe auch Seite 129). Die Behelfslösung *Dummy-Methode* wird aber nicht angewendet, weil dies ein größeres Abweichen von der vorgegebenen Strukturierung nach dem Transformationsschema [Aut93] voraussetzen würde.

- **Anbindung an andere Systeme**
  Die Anbindung an das Beweissystem INKA (siehe [BHHW86]) ist bisher nur ansatzweise realisiert und befindet sich noch in der Testphase. Insbesondere folgendes Problem ist bisher nicht gelöst: Bevor eine modularisierte Spezifikation nach INKA übersetzt werden kann, ist z. Zt. eine Transformation (engl. *flattening*) der Spezifikation in eine nicht weiter strukturierte — aus einem einzigen Modul bestehende — atomare Spezifikation notwendig. Ein solches *flattening* ist mit erheblichem Zeit-/ und Nervenaufwand verbunden, wenn es von Hand durchgeführt wird und sollte deshalb automatisch erfolgen. Zur Zeit ist ein automatisches *flattening* jedoch nicht möglich, weil ein entsprechendes System bisher nicht realisiert wurde.

  Soll das OBSCURE-System weiterhin in größeren Fallstudien als formale Spezifikationssprache verwendet werden, wäre es sinnvoll ein System zu erstellen, daß die automatische Transformation von semiformalen Beschreibungen (z. B. aufbauend auf E/R- und DFD-Diagrammen) in OBSCURE-Spezifikationen realisiert bzw. unterstützt.

[6] Eine Polymorphe Operation wird gleichzeitig über mehreren Sorten definiert. Dadurch erspart man sich eine lästige wiederholte Definition dieser Operation für mehrere einzelne Sorten (z. B. Operation "+" gleichzeitig einführen für natürliche und ganze Zahlen). Nähere Erläuterungen zu diesem Begriff werden z. B. in der Sprachbeschreibung zur Spezifikationssprache SPECTRUM vorgestellt (siehe [BFG+91]).

6.4 Anmerkungen zu SUNRISE

Diese Arbeit hat angedeutet, welche umfangreichen Maßnahmen notwendig sind, um eine Sicherheits-Güteklassifizierung einer hohen Qualitätsstufe für das SUNRISE-System zu erwirken. Eine formale Verifikation wurde dabei noch ausgeklammert. Weitere Aspekte, wie z. B. die betriebliche Organisation wurden ebenfalls nicht angesprochen, sind aber zum Erhalt eines Sicherheits-Gütezertifikats ebenso wichtig.

Wie viele andere Softwareprojekte in der Industrie, ist auch das SUNRISE-Projekt mit einem engen Entwicklungszeitraum und einer dünnen Personalbesetzung konfrontiert. Diese Ausgangssituation macht eine Software-Entwicklung nach der hier angedeuteten formalen Methode, mit dem Ziel ein hohes Maß an Sicherheit zu erreichen, nahezu unmöglich.

Formale Software-Entwicklung hat gegenüber einem üblichen Vorgehen in der Industrie den entscheidenden Nachteil, daß sehr viel Zeit für die ersten Phasen aufgewendet werden muß — dies belegt auch diese Arbeit — und somit erst sehr spät eine konkrete Implementierung in einer Programmiersprache erfolgen kann. Ohne ein Umdenken der Auftraggeber, die möglichst frühzeitig eine erste Version ihres Programms erhalten möchten, und auch der Software-Entwickler selbst, ist ein formales Vorgehen kaum durchführbar. Auch hinsichtlich der Qualifizierung der Softwareentwickler stellen sich zusätzliche Anforderungen. Es kann einem System-Entwickler wohl kaum zugemutet werden, sich in kurzer Zeit — womöglich neben weiteren Tätigkeiten — in die Thematik der formalen Software-Entwicklung einzuarbeiten. Auch existieren zur Zeit noch recht wenige komplette Fallstudien (von der Anforderungsspezifkation bis hin zur Implementierung und mit formaler Verifikation) zu diesem Thema, die als Vorlage dienen könnten.

Für das SUNRISE-System ergibt sich die Folgerung, daß es zum Erhalt eines Gütezertifikats nach [ITS91] der Qualitätsstufe E6 einer Vielzahl weiterer Maßnahmen und Dokumente bedarf, deren Realisierung sehr viel Zeit und/oder weitere Mitarbeiter erfordern würde.

Außerdem wurde in der Anmerkung auf Seite 97 darauf hingewiesen, daß eine nachträgliche Anreicherung eines konventionell entwickelten Systems mit formalen Beschreibungen und evtl. einer formalen Verifikation natürlich sehr problematisch ist. Im Hinblick auf den Erhalt eines Sicherheits-Gütezertifkats ist es sehr zu empfehlen, bei einer Software-Entwicklung vom vorneherein den Einsatz formaler Methoden vorzusehen.

Sinnvoll wäre eine Anfertigung semiformaler bzw. formaler Systembeschreibungen zu abstrakteren Entwicklungsebenen des SUNRISE-Systems aber nicht nur im Hinblick auf den Erhalt eines Sicherheit-Gütezertifikats, sondern auch hinsichtlich der späteren Wartung des Systems, sowie zur Einarbeitung von Systemverwaltern und auch Anwendern:

- Vorteil für die Wartung des SUNRISE-Systems
  Bei Veränderungen einzelner Systemkomponenten oder zum Integrieren zusätzlicher Funktionen zu einem späteren Zeitpunkt sind abstrakte Beschreibungen des Systems von sehr großem Nutzen. Vor allem wenn die

ursprünglichen Systementwickler nicht mehr zur Verfügung stehen, sind solche Maßnahmen oft sehr problematisch. Abstrakte und leicht verständliche Systembeschreibungen können in solchen Fällen wesentlich zum Verständnis der Systemarchitektur und letztlich auch der Implementierung beitragen. Dadurch kann möglicherweise ein *herumbasteln* an der Software durch ein *solides Ändern* ersetzt werden. Natürlich müssen dann auch die abstrakten Systembeschreibungen entsprechend angepaßt werden.

- Einarbeitung von Systemverwaltern
  Systemverwalter sollten nicht nur die ihnen zugedachten Verwaltungsaufgaben übernehmen, sondern auch das System grundsätzlich verstehen, um bei Problemfällen zumindest als kompetente Ansprechpartner für die Systementwickler zu fungieren. Zur Einarbeitung in das System und insbesondere zum Verständnis der Systemarchitektur, sind abstrakte Beschreibungen sehr sinnvoll. Der Einarbeitende kann anhand dieser Dokumente den gesamten Entwicklungsprozess nachvollziehen und lernt somit zwischen wesentlichen Aspekten des Systems und mehr oder weniger groben Implementierungsdetails zu unterscheiden. Auf diese Weise kann er sich ein tiefes Verständnis zum System und dessen Entwicklungsgeschichte erarbeiten.

6.5 Anmerkungen zum Thema Verifikation

In der Aufgabenstellung zur Diplomarbeit war auch die Durchführung eines formalen Beweises vorgesehen. Eine Aussage zur Sicherheit des SUNRISE-Systems wurde dazu in Abschnitt 2.4 informal erläutert und später auch formal spezifiziert. Die Gültigkeit dieser Aussage bzgl. des spezifizierten Systemausschnitts kann prinzipiell formal verifiziert werden. Dies war aber innerhalb dieser Arbeit nicht möglich. Die Gründe hierfür werden im folgenden erläutert.

Es werden Möglichkeiten vorgestellt, nach denen eine formale Verifikation hätte erfolgen können, und es wird gleichzeitig angegeben warum diese Möglichkeiten nicht angewendet werden konnten:

- Die erste Möglichkeit betrifft die Verwendung eines Kalküls über strukturierten OBSCURE-Spezifikationen[7]. Damit könnte eine formale Verifikation von Hand (oder mit einem System) relativ elegant erfolgen. Ein solcher Kalkül wurde bisher nicht entwickelt (und natürlich nicht implementiert). Eine formale Verifikation nach dieser Möglichkeit konnte also nicht erfolgen.

- Die strukturierte OBSCURE-Spezifikation zu SUNRISE hätte in eine atomare OBSCURE-Spezifikation transformiert werden können (engl. *flattening*), um dann die Aussage bzgl. der atomaren Spezifikation zu verifizieren. Zwar stellt

[7] Strukturierte OBSCURE-Spezifikationen sind aus mehreren atomaren Modulen aufgebaut, die durch die Sprachkonstrukte (z. B. PLUS, COMPOSE) von OBSCURE miteinander verknüpft sind. Im Gegensatz zu strukturierten Spezifikationen bestehen atomare OBSCURE-Spezifikationen nur aus einem einzigen Modul. Nähere Erläuterungen zu den Begriffen finden sich in [LL93].

[Tre91] einen Kalkül für atomare algorithmische Spezifikationen vor, doch kann dieser nicht ohne vorherige Anpassung an die aktuelle Version von OBSCURE eingesetzt werden. Wichtiger ist aber, daß bereits die Transformation einer strukturierten OBSCURE-Spezifikation in eine atomare für große Fallbeispiele nur unter erheblichem Zeitaufwand durchgeführt werden kann und zudem zu einer sehr unübersichtlichen atomaren Spezifikation führt. Wegen der Unübersichtlichkeit schleichen sich zudem leicht Fehler ein. Ein System, daß eine solche Transformation automatisch durchführt und damit die erwähnten Probleme lösen würde, wurde bisher nicht implementiert.

- Die Anbindung des automatischen Beweissystems INKA (siehe [BHHW86]) an das OBSCURE-System ist bereits teilweise realisiert, befindet sich aber noch in der Testphase. Allerdings ist auch hier eine Transformation der strukturierten OBSCURE-Spezifikation in eine atomare notwendig, bevor eine Übersetzung in die INKA-Eingabesprache erfolgen kann. Die oben bereits erwähnten Probleme hinsichtlich der erforderlichen Transformation gelten auch hier.

Weil zur Behebung oder Überwindung der geschilderten Probleme zu wenig Zeit existierte und dies auch den Rahmen dieser Diplomarbeit überschritten hätte, wurde keine formale Verifikation durchgeführt.

6.6 Vorschlag für eine Weiterführung der Fallstudie

Diese Arbeit kann zumindest in zwei Richtungen fortgesetzt werden: Die erste Möglichkeit ist, aus der hier vorgestellten Spezifikation eine Entwurfsspezifikation abzuleiten und diesen Verfeinerungsprozeß weiter in Richtung der bestehenden oder einer neuen Implementierung fortzusetzen. Die zweite Möglichkeit betrifft den Aspekt der formalen Verifikation. Die Gültigkeit der formalen Sicherheitsaussage und die Einhaltung der Integritätsbedingungen könnte, nach Behebung der geschilderten Probleme, mit dem INKA-System oder einem anderen Beweissystem formal bewiesen werden. Später könnte auch der Versuch unternommen werden, die Korrektheit des Verfeinerungsprozeß zu verifizieren.

Es soll nun untersucht werden, ob eine Fortsetzung dieser Diplomarbeit überhaupt sinnvoll ist. Folgende Argumente sprechen für eine Fortsetzung dieses Fallbeispiels zur formalen Software-Entwicklung:

1. Bezüglich der Entwurfsphasen und dem Übergang zu einer Implementierung existieren auch nach dieser Diplomarbeit kaum Erfahrungen zur Praxisrelevanz der Sprache OBSCURE und des OBSCURE-Systems.

2. Hinsichtlich der formalen Verifikation von Sicherheitsaussagen und der Korrektheit von Verfeinerungsschritten speziell im Hinblick auf OBSCURE-Spezifikationen, konnte diese Arbeit keine Erkenntnisse beisteuern.

3. Das SUNRISE-System ist ein interessanter Untersuchungsgegenstand, weil es zur Verwaltung essentieller System-Informationen letztlich auf ein anderes

System — das UNIX-Dateisystem — zurückgreift. Speziell diese Detailentscheidung könnte einer näheren Untersuchung unterzogen werden.

4. Durch eine Fortführung der Fallstudie könnte dem IBMT auch ein einführendes Verständnis zu den weiteren Phasen der formalen Software-Entwicklung vermittelt werden. Diese Arbeit spricht nur den ersten Schritt einer formalen Software-Entwicklung an.

Die wesentlichen Gegenargumente einer Fortsetzung werden durch die folgenden Stichpunkte beschrieben:

1. Die Fortführung dieser Fallstudie kann nicht zum direkten Erwerb eines Softwaregütesiegels beitragen. Zum einen sind dazu weitere Maßnahmen notwendig, die z. B. die betriebliche Organisation des IBMT betreffen, und zum anderen wird durch diese Fallstudie nur ein kleiner Ausschnitt des gesamten Systems einer formalen Analyse unterzogen. Eine Ausweitung auf das gesamte System wäre mit einem wesentlich höheren Aufwand verbunden.

2. Nachdem die Leistungsgrenze des OBSCURE-Systems bereits durch diese (kleine) Anforderungsspezifikation tangiert wurde, erscheint eine Weiterführung des Fallbeispiels auf der Basis des bestehenden OBSCURE-Systems als wenig erfolgversprechend. In Abschnitt 6.3 wurden die wichtigsten Probleme angesprochen.

3. Die Anbindung an ein Verifikationssystem ist bisher nur teilweise realisiert (zur Übersetzung in die INKA-Eingabesprache fehlt z. B. ein System zum automatischen *flattening* einer großen strukturierten Spezifikation). Ein eigener Kalkül für OBSCURE, der auch Beweise über strukturierten Spezifikationen ermöglichen könnte, wurde bisher nicht entwickelt.

4. Während die Durchführung von formalen Beweisen normalerweise keiner Unterstützung durch das IBMT bedarf, muß zur Spezifikation der Entwurfsebenen damit gerechnet werden, daß intensive Diskussionen mit den Systementwicklern notwendig sind. Der Zeitaufwand für das IBMT würde sich also keineswegs verringern.

Die Gegenargumente zeigen, daß einer Fortsetzung dieser Arbeit — sei es im Hinblick auf die Spezifikation von Entwurfsebenen oder auf die Durchführung einer formalen Verifikation — auf jeden Fall eine Weiterentwicklung des OBSCURE-Systems (bzw. der Sprache OBSCURE) vorausgehen sollte. Diese Frage sollte deshalb zunächst geklärt werden, bevor eine Entscheidung zur Fortführung dieser Fallstudie getroffen werden kann.

Anhang

Anhang A

Die Spezifikation der Datenmodellebene

In diesem Kapitel wird die formale Spezifikation der Datenmodellebene zum Ausschnitt **duales Speicherkonzept** vorgestellt. Wie bereits in Abschnitt 3.3 angesprochen, wird diese Spezifikation erstellt durch Anwendung des Transformationsschemas [Aut93] von E/R-Diagrammen in OBSCURE-Spezifikationen *"von Hand"* auf das E/R-Diagramm in Abbildung 2.8 und die dazugehörenden Attribut-Diagramme in den Abbildungen 2.10 bis 2.16. Leichte Abweichungen vom Schema [Aut93] wurden dabei aus folgenden Gründen notwendig:

- Die OBSCURE-Datenbank war nicht in der Lage, den Modul DATENBANKEBENE (siehe Seite 149) aufzunehmen. Eine Umstruktierung mußte hier also aus technischen Gründen erfolgen.
- [Aut93] sieht eine Unterscheidung zwischen abstrakten und konkreten Schlüsseln vor. Weil in dieser Fallstudie Schlüssel nur von geringer Bedeutung sind und sich das Problem der Schlüsselgenerierung überhaupt nicht stellt, wird von dieser Unterscheidung abgesehen. Hier werden, gemäß der Terminologie von [Aut93], nur konkrete Schlüssel betrachtet. Auch zur Spezifikation konkreter Schlüssel weicht diese Spezifikation ein wenig von [Aut93] ab.
- Das Schema [Aut93], das in dieser Fallstudie zum erstenmal angewendet wird, enthält selbst noch kleinere Fehler. Auch aus diesem Grund ist geringfügiges Abweichen an einigen Stellen unvermeidbar.
- Im Vergleich zu [Aut93] wurden einige Namensänderungen vorgenommen. Dies betrifft Sorten-, Operations- und Modulnamen. Ziel war es in diesen Fällen, kürzere oder intuitivere Namen zu wählen.

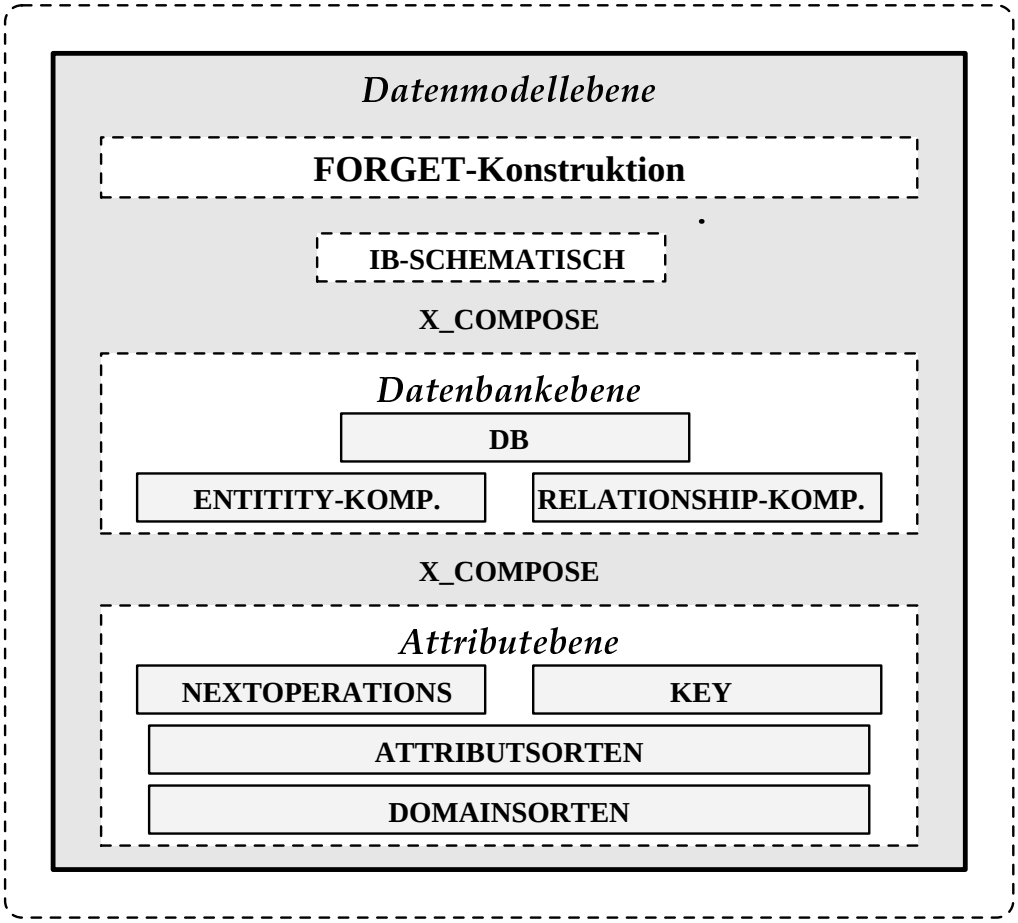

Abbildung A.1: Struktur der Spezifikation zur Datenmodellebene

A.1 Struktur der Spezifikation

Einen Überblick zur Strukturierung der Spezifikation der Datenmodellebene vermittelt die Abbildung A.1. Außerdem enthält Anhang C eine vollständige Übersicht zur Modulstruktur der Gesamtspezifikation (Aktivitäten- und Datenmodellebene). Die Attributebene dient der Spezifikation der Attributsorten (Modul ATTRIBUTSORTEN). Dazu werden zunächst die benötigten Domainsorten (Modul DOMAINSORTEN) spezifiziert. Die Module KEY und NEXTOPERATIONS stehen im Zusammenhang zur Spezifikation von Schlüsseln. In KEY werden spezielle Schlüsselsorten eingeführt und in NEXTOPERATIONS wird die, in dieser Fallstudie nicht wichtige, Generierung neuer Schlüsselwerte behandelt. In der Datenbankebene werden zunächst Entity- und Relationship-Komponenten (Module ENTITY-KOMPONENTE und RELATIONSHIP-KOMPONENTE) spezifiziert, die die Grundlage zur Spezifikation der Datenbanksorte bilden. Durch den Modul DB wird dann die Datenbanksorte *Db* selbst eigeführt und ferner werden

einige Operationen bereitgestellt, die einen abstrakten Umgang mit dieser Datenbanksorte ermöglichen sollen. Anschließend werden die schematischen Integritätsbedingungen im Modul IB-SCHEMATISCH spezifiziert. Diese werden dort zu einem Prädikat OK-schematisch zusammengefaßt. Zuletzt werden viele, nach außen nicht notwendigerweise zu exportierende, Sorten und Operationen *vergessen* (FORGET-Konstruktion).

A.2 Spezifikation der Attributebene

Die Spezifikation der Attributebene setzt sich zusammen aus den Modulen DOMAINSORTEN, ATTRIBUTSORTEN, NEXTOPERATIONS und KEY. Diese Module werden, mit Ausnahme von NEXTOPERATIONS, in den folgenden Unterabschnitten erläutert. In NEXTOPERATIONS wird die Generierung neuer Schlüsselwerte spezifiziert. In dieser Arbeit stellt sich das Problem der Schlüsselgenerierung nicht. Aus Gründen der Vollständigkeit wird der Modul NEXTOPERATIONS zwar in die Gesamtspezifikation aufgenommen, wegen seiner nebensächlichen Bedeutung hier aber nicht erläutert. Zunächst wird der Modul ATTRIBUTEBENE vorgestellt.

Modul: ATTRIBUTEBENE
Autor: cb *Datum: 15. Dezember 93*
Inhalt: Zusammensetzen der Bestandteile der Attributebene.

(**INCLUDE** NEXTOPERATIONS **PLUS INCLUDE** KEY)
X_COMPOSE
INCLUDE ATTRIBUTSORTEN
X_COMPOSE
INCLUDE DOMAINSORTEN

Ende Modul ATTRIBUTEBENE

A.2.1 Domainsorten

Domainsorten bilden die Grundlage zur Spezifikation der Attributsorten. Beispiel für eine Domainsorte ist eine Sorte *Datum*; sie wird verwendet zur Spezifikation einer Attributsorte *A-Datum*. Der Unterschied zwischen Domainsorten und Attributsorten wird in Abschnitt A.2.2 erläutert.

Hinsichtlich des Ausschnitts **duales Speicherkonzepts** sind insgesamt folgende Domainsorten zu spezifizieren:
Name, *Addresse*, *Datum*, *Kennung*, *DatenId*, *Parameterliste*, *History*, *Binaerdaten*, *Roi*, *AufenthaltsId*, *Familienstand*, *Nationalitaet*, *Beruf*, *Religion*, *Diagnose*, *Aufenthaltsinfo*, *CreationInfo*, *PatId*, *Ort*, *Geschlecht*, *HospitalId*, *RootId*.

Zur Beschreibung der Anforderungen an die Systemaktivität WSAV ist aber eine detaillierte Beschreibung der Domainsorten nicht erforderlich. Im Spezifikationstext zu WSAV treten nur einige Domainsorten zu Schlüsselattributsorten und die Domainsorte *Name* auf. Für diese Sorten genügt es aber zu fordern, daß unendlich viele Träger und eine Gleichheitsoperation existieren. Wie die Sorten konkret spezifiziert sind, ist für die essentielle Aktivität WSAV nicht von Bedeutung [1]. Aus diesem Grund und zur Reduzierung des Umfangs, wäre es an dieser Stelle sinnvoll, auf eine konkrete Spezifikation der Domainsorten zu verzichten, und diese (im Sinne eines losen Spezifizierens) zu importieren und gegebenenfalls durch Importaxiome einzuschränken.

Dies ist zwar prinzipiell möglich, führt jedoch dazu, daß aufgrund des nichtleeren Imports der resultierenden Gesamtspezifikation kein Rapid-Prototyping mit dem OBSCURE-Interpreter möglich ist. Hierzu ist ein leerer Import der Gesamtspezifikation unbedingt erforderlich. Weil eine ausführliche Spezifikation aller Domainsorten mit großem Aufwand verbunden ist, wendet diese Arbeit einen Trick an, mit dem dafür gesorgt werden kann, daß ein leerer Import der Gesamtspezifikation erreicht, aber dennoch nicht zuviel unnötige Arbeit und Zeit für die Spezifikation der Domainsorten aufgewendet wird: Alle Domainsorten werden durch eine Umbenennung der ganzen Zahlen (deren Spezifikation ist im OBSCURE-System vorgegeben) spezifiziert. Dadurch wird zwar für alle Domainsorten gefordert, daß unendlich viele Träger existieren (was man z. B. durch Quotientenbildung aber korrigieren könnte), doch soll dies zunächst nicht weiter stören. Wichtig ist aber, daß man sich dieses Tricks bewußt ist und für die späteren Entwurfsphasen hieraus keine falschen Anforderungen ableitet bzw. bei der Generierung von Beweisverpflichtungen vorsichtig ist.

Die Abbildung von ganzen Zahlen (Träger der Sorte *integer*) auf Träger einer Domainsorte wird in dem parametrisierten Modul DS spezifiziert, der nun vorgestellt wird.

Anmerkung: Eine Umbenennung der Sorte *integer* mit dem I_RENAME- oder E_RENAME-Sprachkonstrukt von OBSCURE ist nicht möglich.

Modul: DS
Autor: cb *Datum: 10. Dezember 93*
Inhalt: Parametrisierter Modul zur Spezifikation von Domainsorten.

CREATE
SORTS
Domainsort
OPNS
d _ : integer → Domainsort
d-r _: Domainsort → integer
_ kleiner _: Domainsort Domainsort → bool

[1] Man beachte, daß dies speziell für die essentielle Aktivität WSAV gilt und sich nicht unbedingt auf andere Systemaktivitäten überträgt.

```
        _ = _ : Domainsort Domainsort → bool
SEMANTICS
 CONSTRS
        d _ : integer → Domainsort
 VARS
        x1, x2: Domainsort
        i1, i2: integer
 PROGRAMS
        d-r x1            ←  CASE x1 OF
                                  d i1: i1
                             ESAC ;
        x1 kleiner x2     ←  d-r x1  < d-r x2;
ENDCREATE
PARAMS [ SORTS Domainsort
         OPNS d _**,d-r _** ]
```

Ende Modul DS

Anwendung findet der Modul DS im Modul DAMAINSORTEN. Durch je einen parametrisierten Aufruf wird dort eine Domainsorte spezifiziert.

Modul: DOMAINSORTEN
Autor: cb *Datum: 10. Dezember 93*
Inhalt: Spezifikation der Domainsorten.

```
(
INCLUDE DS [ SORTS Name OPNS name _, name-r _]
PLUS
INCLUDE DS [ SORTS Addresse OPNS adresse _, adresse-r _]
PLUS
INCLUDE DS [ SORTS Datum OPNS datum _, datum-r _]
PLUS
INCLUDE DS [ SORTS Kennung OPNS kennung _, kennung-r _]
PLUS
INCLUDE DS [ SORTS DatenId OPNS datenid _, datenid-r _]
PLUS
INCLUDE DS [ SORTS Parameterliste OPNS paraliste _, paraliste-r _]
PLUS
INCLUDE DS [ SORTS History OPNS history _, history-r _]
PLUS
INCLUDE DS [ SORTS Binaerdaten OPNS bindaten _, bindaten-r _]
PLUS
INCLUDE DS [ SORTS Roi OPNS roi _, roi-r _]
PLUS
```

INCLUDE DS [**SORTS** AufenthaltsId **OPNS** aufid _, aufid-r _]
PLUS
INCLUDE DS [**SORTS** Familienstand **OPNS** stand _, stand-r _]
PLUS
INCLUDE DS [**SORTS** Nationalitaet **OPNS** nation _, nation-r _]
PLUS
INCLUDE DS [**SORTS** Beruf **OPNS** beruf _, beruf-r _]
PLUS
INCLUDE DS [**SORTS** Religion **OPNS** religion _, religion-r _]
PLUS
INCLUDE DS [**SORTS** Diagnose **OPNS** diagn _, diagn-r _]
PLUS
INCLUDE DS [**SORTS** Aufenthaltsinfo **OPNS** aufinfo _, aufinfo-r _]
PLUS
INCLUDE DS [**SORTS** CreationInfo **OPNS** creationinfo _, creationinfo-r _]
PLUS
INCLUDE DS [**SORTS** PatId **OPNS** patid _, patid-r _]
PLUS
INCLUDE DS [**SORTS** Ort **OPNS** ort _, ort-r _]
PLUS
INCLUDE DS [**SORTS** Geschlecht **OPNS** geschl _, geschl-r _]
PLUS
INCLUDE DS [**SORTS** HospitalId **OPNS** hospid _, hospid-r _]
PLUS
INCLUDE DS [**SORTS** RootId **OPNS** rootid _, rootid-r _]
)

Ende Modul DOMAINSORTEN

Anmerkung: Es soll an dieser Stelle noch einmal darauf hingewiesen werden, daß die Spezifikation der Domainsorten als eine Umbenennung der ganzen Zahlen nur deshalb erfolgt, weil das OBSCURE-System ein Rapid Prototyping nur bei einem leeren Import ermöglicht. Wird kein Rapid-Prototyping angestrebt, genügt es, die Domainsorten (und einige wenige Operationen) zu importieren und unspezifiziert zu lassen.

A.2.2 Attributsorten

Nachdem sich der vorige Unterabschnitt mit den Domainsorten beschäftigte, werden nun die Attributsorten vorgestellt. Zunächst wird erläutert, was Domainsorten und Attributsorten voneinander unterscheidet:
Betrachtet man die Abbildungen zu den einzelnen Entitytypen (Abbildungen 2.10 bis 2.16), so könnte man annehmen, daß Domainsorten — wie z. B. die Sorten *DatenId*, *Name*, *Parameterliste*, *History*, *Binaerdaten*, *ROI* und *CreationInfo*, die zur Spezifikation des Entitytyps *Sunrise-Datenobjekt* (siehe Abbildung 2.10) benötigt

werden — ohne weiteres auch als Attributsorten fungieren können. Nach dieser Überlegung könnte man den Entitytyp *Sunrise-Datenobjekt* z. B. als ein Siebentupel über diesen Domainsorten spezifizieren. Damit schließt man jedoch prinzipiell die Existenz leerer Attributeinträge aus. Es sei denn, die entsprechende Domainsorte sähe bereits einen *leeren Träger* vor (z. B. der leere String *e* der Sorte *String*), was man aber im allgemeinen nicht voraussetzen kann (z. B. bei der Domainsorte *Datum*).

Diese Überlegung zeigt, warum Domainsorten nicht unmittelbar zur Spezifikation von Entitytypen herangezogen werden können, sondern in einem Zwischenschritt um je eine Konstante — die den leeren Attributeintrag der entsprechenden Attributsorte repräsentiert — erweitert werden müssen. Weiteres zu diesem Thema ist zu finden in [Aut93], Seite 17 f. bzw. in [Hec93], Seite 55 f. Die folgende parametrisierte Spezifikation (Modul ATTR) führt einen *Liftingkonstruktor a*, eine Operation *a-r* und eine Konstante *empty* ein. Der Liftingkonstruktor *a* erhebt bzw. liftet einen Träger der Domainsorte zu einem Träger der zugeordneten Attributsorte. Die Umkehrung von *a* wird durch die Operation *a-r* realisiert. Die zusätzliche Konstante *empty* repräsentiert den leeren Attributeintrag.

Modul: ATTR
Autor: cb *Datum: 14. Januar 93*
Inhalt: Parametrisierter Modul zur Spezifikation einer Attributsorte durch Anreicherung einer Domainsorte um einen zusätzlichen Träger, der den leeren Attributeintrag repräsentiert.

IMPORTS
SORTS
Domainsorte
OPNS

_ kleiner _	: Domainsorte Domainsorte	→ bool
_ = _	: Domainsorte Domainsorte	→ bool

CREATE
SORTS
Attributsorte
OPNS

a _	: Domainsorte	→ Attributsorte
empty	:	→ Attributsorte
a-r _	: Attributsorte	→ Domainsorte
_ kleiner _	: Attributsorte Attributsorte	→ bool
_ = _	: Attributsorte Attributsorte	→ bool

SEMANTICS
CONSTRS

a _	: Domainsorte	→ Attributsorte
empty	:	→ Attributsorte

VARS
x1, x2: Attributsorte

```
  d1, d2: Domainsorte
PROGRAMS
  a-r x1 ← CA   SE  x1 OF
                        a d1: d1
                  ELSE  ERROR(Domainsorte)
                  ESAC ;

  x1 kleiner x2   ← CASE x1 OF
                        a d1: CASE x2 OF
                                        a d2: d1 kleiner d2
                                  ELSE  false
                                  ESAC
                  ELSE  CASE x2 OF
                              empty: false
                        ELSE  true
                        ESAC
                  ESAC ;
  ENDCREATE
PARAMS( SORTS Domainsorte)
      [ SORTS Attributsorte
        OPNS a _**,a-r _**, empty ** ]
```

Ende Modul ATTR

In dem folgenden Modul ATTRIBUTSORTEN werden die Domainsorten mit Hilfe des soeben vorgestellten Moduls ATTR zu Attributsorten *geliftet*. Dies erfolgt für jede zu spezifizierende Attributsorte durch einen parametrisierten Aufruf von ATTR. Als Importparameter wird dabei die entsprechende Domainsorte übergeben. Als Exportparameter werden der neue Attributsortenname (A-...), sowie die Namen des Liftingkonstruktors (~), der neuen Konstante (e-a-...) und der Umkehrfunktion des Liftingkonstruktors (~~) eingesetzt.

Modul: ATTRIBUTSORTEN
Autor: cb *Datum: 14. Dezember 93*
Inhalt: Spezifikation der Attributsorten.

```
(
```

Attributsorten zum Entitytyp BENUTZER

```
INCLUDE ATTR ( SORTS Name )
             [ SORTS A-Name OPNS ~ _, ~~ _, e-a-name ]

    PLUS
INCLUDE ATTR ( SORTS Addresse )
             [ SORTS A-Adresse OPNS ~ _, ~~ _, e-a-adresse ]
```

PLUS
INCLUDE ATTR (**SORTS** Datum)
[**SORTS** A-Datum **OPNS** ˜ _, ˜˜ _, e-a-datum]
PLUS
INCLUDE ATTR (**SORTS** Kennung)
[**SORTS** A-Kennung **OPNS** ˜ _, ˜˜ _, e-a-kennung]

Attributsorten zum Entitytyp S.-DATENOBJEKT
— A-Name wurde bei Benutzer spezifiziert
— A-CreationInfo wird bei Aufenthalt spezifiziert

PLUS
INCLUDE ATTR (**SORTS** DatenId)
[**SORTS** A-DatenId **OPNS** ˜ _, ˜˜ _, e-a-datenid]
PLUS
INCLUDE ATTR (**SORTS** Parameterliste)
[**SORTS** A-Parameterliste **OPNS** ˜ _, ˜˜ _, e-a-paraliste]
PLUS
INCLUDE ATTR (**SORTS** History)
[**SORTS** A-History **OPNS** ˜ _, ˜˜ _, e-a-history]
PLUS
INCLUDE ATTR (**SORTS** Binaerdaten)
[**SORTS** A-Binaerdaten **OPNS** ˜ _, ˜˜ _, e-a-bindaten]
PLUS
INCLUDE ATTR (**SORTS** Roi)
[**SORTS** A-Roi **OPNS** ˜ _, ˜˜ _, e-a-roi]

Attributsorten zum Entitytyp DATENKLASSE
— A-Name wurde bei Benutzer spezifiziert

Attributsorten zum Entitytyp AUFENTHALT
— A-Adresse wurde bereits bei BENUTZER spezifiziert
— A-PatId wird bei PATIENT spezifiziert

PLUS
INCLUDE ATTR (**SORTS** AufenthaltsId)
[**SORTS** A-AufenthaltsId **OPNS** ˜ _, ˜˜ _, e-a-aufid]
PLUS
INCLUDE ATTR (**SORTS** Familienstand)
[**SORTS** A-Familienstand **OPNS** ˜ _, ˜˜ _, e-a-stand]
PLUS
INCLUDE ATTR (**SORTS** Aufenthaltsinfo)
[**SORTS** A-Aufenthaltsinfo **OPNS** ˜ _, ˜˜ _, e-a-aufinfo]
PLUS
INCLUDE ATTR (**SORTS** Nationalitaet)
[**SORTS** A-Nationalitaet **OPNS** ˜ _, ˜˜ _, e-a-nation]

PLUS
INCLUDE ATTR (**SORTS** Beruf)
[**SORTS** A-Beruf **OPNS** ~ _, ~~ _, e-a-beruf]
PLUS
INCLUDE ATTR (**SORTS** Religion)
[**SORTS** A-Religion **OPNS** ~ _, ~~ _, e-a-religion]
PLUS
INCLUDE ATTR (**SORTS** Diagnose)
[**SORTS** A-Diagnose **OPNS** ~ _, ~~ _, e-a-diagn]
PLUS
INCLUDE ATTR (**SORTS** CreationInfo)
[**SORTS** A-CreationInfo **OPNS** ~ _, ~~ _, e-a-creationinfo]

Attributsorten zum Entitytyp PATIENT
— *A-Name, A-Datum wurden bereits bei BENUTZER spezifiziert*
— *A-CreationInfo wurde bereits bei AUFENTHALT spezifiziert*

PLUS
INCLUDE ATTR (**SORTS** PatId)
[**SORTS** A-PatId **OPNS** ~ _, ~~ _, e-a-patid]
PLUS
INCLUDE ATTR (**SORTS** Ort)
[**SORTS** A-Ort **OPNS** ~ _, ~~ _, e-a-ort]
PLUS
INCLUDE ATTR (**SORTS** Geschlecht)
[**SORTS** A-Geschlecht **OPNS** ~ _, ~~ _, e-a-geschl]

Attributsorten zum Entitytyp HOSPITAL
— *A-Name, A-Adresse wurden bereits bei BENUTZER spezifiziert*
— *A-CreationInfo, wurde bereits bei AUFENTHALT spezifiziert*

PLUS
INCLUDE ATTR (**SORTS** HospitalId)
[**SORTS** A-HospitalId **OPNS** ~ _, ~~ _, e-a-hospid]

Attributsorten zum Entitytyp ROOT

PLUS
INCLUDE ATTR (**SORTS** RootId)
[**SORTS** A-RootId **OPNS** ~ _, ~~ _, e-a-rootid]
)
Ende Modul ATTRIBUTSORTEN

A.2.3 Schlüssel

Schlüssel dienen der Identifizierung von Entities (und Relationships) in einem essentiellen Speicher bzw. Datenbank. Prinzipiell kann die Schlüsselsorte eines Entitytyps spezifiziert werden als ein n-Tupel über den Domainsorten der n Attributsorten des Entitytyps, die mit dem Zusatz *primary key* versehen sind. Bei fast allen Entitytypen ist jeweils nur eine Attributsorte mit Schlüsselfunktion ausgezeichnet. In diesen Fällen ist keine Tupelbildung erforderlich[2]. Nur der Entitytyp *Aufenthalt* (siehe Abbildung 2.13) zeichnet zwei Attributsorten mit Schlüsselfunktion aus. In diesem Fall ist also eine Paarbildung über den Domainsorten der ausgezeichneten Attributsorten erforderlich, um die Schlüsselsorte des Entitytyps *Aufenthalt* zu spezifizieren. Dies erfolgt im Modul KEY. Weitere Anmerkungen zum Thema Schlüssel finden sich in [Hec93] auf Seite 47 ff.

Anmerkung: In [Hec93] finden sich insbesondere Erläuterungen zu abstrakten Schlüsseln, von denen hier abgesehen wird. Insgesamt spielen Schlüssel in dieser Spezifikation nur eine nebensächliche Rolle.

Modul: KEY
Autor: cb *Datum: 15. Dezember 93*
Inhalt: Spezifikation der Schlüsselsorte PatId-x-AufenthaltsId.

INCLUDE TUPEL-2(**SORTS** PatId AufenthaltsId)
[**SORTS** PatId-x-AufenthaltsId
OPNS {_/_}, get_fst, get_snd, set_fst, set_snd]

Ende Modul KEY

A.3 Spezifikation der Datenbankebene

Nachdem die im Vorigen Abschnitt die Attributebene vorgestellt wurde, ist es das Ziel dieses Abschnitts, die Spezifikation der Datenbankebene zu erläutern. Diese setzt sich nach Abbildung A.1 aus den Modulen DB, ENTITY-KOMPONENTE und RELATIONSHIP-KOMPONENTE zusammen. Zuerst werden Entity-Komponenten (Sorte *Entity-Db*) im Modul ENTITY-KOMPONENTE spezifiziert, danach Relationship-Komponenten (Sorte *Relationship-Db*) im Modul RELATIONSHIP-KOMPONENTE und schließlich wird im Modul DB (bzw. in dessen Untermodul RAW_DB) die Datenbanksorte *Db* definiert durch Paarbildung über den Sorten *Entity-Db* und *Relationship-Db*.

[2] Hier erfolgt ein geringfügiges Abweichen von [Aut93]. Dort wird in allen Fällen eine Tupelbildung durchgeführt, auch wenn nur ein Attribut mit Schlüsselfunktion ausgezeichnet ist.

A.3.1 Entity-Komponenten

Die sieben Entitytypen des ausgewählten Ausschnitts (siehe Abbildungen 2.10 bis 2.16) werden eingeführt als n-Tupel über den jeweiligen Attributsorten. Der Entitytyp *Sunrise-Datenobjekt* wird beispielsweise spezifiziert als ein 7-Tupel über den Attributsorten *A-DatenId*, *A-Name*, *A-Parameterliste*, *A-History*, *A-Binaerdaten*, *A-Roi*, *A-CreationInfo*. Ferner wird zu jeder Attributsorte eines Entitytyps ein Prädikat definiert, daß für einen Träger dieser Attributsorte überprüft, ob dieser dem leeren Attributeintrag entspricht.

Anstatt eine solche Spezifikation für jeden Entitytyp neu zu formulieren, werden, wie in [Aut93] vorgeschlagen, parametrisierte Module MK_n-ATTR-ENTITY spezifiziert (momentan existieren solche für n=1..10), denen n Attributsorten als Importparameter und der Name des zu kreierenden Entitytyps, sowie einige Operationsnamen als Exportparameter übergeben werden. Die Operationen, die durch die Exportparameter umbenannt werden, dienen dem Kreieren von Entities, sowie dem Zugriff auf und dem Setzen von Attributwerten. Durch einen parametrisierten Aufruf des Moduls MK_n-ATTR-ENTITY kann dann ein Entitytyp (mit n Attributen) spezifiziert werden. Ein solcher parametrisierter Aufruf wird anschließend erweitert um eine Operation *key*, die den Schlüssel einer Entity des spezifizierten Typs berechnet. Weil die Schlüsselsorten für jeden Entitytyp prinzipiell beliebig aus den Attributsorten des Entitytyps zusammengesetzt sein können, kann diese Operation nicht bereits durch den MK_n-ATTR-ENTITY Modul bereitgestellt werden, sondern die Operation *key* muß für jeden Entitytyp speziell spezifiziert werden.

Zunächst wird, stellvertretend für alle MK_n-ATTR-ENTITY Module, derjenige für $n = 7$ vorgestellt, bevor dann, stellvertretend für die sieben Module zu den Entitytypen, der Modul SUNRISE-DATENOBJEKT präsentiert wird.

Modul: MK_7-ATTR-ENTITY
Autor: cb *Datum: 17. Januar 94*
Inhalt: Parametrisierter Modul zur Spezifikation eines Entitytyps mit sieben Attributen.

(**INCLUDE** TUPEL-7
  (**SORTS** Attr1, Attr2, Attr3, Attr4, Attr5, Attr6, Attr7)
  [**SORTS** Entity-7 **OPNS** create-entity,
  attr1, attr2, attr3, attr4, attr5, attr6, attr7,
  set-attr1, set-attr2, set-attr3, set-attr4,
  set-attr5, set-attr6, set-attr7])
EXTENDED BY
IMPORTS
 OPNS
  empty-attr1: → Attr1
  empty-attr2: → Attr2
  empty-attr3: → Attr3
  empty-attr4: → Attr4

empty-attr5: → Attr5
empty-attr6: → Attr6
empty-attr7: → Attr7

CREATE
OPNS
entity-attr1-ex : Entity-7 → bool
entity-attr2-ex : Entity-7 → bool
entity-attr3-ex : Entity-7 → bool
entity-attr4-ex : Entity-7 → bool
entity-attr5-ex : Entity-7 → bool
entity-attr6-ex : Entity-7 → bool
entity-attr7-ex : Entity-7 → bool

SEMANTICS
VARS
e: Entity-7
PROGRAMS
entity-attr1-ex(e) ← not(attr1(e) = empty-attr1);
entity-attr2-ex(e) ← not(attr2(e) = empty-attr2);
entity-attr3-ex(e) ← not(attr3(e) = empty-attr3);
entity-attr4-ex(e) ← not(attr4(e) = empty-attr4);
entity-attr5-ex(e) ← not(attr5(e) = empty-attr5);
entity-attr6-ex(e) ← not(attr6(e) = empty-attr6);
entity-attr7-ex(e) ← not(attr7(e) = empty-attr7);

ENDCREATE
ENDEXTENDED
PARAMS
(**SORTS** Attr1, Attr2, Attr3, Attr4, Attr5, Attr6, Attr7
OPNS empty-attr1 **, empty-attr2 **, empty-attr3 **,
empty-attr4 **, empty-attr5 **, empty-attr6 **,
empty-attr7 **)
[**SORTS** Entity-7
OPNS create-entity **,
attr1 **, set-attr1 **, attr2 **, set-attr2 **,
attr3 **, set-attr3 **, attr4 **, set-attr4 **,
attr5 **, set-attr5 **, attr6 **, set-attr6 **,
attr7 **, set-attr7 **,
entity-attr1-ex **, entity-attr2-ex **,
entity-attr3-ex **, entity-attr4-ex **
entity-attr5-ex **, entity-attr6-ex **,
entity-attr7-ex **]

Ende Modul MK_7-ATTR-ENTITY

Modul: SUNRISE-DATENOBJEKT
Autor: cb *Datum: 17. Januar 94*
Inhalt: Spezifikation des Entitytyps Sunrise-Datenobjekt.

INCLUDE
MK_7-ATTR-ENTITY
(**SORTS** A-DatenId, A-Name, A-Parameterliste, A-History,
A-Binaerdaten, A-Roi, A-CreationInfo
OPNS e-a-datenid, e-a-name, e-a-paraliste, e-a-history,
e-a-bindaten, e-a-roi, e-a-creationinfo)
[**SORTS** Sunrise-Datenobjekt
OPNS create-Sunrise-Datenobjekt,
DatenId, set-DatenId,
Datenname, set-Datenname,
Parameterliste, set-Parameterliste,
History, set-History,
Binaerdaten, set-Binaerdaten,
Roi, set-Roi,
CreationInfo, set-CreationInfo,
Sunrise-Datenobjekt-DatenId-ex,
Sunrise-Datenobjekt-Datenname-ex,
Sunrise-Datenobjekt-Parameterliste-ex,
Sunrise-Datenobjekt-History-ex,
Sunrise-Datenobjekt-Binaerdaten-ex,
Sunrise-Datenobjekt-Roi-ex,
Sunrise-Datenobjekt-CreationInfo-ex]
EXTENDED BY
IMPORTS
SORTS DatenId
OPNS ˜˜ _: A-DatenId → DatenId
CREATE
OPNS key : Sunrise-Datenobjekt → DatenId
SEMANTICS
VARS p : Sunrise-Datenobjekt
PROGRAMS key(p) ← ˜˜ DatenId(p)
ENDCREATE
ENDEXTENDED

Ende Modul SUNRISE-DATENOBJEKT

Zusammengefaßt werden die Spezifikationen der Entitytypen im Modul ENTITIES.

Modul: ENTITIES
Autor: cb *Datum: 17. Januar 94*
Inhalt: Zusammensetzen der Spezifikationen der sieben Entytypen zum Ausschnitt duales Speicherkonzept.

INCLUDE BENUTZER
PLUS
INCLUDE SUNRISE-DATENOBJEKT
PLUS
INCLUDE DATENKLASSE
PLUS
INCLUDE AUFENTHALT
PLUS
INCLUDE PATIENT
PLUS
INCLUDE HOSPITAL
PLUS
INCLUDE ROOT

Ende Modul ENTITIES

Der nächste Schritt betrifft die Spezifikation von Mengen[3] von Entities. In Anlehnung an [Aut93] wird eine Menge von Entities im folgenden auch *Ansammlung* oder *Collection* genannt. Der Name des Moduls zur Spezifikation von Mengen von S.-Datenobjekten lautet deshalb DATENOBJEKT-COLLECTION; in ihm kreiert wird die Sorte *Datenobjekt-Collection*. Analog zur Spezifikation der Entitytypen, wird zur Spezifikation von Ansammlungen von Entities ein parametrisierter Modul MK_ENTITY_COLLECTION verwendet. Dieser wird zuerst vorgestellt. Es folgt stellvertretend für sämtliche sieben Ansammlungssorten (*Root-Colletion*, *Hospital-Collection*, *Patient-Collection*, *Aufenthalt-Collection*, *Datenobjekt-Collection*, *Benutzer-Collection* und *Datenklasse-Collection*) der Modul DATENOBJEKT-COLLECTION, in dem der Modul MK_ENTITY_COLLECTION auf die Entitysorte *Sunrise-Datenobjekt* angewendet wird.

Modul: MK_ENTITY_COLLECTION
Autor: cb *Datum: 17. Januar 94*
Inhalt: Parametrisierter Modul zur Spezifikation von Ansammlungen (Mengen bzw. Monolisten) von Entities eines bestimmten Entitytyps.

INCLUDE MLIST (**SORTS** Entitytyp)
[**SORTS** Entitycollection
OPNS emptycollection, makecollection]

[3] Anstelle von Mengen werden Monolisten verwendet. Siehe hierzu auch die Anmerkungen in [Hec93], Seite 45 ff. Der Leser kann sich hier aber getrost unter Monolisten herkömmliche Mengen vorstellen.

```
EXTENDED BY
IMPORTS
 SORTS Key-Entitytyp
 OPNS  key : Entitytyp → Key-Entitytyp
       _ = _ : Key-Entitytyp Key-Entitytyp → bool
CREATE
 OPNS  is-in-Entitycollection : Entitytyp Entitycollection → bool
       get-Entitytyp : Key-Entitytyp Entitycollection → Entitytyp
       update-Entitytyp : Entitytyp Entitycollection → Entitycollection
       check-primary-key : Entitycollection → bool
SEMANTICS
 VARS  e, e1 : Entitytyp
       ec : Entitycollection
       ke : Key-Entitytyp
 PROGRAMS
      is-in-Entitycollection(e,ec) ←
        IF ec = emptycollection
        THEN false
        ELSE LET e1 : head(ec)
             IN  IF key(e1) = key(e)
                 THEN true
                 ELSE is-in-Entitycollection(e,tail(ec))
                 FI
             TEL
        FI ;

      get-Entitytyp(ke,ec) ←
        IF ec = emptycollection
        THEN ERROR(Entitytyp)
        ELSE LET e1 : head(ec)
             IN  IF key(e1) = ke
                 THEN e1
                 ELSE get-Entitytyp(ke,tail(ec))
                 FI
             TEL
        FI ;

      update-Entitytyp(e,ec) ←
        IF ec = emptycollection
        THEN ERROR(Entitycollection)
        ELSE LET e1 : head(ec)
             IN  IF key(e1) = key(e)
                 THEN ins(e,tail(ec))
                 ELSE ins(e1,update-Entitytyp(e,tail(ec)))
```

```
                FI
            TEL
        FI ;

    check-primary-key(ec) ←
        IF ec = emptycollection
        THEN true
        ELSE LET e1 : head(ec)
             IN  not(is-in-Entitycollection(e1,tail(ec)))
                 and
                 check-primary-key(tail(ec))
             TEL
        FI ;

ENDCREATE
ENDEXTENDED
PARAMS
 ( SORTS Entitytyp, Key-Entitytyp OPNS key ** )
 [ SORTS Entitycollection
   OPNS  emptycollection **, makecollection **, is-in-Entitycollection **,
         get-Entitytyp **, update-Entitytyp **, check-primary-key ** ]
```

Ende Modul MK_ENTITY_COLLECTION

Modul: DATENOBJEKT-COLLECTION
Autor: cb *Datum: 18. Januar 94*
Inhalt: Spezifikation von Ansammlungen von Entitities des Typs S.-Datenobjekt.

```
INCLUDE MK_ENTITY_COLLECTION
        ( SORTS  Sunrise-Datenobjekt, DatenId
          OPNS   key)
        [ SORTS  Datenobjekt-coll
          OPNS   e-Datenobjekt-coll, make-Datenobjekt-coll,
                 is-in-Datenobjekt-coll, get-Datenobjekt,
                 update-Datenobjekt, primary-key-Datenobjekt ]
```

Ende Modul DATENOBJEKT-COLLECTION

Der Modul ENTITY-COLLECTIONS faßt die sieben Einzelspezifikationen der Ansammlungssorten zusammen.

Modul: ENTITY-COLLECTIONS
Autor: cb *Datum: 18. Januar 94*

Inhalt: Zusammensetzen der Spezifikationen von Ansammlungen von Entities.

INCLUDE AUFENTHALT-COLLECTION
PLUS
INCLUDE PATIENT-COLLECTION
PLUS
INCLUDE HOSPITAL-COLLECTION
PLUS
INCLUDE ROOT-COLLECTION
PLUS
INCLUDE BENUTZER-COLLECTION
PLUS
INCLUDE DATENKLASSE-COLLECTION
PLUS
INCLUDE DATENOBJEKT-COLLECTION

Ende Modul ENTITY-COLLECTIONS

Nun können Entity-Komponenten spezifiziert werden als Siebentupel über den soeben eingeführten Ansammlungssorten. Erweitert wird diese Spezifikation um eine Konstante, die die leere Entity-Komponente repräsentiert. Die Spezifikationen der Entitytypen (Modul ENTITIES) und der Ansammlungssorten (Modul ENTITY-COLLECTIONS) werden mittels des OBSCURE-Sprachkonstrukts X_COMPOSE hinzugefügt, so daß die Spezifikation von Entity-Komponenten des essentiellen Speichers nun fast komplett ist; importiert werden lediglich noch die Attributsorten.

Modul: ENTITY-KOMPONENTE
Autor: cb *Datum: 21. Januar 94*
Inhalt: Spezifikation von Entity-Komponenten.

INCLUDE TUPEL-7
(**SORTS** Benutzer-coll, Datenklasse-coll, Datenobjekt-coll,
Aufenthalt-coll, Patient-coll, Hospital-coll, Root-coll)
[**SORTS** Entity-Db
OPNS {_/_/_/_/_/_/_},
get-Benutzer-coll, get-Datenklasse-coll,
get-Datenobjekt-coll, get-Aufenthalt-coll,
get-Patient-coll, get-Hospital-coll,
get-Root-coll,
set-Benutzer-coll, set-Datenklasse-coll,
set-Datenobjekt-coll, set-Aufenthalt-coll,
set-Patient-coll, set-Hospital-coll,
set-Root-coll]
EXTENDED BY

```
IMPORTS
 OPNS
  e-Benutzer-coll       : → Benutzer-coll
  e-Datenklasse-coll    : → Datenklasse-coll
  e-Datenobjekt-coll    : → Datenobjekt-coll
  e-Aufenthalt-coll     : → Aufenthalt-coll
  e-Patient-coll        : → Patient-coll
  e-Hospital-coll       : → Hospital-coll
  e-Root-coll           : → Root-coll
CREATE
 OPNS
  e-Entity-Db : →       Entity-Db
SEMANTICS
 VARS
 PROGRAMS
  e-Entity-Db ←
   { e-Benutzer-coll / e-Datenklasse-coll / e-Datenobjekt-coll /
   e-Aufenthalt-coll/ e-Patient-coll / e-Hospital-coll / e-Root-coll }
 ENDCREATE
ENDEXTENDED
X_COMPOSE
INCLUDE ENTITY-COLLECTIONS
X_COMPOSE
INCLUDE ENTITIES
```

Ende Modul ENTITY-KOMPONENTE

A.3.2 Relationship-Komponenten

Während im lezten Abschnitt die Spezifikation von Entity-Komponenten vorgestellt wurde, werden nun Relationship-Komponenten spezifiziert. Als erstes werden dazu die elf Relationshiptypen des ausgewählten Ausschnitts (siehe Abbildung 2.8) — dies sind *Rootbezug*, *Hospitalbezug*, *Patientenbezug*, *Aufenthaltsbezug*, *Lokal-fuer*, *Global-fuer*, *Stack-Objekt-von*, *Aktuelles-Stack-Objekt-von*, *Default-Objekt*, *Datenklassenbezug* und *Entstanden-aus* — spezifiziert auf der Grundlage eines parametrisierten Moduls MK_RELATIONSHIP. Parametrisiert ist dieser Modul über zwei Schlüsselsorten. Bei einem Aufruf werden MK_RELATIONSHIP die Schlüsselsorten der beteiligten Entitytypen übergeben, zwischen denen der Relationshiptyp definiert werden soll. Der Modul MK_RELATIONSHIP greift selbst auf einen parametrisierten Standardmodul[4] MAKE_BINARY_RELATION zurück, in dem 2-stellige Relationen spezifiziert werden und der hier, aus Umfangsgründen, nicht vorgestellt wird.

[4]Standardmodule sind Module, die vom OBSCURE-System jedem Benutzer standardmäßig zur Verfügung gestellt werden. Siehe auch Abschnitt A.6.

Stattdessen werden die Signaturinformationen zu diesem Modul aufgelistet. (Die folgende Auflistung wurde mit Pretty-Printer des OBSCURE-Systems automatisch erstellt.)

```
File name: "MAKE_BINARY_RELATION.T"
list of sorts and operations:

imported and exported (inherited).
 SORTS
  Sort2 Sort1
 OPNS
  _ = _ : Sort2 Sort2 -> bool
  _ = _ : Sort1 Sort1 -> bool

not imported and exported (created).
 SORTS
  SoSort2 SoSort1 Binary_Relation Pair
 OPNS
  e_SoSort2 : -> SoSort2
  mk-mlist : SoSort2 -> SoSort2
  reachable_from : Sort1 Binary_Relation -> SoSort2
  ins : Sort2 SoSort2 -> SoSort2
  del : Sort2 SoSort2 -> SoSort2
  codomain : Binary_Relation -> SoSort2
  tail : SoSort2 -> SoSort2
  head : SoSort2 -> Sort2
  get_snd : Pair -> Sort2
  _ is-in _ : Sort2 SoSort2 -> bool
  _ is-in _ : Sort1 SoSort1 -> bool
  _ is-in _ : Pair Binary_Relation -> bool
  _ = _ : SoSort2 SoSort2 -> bool
  _ = _ : SoSort1 SoSort1 -> bool
  _ = _ : Binary_Relation Binary_Relation -> bool
  _ = _ : Pair Pair -> bool
  mk-mlist : SoSort1 -> SoSort1
  domain : Binary_Relation -> SoSort1
  e_SoSort1 : -> SoSort1
  ins : Sort1 SoSort1 -> SoSort1
  reachable : Sort2 Binary_Relation -> SoSort1
  del : Sort1 SoSort1 -> SoSort1
  tail : SoSort1 -> SoSort1
  head : SoSort1 -> Sort1
  get_fst : Pair -> Sort1
  mk-mlist : Binary_Relation -> Binary_Relation
  ins : Pair Binary_Relation -> Binary_Relation
```

```
  e_Binary_Relation : -> Binary_Relation
  del : Pair Binary_Relation -> Binary_Relation
  tail : Binary_Relation -> Binary_Relation
  set_snd : Sort2 Pair -> Pair
  head : Binary_Relation -> Pair
  set_fst : Sort1 Pair -> Pair
  { _ / _ } : Sort1 Sort2 -> Pair
  # : SoSort2 -> integer
  # : SoSort1 -> integer
  # : Binary_Relation -> integer

imported and not exported (hidden).
  ## no sorts
  ## no operations

list of constructors:

imported

exported
  { _ / _ } : Sort1 Sort2 -> Pair

list of parameters:

imported.
 SORTS
  Sort2 Sort1
  ## no operations

exported.
 SORTS
  SoSort2 SoSort1 Pair Binary_Relation
 OPNS
  e_SoSort2 : -> SoSort2
  e_SoSort1 : -> SoSort1
  { _ / _ } : Sort1 Sort2 -> Pair
  e_Binary_Relation : -> Binary_Relation
## The End
```

Anmerkung: Der Spezialfall, daß die beiden Schlüsselsorten, die als Parameter an MAKE_BINARY_RELATION übergeben werden, identisch sind — dies ist z. B. der Fall für Relationshiptypen, die auf nur einem Entitytyp definiert werden (Entitytypen *Entstanden-aus* und *Default-Objekt*) —, kann wegen des Sortenclashproblems[5]

[5]Das Sortenclashproblem stellt sich, wenn an verschiedenen Stellen einer Spezifikation Sorten gleichen Namens generiert werden, die miteinander identifiziert werden sollen. Dieses Problem tritt

nicht mittels MAKE_BINARY_RELATION spezifiziert werden. Für diesen Spezialfall steht ein parametrisierter Modul MK_SPECIAL_BINARY_RELATION bereit. Prinzipiell ist dieser dem Modul MK_BINARY_RELATION sehr ähnlich. MK_SPECIAL_BINARY_RELATION wird dann im Modul MK_SPECIAL_RELATIONSHIP verwendet um Relationshiptypen mit zwei identischen Schlüsselsorten zu definieren. MK_SPECIAL_BINARY_RELATION und MK_SPECIAL_RELATIONSHIP werden nicht vorgestellt.

Es wird nun der Modul MK_RELATIONSHIP vorgestellt.

Modul: MK_RELATIONSHIP
Autor: cb *Datum: 21. Januar 94*
Inhalt: Parametrisierter Modul zur Spezifikation von Relationshiptypen.

INCLUDE MAKE_BINARY_RELATION
(**SORTS** Key1, Key2)
[**SORTS** Relationship, Key1_x_Key2, SoKey1, SoKey2
OPNS e-relationship, {_/_}, e-SoKey1, e-SoKey2]
EXTENDED BY
CREATE
OPNS

estrelationship	: Relationship Key1 Key2	→ Relationship
relrelationship	: Relationship Key1 Key2	→ Relationship
is_in	: Key1 Relationship	→ bool
is_in	: Key2 Relationship	→ bool
is_in	: Key1 Key2 Relationship	→ bool
relationship_1_N	: Relationship	→ bool
relationship_N_1	: Relationship	→ bool
relationship_N_N	: Relationship	→ bool
relationship_1_1	: Relationship	→ bool

SEMANTICS
VARS

k1	: Key1
k2	: Key2
k	: Key1_x_Key2
rel	: Relationship

PROGRAMS
estrelationship(rel,k1,k2) ← ins({k1/k2},rel);
relrelationship(rel,k1,k2) ← del({k1/k2},rel);
is_in(k1,rel) ← k1 is-in domain(rel);
is_in(k2,rel) ← k2 is-in codomain(rel);

insbesondere im Zusammenhang mit der Verwendung von Standardmodulen auf. Als mögliche Lösung wurde am Lehrstuhl von Prof. Dr.-Ing. Loeckx die sogenannte Dummy-Methode diskutiert, auf deren Anwendung hier, aus Übersichtsgründen und um ein weiteres Abweichen von [Aut93] zu vermeiden, verzichtet wurde.

```
  is_in(k1,k2,rel) ← ({k1/k2}) is-in rel;
  relationship_1_N(rel) ←
    IF      rel = e-relationship
    THEN true
    ELSE  LET k : head(rel)
          IN  #(reachable(get_snd(k),rel)) = 1
              and
              relationship_1_N(del(k,rel))
          TEL
    FI ;
  relationship_N_1(rel) ←
    IF      rel = e-relationship
    THEN true
    ELSE  LET k : head(rel)
          IN  #(reachable_from(get_fst(k),rel)) = 1
              and
              relationship_N_1(del(k,rel))
          TEL
    FI ;
  relationship_N_N(rel) ← true;
  relationship_1_1(rel) ← relationship_1_N(rel) and relationship_N_1(rel);
ENDCREATE
ENDEXTENDED
PARAMS
  ( SORTS Key1, Key2 )
  [ SORTS Relationship, Key1_x_Key2, SoKey1, SoKey2
    OPNS e-relationship **, {_/_} **, e-SoKey1 **, e-SoKey2 **,
          estrelationship **, relrelationship **,
          relationship_1_N **, relationship_N_1 **,
          relationship_N_N **, relationship_1_1 ** ]
```

Ende Modul MK_RELATIONSHIP

Durch einen Aufruf von MK_RELATIONSHIP mit den Parametern *DatenId* und *Klassenname* — dies sind die Schlüsselsorten der Entitytypen *Sunrise-Datenobjekt* und *Datenklasse* — kann nun beispielsweise der Relationshiptyp *Datenklassenbezug* spezifizert werden. Analog dazu werden auch die zehn weiteren Relationshiptypen eingeführt. Man beachte lediglich, daß, aus zuvor bereits angesprochenen Gründen, zur Spezifikation der Relationshiptypen *Default-Objekt* und *Entstanden-aus* der Modul MK_SPECIAL_RELATIONSHIP anstelle von MK_RELATIONSHIP eingesetzt werden muß.

Modul: DATENKLASSENBEZUG

Autor: cb *Datum: 22. Januar 94*
Inhalt: Spezifikation des Relationshiptyps Datenklassenbezug.

INCLUDE MK_RELATIONSHIP
(**SORTS** DatenId, Name)
[**SORTS** Datenklassenbezug, EoDatenklassenbezug,
SoDatenId-Datenklassenbezug,
SoKlassenname-Datenklassenbezug
OPNS e-Datenklassenbezug,
Datenklassenbezug,
e-SoDatenId-Datenklassenbezug,
e-SoKlassenname-Datenklassenbezug,
est-Datenklassenbezug, rel-Datenklassenbezug,
Datenklassenbezug_1_N, Datenklassenbezug_N_1,
Datenklassenbezug_N_N, Datenklassenbezug_1_1]

Ende Modul DATENKLASSENBEZUG

Zusammengesetzt werden die Spezifikationen der elf Relationshiptypen im Modul RELATIONSHIPS.

Modul: RELATIONSHIPS
Autor: cb *Datum: 22. Januar 94*
Inhalt: Zusammensetzen der Spezifikationen der elf Relationshiptypen.

INCLUDE ROOTBEZUG
PLUS
INCLUDE HOSPITALBEZUG
PLUS
INCLUDE PATIENTENBEZUG
PLUS
INCLUDE AUFENTHALTSBEZUG
PLUS
INCLUDE LOKAL-FUER
PLUS
INCLUDE GLOBAL-FUER
PLUS
INCLUDE STACK-OBJEKT-VON
PLUS
INCLUDE AKTUELLES-STACK-OBJEKT-VON
PLUS
INCLUDE DATENKLASSENBEZUG
PLUS
INCLUDE DEFAULT-OBJEKT

PLUS
INCLUDE ENTSTANDEN-AUS

Ende Modul RELATIONSHIPS

Relationship-Komponenten (Sorte *Relationship-Db*) des essentiellen Speichers können nun spezifiziert werden als 11-Tupel über den eingeführten Relationshiptypen. Dies erfolgt im Modul RELATIONSHIP-KOMPONENTE.

Modul: RELATIONSHIP-KOMPONENTE
Autor: cb *Datum: 24. Januar 94*
Inhalt: Spezifikation von Relationship-Komponenten.

INCLUDE TUPEL-11
(**SORTS** Rootbezug,
Hospitalbezug, Patientenbezug,
Aufenthaltsbezug, Lokal-fuer,
Global-fuer, Stack-Objekt-von,
Aktuelles-Stack-Objekt-von,
Datenklassenbezug, Default-Objekt,
Entstanden-aus)
[**SORTS** Relationship-Db
OPNS {-/-/-/-/-/-/-/-/-/-/-},
get-Rootbezug,
get-Hospitalbezug,
get-Patientenbezug,
get-Aufenthaltsbezug,
get-Lokal-fuer,
get-Global-fuer,
get-Stack-Objekt-von,
get-Aktuelles-Stack-Objekt-von,
get-Datenklassenbezug,
get-Default-Objekt,
get-Entstanden-aus,
set-Rootbezug,
set-Hospitalbezug,
set-Patientenbezug,
set-Aufenthaltsbezug,
set-Lokal-fuer,
set-Global-fuer,
set-Stack-Objekt-von,
set-Aktuelles-Stack-Objekt-von,
set-Datenklassenbezug,
set-Default-Objekt,

set-Entstanden-aus]

EXTENDED BY
IMPORTS
OPNS

e-Rootbezug	: → Rootbezug
e-Hospitalbezug	: → Hospitalbezug
e-Patientenbezug	: → Patientenbezug
e-Aufenthaltsbezug	: → Aufenthaltsbezug
e-Lokal-fuer	: → Lokal-fuer
e-Global-fuer	: → Global-fuer
e-Stack-Objekt-von	: → Stack-Objekt-von
e-Aktuelles-Stack-Objekt-von	: → Aktuelles-Stack-Objekt-von
e-Datenklassenbezug	: → Datenklassenbezug
e-Default-Objekt	: → Default-Objekt
e-Entstanden-aus	: → Entstanden-aus

CREATE
OPNS

e-Relationship-Db	: → Relationship-Db

SEMANTICS
VARS
PROGRAMS

e-Relationship-Db ← {
e-Rootbezug/e-Hospitalbezug/e-Patientenbezug/
e-Aufenthaltsbezug/e-Lokal-fuer/e-Global-fuer/
e-Stack-Objekt-von/e-Aktuelles-Stack-Objekt-von/
e-Datenklassenbezug/e-Default-Objekt/e-Entstanden-aus}

ENDCREATE
ENDEXTENDED
X_COMPOSE
INCLUDE RELATIONSHIPS

Ende Modul RELATIONSHIP-KOMPONENTE

A.3.3 Spezifikation der Datenbanksorte *Db*

Datenbankvertreter (Träger der Sorte *Db*) werden nun spezifiziert als Paare bestehend aus einer Entity- und einer Relationship-Komponente. Dies erfolgt im Modul RAW_DB (Untermodul von DB).

Modul: RAW_DB
Autor: cb *Datum: 18. Januar 94*
Inhalt: Spezifikation der Sorte Db (essentieller Speicher zum Ausschnitt duales Speicherkonzept).

INCLUDE TUPEL-2(**SORTS** Entity-Db, Relationship-Db)
[**SORTS** Db,
OPNS {_/_},
get-Entity-Db, get-Relationship-Db,
set-Entity-Db, set-Relationship-Db]
EXTENDED BY
IMPORTS
OPNS
e-Entity-Db : → Entity-Db
e-Relationship-Db : → Relationship-Db
CREATE
OPNS
e-Db : → Db
SEMANTICS
VARS
PROGRAMS
e-Db ← {e-Entity-Db/e-Relationship-Db}
ENDCREATE
ENDEXTENDED

Ende Modul RAW_DB

Im Modul DB wird der Modul RAW_DB erweitert um Operationen, die einen abstrakteren Umgang mit der Datenbank (dem essentiellen Speicher) ermöglichen sollen und somit zu einer kürzeren und übersichtlicheren Spezifikation der Aktivitätenebene beitragen können. Dazu betrachte man folgendes Beispiel: Angenommen man möchte zur Spezifikation einer essentiellen Aktivität auf eine konkrete Entity — z. B. diejenige mit der DatenId *di* — in einer nichtleeren Datenbank *db* zugreifen. Dies könnte man mit den bis jetzt zur Verfügung gestellten Operationen wie folgt tun: get-Datenobjekt(di,get-Datenobjekt-coll(get-Entity-Db(db))). Leider ist diese Zeile nicht besonders einfach zu lesen (und zu verstehen), weil die Formulierung zu sehr auf den Details der Spezifikation der Datenbanksorte aufbaut. Solche Details sind aber überhaupt nicht das Ziel der Spezifikation. Im Gegenteil, eigentlich sollte die Datenbanksorte des Zielsystems so abstrakt wie möglich beschrieben werden. Deshalb werden nun neue Operationen eingeführt, die den wünschenswerten abstrakten Umgang mit der Datenbanksorte ermöglichen. Der Spezifikationstext dieser Operationen enthält natürlich wieder Formulierungen, wie oben vorgestellt. Doch in der Spezifikation zur Aktivitätenebene können nun die abstrakteren Operationen verwendet werden. Es wird z. B. eine Operation *get-Sunrise-Datenobjekt* : *DatenId Db -> Sunrise-Datenobjekt* spezifiziert, die für eine gegebene DatenId *di* den Zugriff auf die entsprechende Entity des Typs *Sunrise-Datenobjekt* in einer Datenbank *db* wie folgt erleichtert: *get-Sunrise-Datenobjekt(di,db)*.

Anmerkung: Von den in [Aut93] vorgesehenen Operationen werden nur die bereitgestellt, die für die spezielle Aktivitätenebene dieser Fallstudie nützlich sind.

Modul: DB
Autor: cb *Datum: 25. Januar 94*
Inhalt: Erweiterung der Spezifikation RAW_DB um Operationen, die einen abstrakteren Umgang mit der Datenbanksorte ermöglichen sollen.

INCLUDE RAW_DB
EXTENDED BY
IMPORTS
SORTS
Kennung Aktuelles-Stack-Objekt-von Default-Objekt DatenId
PatId-x-AufenthaltsId PatId Name
Sunrise-Datenobjekt Global-fuer Lokal-fuer
Datenobjekt-coll Stack-Objekt-von Aufenthaltsbezug Patientenbezug
Datenklassenbezug Entstanden-aus
OPNS
get-Stack-Objekt-von : Relationship-Db → Stack-Objekt-von
get-Global-fuer : Relationship-Db → Global-fuer
get-Lokal-fuer : Relationship-Db → Lokal-fuer
get-Aktuelles-Stack-Objekt-von :
Relationship-Db → Aktuelles-Stack-Objekt-von
get-Entstanden-aus : Relationship-Db → Entstanden-aus
get-Default-Objekt : Relationship-Db → Default-Objekt
get-Aufenthaltsbezug : Relationship-Db → Aufenthaltsbezug
get-Datenklassenbezug : Relationship-Db → Datenklassenbezug
get-Patientenbezug : Relationship-Db → Patientenbezug
is_in : Kennung Stack-Objekt-von → bool
is_in : DatenId Stack-Objekt-von → bool
is_in : DatenId Kennung Global-fuer → bool
is_in : DatenId Global-fuer → bool
is_in : DatenId Kennung Lokal-fuer → bool
is_in : DatenId Lokal-fuer → bool
is_in-fst : DatenId Default-Objekt → bool
is_in-snd : DatenId Default-Objekt → bool
is_in-fst : DatenId Entstanden-aus → bool
is_in : PatId-x-AufenthaltsId PatId Patientenbezug → bool
is_in : DatenId PatId-x-AufenthaltsId Aufenthaltsbezug → bool
est-Aktuelles-Stack-Objekt-von :
Aktuelles-Stack-Objekt-von DatenId Kennung → Aktuelles-Stack-Objekt-von
est-Stack-Objekt-von :
Stack-Objekt-von DatenId Kennung → Stack-Objekt-von
est-Global-fuer : Global-fuer DatenId Kennung → Global-fuer
est-Lokal-fuer : Lokal-fuer DatenId Kennung → Lokal-fuer
est-Entstanden-aus : Entstanden-aus DatenId DatenId → Entstanden-aus

est-Default-Objekt : Default-Objekt DatenId DatenId → Default-Objekt
rel-Lokal-fuer : Lokal-fuer DatenId Kennung → Lokal-fuer
rel-Stack-Objekt-von :
Stack-Objekt-von DatenId Kennung → Stack-Objekt-von
rel-Entstanden-aus : Entstanden-aus DatenId DatenId → Entstanden-aus
rel-Aktuelles-Stack-Objekt-von :
Aktuelles-Stack-Objekt-von DatenId Kennung → Aktuelles-Stack-Objekt-von
rel-Aufenthaltsbezug :
Aufenthaltsbezug DatenId PatId-x-AufenthaltsId → Aufenthaltsbezug
rel-Datenklassenbezug :
Datenklassenbezug DatenId Name → Datenklassenbezug
rel-Default-Objekt : Default-Objekt DatenId DatenId → Default-Objekt
get-Datenobjekt-coll : Entity-Db → Datenobjekt-coll
set-Datenobjekt-coll : Datenobjekt-coll Entity-Db → Entity-Db
get-Datenobjekt : DatenId Datenobjekt-coll → Sunrise-Datenobjekt
is-in-Datenobjekt-coll : Sunrise-Datenobjekt Datenobjekt-coll → bool
update-Datenobjekt :
Sunrise-Datenobjekt Datenobjekt-coll → Datenobjekt-coll
del : Sunrise-Datenobjekt Datenobjekt-coll → Datenobjekt-coll
set-Lokal-fuer : Lokal-fuer Relationship-Db → Relationship-Db
set-Global-fuer : Global-fuer Relationship-Db → Relationship-Db
set-Aktuelles-Stack-Objekt-von :
Aktuelles-Stack-Objekt-von Relationship-Db → Relationship-Db
set-Aufenthaltsbezug : Aufenthaltsbezug Relationship-Db → Relationship-Db
set-Datenklassenbezug :
Datenklassenbezug Relationship-Db → Relationship-Db
set-Stack-Objekt-von : Stack-Objekt-von Relationship-Db → Relationship-Db
set-Entstanden-aus : Entstanden-aus Relationship-Db → Relationship-Db
set-Default-Objekt : Default-Objekt Relationship-Db → Relationship-Db

CREATE
SORTS
OPNS
is-in-Stack-Objekt-von : Kennung Db → bool
is-in-Global-fuer : DatenId Kennung Db → bool
is-in-Global-fuer : DatenId Db → bool
is-in-Lokal-fuer : DatenId Kennung Db → bool
is-in-fst-Default-Objekt : DatenId Db → bool
is-in-snd-Default-Objekt : DatenId Db → bool
is-in-Stack-Objekt-von : DatenId Db → bool
is-in-Lokal-fuer : DatenId Db → bool
is-in-fst-Entstanden-aus : DatenId Db → bool
is-in-Patientenbezug : PatId-x-AufenthaltsId PatId Db → bool
is-in-Aufenthaltsbezug : DatenId PatId-x-AufenthaltsId Db → bool
get-Aktuelles-Stack-Objekt-von : Db → Aktuelles-Stack-Objekt-von

get-Default-Objekt : Db → Default-Objekt
get-Entstanden-aus : Db → Entstanden-aus
get-Stack-Objekt-von : Db → Stack-Objekt-von
get-Global-fuer : Db → Global-fuer
get-Lokal-fuer : Db → Lokal-fuer
get-Datenklassenbezug : Db → Datenklassenbezug
get-Aufenthaltsbezug : Db → Aufenthaltsbezug
est-Aktuelles-Stack-Objekt-von : Db DatenId Kennung → Db
est-Stack-Objekt-von : Db DatenId Kennung → Db
est-Global-fuer : Db DatenId Kennung → Db
est-Lokal-fuer : Db DatenId Kennung → Db
est-Entstanden-aus : Db DatenId DatenId → Db
est-Default-Objekt : Db DatenId DatenId → Db
rel-Default-Objekt : Db DatenId DatenId → Db
rel-Lokal-fuer : Db DatenId Kennung → Db
rel-Stack-Objekt-von : Db DatenId Kennung → Db
rel-Entstanden-aus : Db DatenId DatenId → Db
rel-Aktuelles-Stack-Objekt-von : Db DatenId Kennung → Db
rel-Datenklassenbezug : Db DatenId Name → Db ##
rel-Aufenthaltsbezug : Db DatenId PatId-x-AufenthaltsId → Db ##
get-Sunrise-Datenobjekt : DatenId Db → Sunrise-Datenobjekt
update-Sunrise-Datenobjekt : Sunrise-Datenobjekt Db → Db
del-Sunrise-Datenobjekt : Sunrise-Datenobjekt Db → Db
is-in-Db : Sunrise-Datenobjekt Db → bool

SEMANTICS

VARS

db : Db
k : Kennung
di,di' : DatenId
sd : Sunrise-Datenobjekt
asov : Aktuelles-Stack-Objekt-von
rdb : Relationship-Db
sov : Stack-Objekt-von
lf : Lokal-fuer
gf : Global-fuer
ea : Entstanden-aus
kb : Datenklassenbezug
ab : Aufenthaltsbezug
do : Default-Objekt
dc : Datenobjekt-coll
edb : Entity-Db
ai : PatId-x-AufenthaltsId
pi : PatId
kln : Name

PROGRAMS

```
is-in-Stack-Objekt-von(k,db) ←
  is_in(k,get-Stack-Objekt-von(get-Relationship-Db(db)));
is-in-Stack-Objekt-von(di,db) ←
  is_in(di,get-Stack-Objekt-von(get-Relationship-Db(db)));
is-in-Global-fuer(di,k,db) ←
  is_in(di,k,get-Global-fuer(get-Relationship-Db(db)));
is-in-Global-fuer(di,db) ←
  is_in(di,get-Global-fuer(get-Relationship-Db(db)));
is-in-Lokal-fuer(di,k,db) ←
  is_in(di,k,get-Lokal-fuer(get-Relationship-Db(db)));
is-in-Lokal-fuer(di,db) ←
  is_in(di,get-Lokal-fuer(get-Relationship-Db(db)));
is-in-fst-Default-Objekt(di,db) ←
  is_in-fst(di,get-Default-Objekt(get-Relationship-Db(db)));
is-in-snd-Default-Objekt(di,db) ←
  is_in-snd(di,get-Default-Objekt(get-Relationship-Db(db)));
is-in-fst-Entstanden-aus(di,db) ←
  is_in-fst(di,get-Entstanden-aus(get-Relationship-Db(db)));
is-in-Patientenbezug(ai,pi,db) ←
  is_in(ai,pi,get-Patientenbezug(get-Relationship-Db(db)));
is-in-Aufenthaltsbezug(di,ai,db) ←
  is_in(di,ai,get-Aufenthaltsbezug(get-Relationship-Db(db)));
get-Aktuelles-Stack-Objekt-von(db) ←
  get-Aktuelles-Stack-Objekt-von(get-Relationship-Db(db));
get-Default-Objekt(db) ← get-Default-Objekt(get-Relationship-Db(db));
get-Entstanden-aus(db) ← get-Entstanden-aus(get-Relationship-Db(db));
get-Stack-Objekt-von(db) ← get-Stack-Objekt-von(get-Relationship-Db(db));
get-Global-fuer(db) ← get-Global-fuer(get-Relationship-Db(db));
get-Lokal-fuer(db) ← get-Lokal-fuer(get-Relationship-Db(db));
get-Datenklassenbezug(db) ← get-Datenklassenbezug(get-Relationship-Db(db));
get-Aufenthaltsbezug(db) ← get-Aufenthaltsbezug(get-Relationship-Db(db));
est-Aktuelles-Stack-Objekt-von(db,di,k) ←
  LET asov : get-Aktuelles-Stack-Objekt-von(get-Relationship-Db(db)),
      rdb : get-Relationship-Db(db)
  IN  set-Relationship-Db(set-Aktuelles-Stack-Objekt-von(
      est-Aktuelles-Stack-Objekt-von(asov,di,k),rdb),db)
  TEL ;
est-Stack-Objekt-von(db,di,k) ←
  LET sov : get-Stack-Objekt-von(get-Relationship-Db(db)),
      rdb : get-Relationship-Db(db)
  IN  set-Relationship-Db(set-Stack-Objekt-von(
      est-Stack-Objekt-von(sov,di,k),rdb),db)
  TEL ;
```

```
est-Global-fuer(db,di,k) ←
   LET gf : get-Global-fuer(get-Relationship-Db(db)),
       rdb : get-Relationship-Db(db)
   IN  set-Relationship-Db(set-Global-fuer(
       est-Global-fuer(gf,di,k),rdb),db)
   TEL ;
est-Lokal-fuer(db,di,k) ←
   LET lf : get-Lokal-fuer(get-Relationship-Db(db)),
       rdb : get-Relationship-Db(db)
   IN  set-Relationship-Db(set-Lokal-fuer(
       est-Lokal-fuer(lf,di,k),rdb),db)
   TEL ;
est-Entstanden-aus(db,di,di') ←
   LET ea : get-Entstanden-aus(get-Relationship-Db(db)),
       rdb : get-Relationship-Db(db)
   IN  set-Relationship-Db(set-Entstanden-aus(
       est-Entstanden-aus(ea,di,di'),rdb),db)
   TEL ;
est-Default-Objekt(db,di,di') ←
   LET do : get-Default-Objekt(get-Relationship-Db(db)),
       rdb : get-Relationship-Db(db)
   IN  set-Relationship-Db(set-Default-Objekt(
       est-Default-Objekt(do,di,di'),rdb),db)
   TEL ;
rel-Lokal-fuer(db,di,k) ←
   LET lf : get-Lokal-fuer(get-Relationship-Db(db)),
       rdb : get-Relationship-Db(db)
   IN  set-Relationship-Db(set-Lokal-fuer(
       rel-Lokal-fuer(lf,di,k),rdb),db)
   TEL ;
rel-Stack-Objekt-von(db,di,k) ←
   LET sov : get-Stack-Objekt-von(get-Relationship-Db(db)),
       rdb : get-Relationship-Db(db)
   IN  set-Relationship-Db(set-Stack-Objekt-von(
       rel-Stack-Objekt-von(sov,di,k),rdb),db)
   TEL ;
rel-Entstanden-aus(db,di,di') ←
   LET ea : get-Entstanden-aus(get-Relationship-Db(db)),
       rdb : get-Relationship-Db(db)
   IN  set-Relationship-Db(set-Entstanden-aus(
       rel-Entstanden-aus(ea,di,di'),rdb),db)
   TEL ;
rel-Aktuelles-Stack-Objekt-von(db,di,k) ←
   LET asov : get-Aktuelles-Stack-Objekt-von(get-Relationship-Db(db)),
```

```
            rdb : get-Relationship-Db(db)
      IN    set-Relationship-Db(set-Aktuelles-Stack-Objekt-von(
            rel-Aktuelles-Stack-Objekt-von(asov,di,k),rdb),db)
      TEL ;
  rel-Datenklassenbezug(db,di,kln) ←
      LET kb : get-Datenklassenbezug(get-Relationship-Db(db)),
            rdb : get-Relationship-Db(db)
      IN    set-Relationship-Db(set-Datenklassenbezug(
            rel-Datenklassenbezug(kb,di,kln),rdb),db)
      TEL ;
  rel-Aufenthaltsbezug(db,di,ai) ←
      LET ab : get-Aufenthaltsbezug(get-Relationship-Db(db)),
            rdb : get-Relationship-Db(db)
      IN    set-Relationship-Db(set-Aufenthaltsbezug(
            rel-Aufenthaltsbezug(ab,di,ai),rdb),db)
      TEL ;
  rel-Default-Objekt(db,di,di') ←
      LET do : get-Default-Objekt(get-Relationship-Db(db)),
            rdb : get-Relationship-Db(db)
      IN    set-Relationship-Db(set-Default-Objekt(
            rel-Default-Objekt(do,di,di'),rdb),db)
      TEL ;
  get-Sunrise-Datenobjekt(di,db) ←
      get-Datenobjekt(di,get-Datenobjekt-coll(get-Entity-Db(db)));
  update-Sunrise-Datenobjekt(sd,db) ←
      LET dc : get-Datenobjekt-coll(get-Entity-Db(db)),
            edb : get-Entity-Db(db)
      IN    set-Entity-Db(set-Datenobjekt-coll(
            update-Datenobjekt(sd,dc),edb),db)
      TEL ;
  del-Sunrise-Datenobjekt(sd,db) ←
      LET dc : get-Datenobjekt-coll(get-Entity-Db(db)),
            edb : get-Entity-Db(db)
      IN    set-Entity-Db(set-Datenobjekt-coll(
            del(sd,dc),edb),db)
      TEL ;
  is-in-Db(sd,db) ←
      LET dc : get-Datenobjekt-coll(get-Entity-Db(db))
      IN    is-in-Datenobjekt-coll(sd,dc)
      TEL ;
ENDCREATE
ENDEXTENDED
```

Ende Modul DB

A.3.4 Zusammensetzen der Datenbankebene

Im Modul DATENBANKEBENE sollen die Bestandteile der Datenbankebene miteinander verknüpft werden.

Modul: DATENBANKEBENE
Autor: cb *Datum: 17. Januar 94*
Inhalt: Zusammensetzen der Bestandteile der Datenbankebene (dieser Modul kann nicht in der OBSCURE*-Moduldatenbank abgelegt werden und ist nicht Bestandteil der Gesamtspezifikation).*

INCLUDE DB
X_COMPOSE
(**INCLUDE** ENTITY-KOMPONENTE
PLUS INCLUDE RELATIONSHIP-KOMPONENTE)

Ende Modul DATENBANKEBENE

Anmerkung: Ein Parsen des Moduls DATENBANKEBENE war zwar noch möglich, nicht aber das Ablegen in der OBSCURE-Moduldatenbank. Deren Leistungsgrenze wurde durch die umfangreiche Signatur dieses Moduls überschritten. Deshalb wurde der Modul DATENBANKEBENE nicht in der OBSCURE-Moduldatenbank abgelegt, sondern dessen Spezifikationstext in dem Modul DATENMODELLEBENE (siehe Seite 149) anstelle von 'DATENBANKEBENE' eingesetzt. Das Ablegen des Moduls DATENMODELLEBENE in der OBSCURE-Moduldatenbank ist deshalb möglich, weil eine FORGET-Konstruktion dafür sorgt, daß viele weiter nicht benötigte Sorten und Operationen vergessen werden und somit die Signatur dieses Moduls deutlich verringert wird. Diese FORGET-Konstruktion am Ende von DATENMODELLEBENE ist nach [Aut93] nicht beabsichtigt, sondern erfolgt nur aufgrund der eingeschränkten Leistungsfähigkeit des OBSCURE-Systems. Natürlich hätte auf diese Weise auch schon für den Modul DATENBANKEBENE Abhilfe geschaffen werden können, doch sollte möglichst nur an einer Stelle eine solche Hilfskonstruktion eingeführt werden.

A.4 Spezifikation der schematischen Integritätsbedingungen

Im Modul IB-SCHEMATISCH werden Prädikate spezifiziert, die die Einhaltung der schematischen Integritätsbedingungen (siehe Seite 36) überprüfen. Zusammengefaßt werden diese zu einem Prädikat *OK-schematisch*, das die Gültigkeit aller schematischen Integritätsbedingungen für eine Datenbank abprüft. Nur *OK-schematisch* wird zur Formulierung des OK-Prädikats *OK* (siehe Modul OK-PRAEDIKAT auf Seite 63 ff.) weiter benötigt, alle anderen Prädikate können also vergessen werden.

Anmerkung: Die Unterscheidung zwischen schematischen Integritätsbedingungen und speziellen basiert auf der Ausdrucksmächtigkeit der verwendeten E/R-Modellierungstechnik. Man könnte sich sicherlich auch weitere Symbole zur E/R-Modellierung definieren, mit deren Hilfe die, hier als speziell eingestuften, Integritätsbedingungen (siehe Seite 38 bzw. 64) beschrieben werden könnten. Dann könnten diese möglicherweise auch bei der Transformation in OBSCURE-Spezifikationen berücksichtigt werden.

Modul: IB-SCHEMATISCH
Autor: cb *Datum: 25. Januar 94*
Inhalt: Spezifikation der schematischen Integritätsbedingungen.

IMPORTS
SORTS
Db Entity-Db Relationship-Db Root-coll Hospital-coll
Patient-coll Aufenthalt-coll Benutzer-coll
Datenobjekt-coll Datenklasse-coll Root Hospital Patient Aufenthalt
Benutzer Sunrise-Datenobjekt Datenklasse
Aufenthaltsbezug Patientenbezug Hospitalbezug Rootbezug Datenklassenbezug
Entstanden-aus Default-Objekt Aktuelles-Stack-Objekt-von Stack-Objekt-von
Global-fuer Lokal-fuer
DatenId AufenthaltsId PatId HospitalId PatId-x-AufenthaltsId
A-DatenId A-AufenthaltsId A-PatId A-HospitalId
OPNS
get-Entity-Db : Db → Entity-Db
get-Relationship-Db : Db → Relationship-Db
get-Root-coll : Entity-Db → Root-coll
get-Hospital-coll : Entity-Db → Hospital-coll
get-Patient-coll : Entity-Db → Patient-coll
get-Aufenthalt-coll : Entity-Db → Aufenthalt-coll
get-Benutzer-coll : Entity-Db → Benutzer-coll
get-Datenobjekt-coll : Entity-Db → Datenobjekt-coll
get-Datenklasse-coll : Entity-Db → Datenklasse-coll
e-Root-coll : → Root-coll
e-Hospital-coll : → Hospital-coll
e-Patient-coll : → Patient-coll
e-Aufenthalt-coll : → Aufenthalt-coll
e-Benutzer-coll : → Benutzer-coll
e-Datenobjekt-coll : → Datenobjekt-coll
e-Datenklasse-coll : → Datenklasse-coll
head : Root-coll → Root
tail : Root-coll → Root-coll
head : Hospital-coll → Hospital
tail : Hospital-coll → Hospital-coll

head : Patient-coll → Patient
tail : Patient-coll → Patient-coll
head : Aufenthalt-coll → Aufenthalt
tail : Aufenthalt-coll → Aufenthalt-coll
head : Benutzer-coll → Benutzer
tail : Benutzer-coll → Benutzer-coll
head : Datenobjekt-coll → Sunrise-Datenobjekt
tail : Datenobjekt-coll → Datenobjekt-coll
head : Datenklasse-coll → Datenklasse
tail : Datenklasse-coll → Datenklasse-coll
_ = _ : Root-coll Root-coll → bool
_ = _ : Hospital-coll Hospital-coll → bool
_ = _ : Patient-coll Patient-coll → bool
_ = _ : Aufenthalt-coll Aufenthalt-coll → bool
_ = _ : Benutzer-coll Benutzer-coll → bool
_ = _ : Datenobjekt-coll Datenobjekt-coll → bool
_ = _ : Datenklasse-coll Datenklasse-coll → bool
Root-RootId-ex : Root → bool
Hospital-HospitalId-ex : Hospital → bool
Patient-PatId-ex : Patient → bool
Patient-Name-ex : Patient → bool
Patient-Geschlecht-ex : Patient → bool
Patient-CreationInfo-ex : Patient → bool
Aufenthalt-PatId-ex : Aufenthalt → bool
Aufenthalt-AufenthaltsId-ex : Aufenthalt → bool
Aufenthalt-Aufenthaltsinfo-ex : Aufenthalt → bool
Aufenthalt-CreationInfo-ex : Aufenthalt → bool
Benutzer-Kennung-ex : Benutzer → bool
Sunrise-Datenobjekt-DatenId-ex : Sunrise-Datenobjekt → bool
Sunrise-Datenobjekt-Parameterliste-ex : Sunrise-Datenobjekt → bool
Sunrise-Datenobjekt-History-ex : Sunrise-Datenobjekt → bool
Sunrise-Datenobjekt-CreationInfo-ex : Sunrise-Datenobjekt → bool
Datenklasse-Klassenname-ex : Datenklasse → bool
get-Aufenthaltsbezug : Relationship-Db → Aufenthaltsbezug
get-Patientenbezug : Relationship-Db → Patientenbezug
get-Hospitalbezug : Relationship-Db → Hospitalbezug
get-Rootbezug : Relationship-Db → Rootbezug
get-Datenklassenbezug : Relationship-Db → Datenklassenbezug
get-Entstanden-aus : Relationship-Db → Entstanden-aus
get-Default-Objekt : Relationship-Db → Default-Objekt
get-Aktuelles-Stack-Objekt-von : Relationship-Db
→ Aktuelles-Stack-Objekt-von
get-Stack-Objekt-von : Relationship-Db → Stack-Objekt-von
get-Global-fuer : Relationship-Db → Global-fuer

get-Lokal-fuer : Relationship-Db → Lokal-fuer
DatenId : Sunrise-Datenobjekt → A-DatenId
˜˜ _ : A-DatenId → DatenId
AufenthaltsId : Aufenthalt → A-AufenthaltsId
˜˜ _ : A-AufenthaltsId → AufenthaltsId
HospitalId : Hospital → A-HospitalId
˜˜ _ : A-HospitalId → HospitalId
PatId : Aufenthalt → A-PatId
PatId : Patient → A-PatId
˜˜ _ : A-PatId → PatId
{_/_} : PatId AufenthaltsId → PatId-x-AufenthaltsId
is_in : DatenId Aufenthaltsbezug → bool
is_in : PatId-x-AufenthaltsId Patientenbezug → bool
is_in : PatId Hospitalbezug → bool
is_in : HospitalId Rootbezug → bool
is_in : DatenId Datenklassenbezug → bool
primary-key-Root : Root-coll → bool
primary-key-Hospital : Hospital-coll → bool
primary-key-Patient : Patient-coll → bool
primary-key-Aufenthalt : Aufenthalt-coll → bool
primary-key-Datenobjekt : Datenobjekt-coll → bool
primary-key-Benutzer : Benutzer-coll → bool
primary-key-Datenklasse : Datenklasse-coll → bool
Aufenthaltsbezug_N_1 : Aufenthaltsbezug → bool
Patientenbezug_N_1 : Patientenbezug → bool
Hospitalbezug_N_1 : Hospitalbezug → bool
Rootbezug_N_1 : Rootbezug → bool
Datenklassenbezug_N_1 : Datenklassenbezug → bool
Entstanden-aus_N_1 : Entstanden-aus → bool
Default-Objekt_1_1 : Default-Objekt → bool
Aktuelles-Stack-Objekt-von_1_1 : Aktuelles-Stack-Objekt-von → bool
Stack-Objekt-von_N_1 : Stack-Objekt-von → bool
Global-fuer_N_1 : Global-fuer → bool
Lokal-fuer_N_1 : Lokal-fuer → bool

CREATE

OPNS

OK-schematisch	: Db	→ bool
mand-Root	: Root-coll	→ bool
mand-Hospital	: Hospital-coll	→ bool
mand-Patient	: Patient-coll	→ bool
mand-Aufenthalt	: Aufenthalt-coll	→ bool
mand-Datenobjekt	: Datenobjekt-coll	→ bool
mand-Benutzer	: Benutzer-coll	→ bool
mand-Datenklasse	: Datenklasse-coll	→ bool

zw-part-Aufenthaltsbezug : Datenobjekt-coll Aufenthaltsbezug → bool
zw-part-Patientenbezug : Aufenthalt-coll Patientenbezug → bool
zw-part-Hospitalbezug : Patient-coll Hospitalbezug → bool
zw-part-Rootbezug : Hospital-coll Rootbezug → bool
zw-part-Datenklassenbezug : Datenobjekt-coll Datenklassenbezug → bool

SEMANTICS

VARS

db : Db
edb : Entity-Db
rdb : Relationship-Db
rc : Root-coll
hc : Hospital-coll
pc : Patient-coll
ac : Aufenthalt-coll
bc : Benutzer-coll
dc : Datenobjekt-coll
kc : Datenklasse-coll
ab : Aufenthaltsbezug
pb : Patientenbezug
hb : Hospitalbezug
rb : Rootbezug
kb : Datenklassenbezug

PROGRAMS

OK-schematisch(db) ←
LET edb : get-Entity-Db(db),
rdb : get-Relationship-Db(db)
IN mand-Root(get-Root-coll(edb)) and
mand-Hospital(get-Hospital-coll(edb)) and
mand-Patient(get-Patient-coll(edb)) and
mand-Aufenthalt(get-Aufenthalt-coll(edb)) and
mand-Datenobjekt(get-Datenobjekt-coll(edb)) and
mand-Benutzer(get-Benutzer-coll(edb)) and
mand-Datenklasse(get-Datenklasse-coll(edb)) and
primary-key-Root(get-Root-coll(edb)) and
primary-key-Hospital(get-Hospital-coll(edb)) and
primary-key-Patient(get-Patient-coll(edb)) and
primary-key-Aufenthalt(get-Aufenthalt-coll(edb)) and
primary-key-Datenobjekt(get-Datenobjekt-coll(edb)) and
primary-key-Benutzer(get-Benutzer-coll(edb)) and
primary-key-Datenklasse(get-Datenklasse-coll(edb)) and
Aufenthaltsbezug_N_1(get-Aufenthaltsbezug(rdb)) and
Patientenbezug_N_1(get-Patientenbezug(rdb)) and
Hospitalbezug_N_1(get-Hospitalbezug(rdb)) and
Rootbezug_N_1(get-Rootbezug(rdb)) and

```
        Datenklassenbezug_N_1(get-Datenklassenbezug(rdb))                and
        Entstanden-aus_N_1(get-Entstanden-aus(rdb))                      and
        Default-Objekt_1_1(get-Default-Objekt(rdb))                      and
        Aktuelles-Stack-Objekt-von_1_1(
         get-Aktuelles-Stack-Objekt-von(rdb))                            and
        Stack-Objekt-von_N_1(get-Stack-Objekt-von(rdb))                  and
        Global-fuer_N_1(get-Global-fuer(rdb))                            and
        Lokal-fuer_N_1(get-Lokal-fuer(rdb))                              and
        zw-part-Aufenthaltsbezug(get-Datenobjekt-coll(edb),
         get-Aufenthaltsbezug(rdb))                                      and
        zw-part-Patientenbezug(get-Aufenthalt-coll(edb),
         get-Patientenbezug(rdb))                                        and
        zw-part-Hospitalbezug(get-Patient-coll(edb),
         get-Hospitalbezug(rdb))                                         and
        zw-part-Rootbezug(get-Hospital-coll(edb),
         get-Rootbezug(rdb))                                             and
        zw-part-Datenklassenbezug(get-Datenobjekt-coll(edb),
         get-Datenklassenbezug(rdb))
 TEL ;
mand-Root(rc) ←          IF rc = e-Root-coll THEN true
                         ELSE
                          Root-RootId-ex(head(rc)) and
                          mand-Root(tail(rc))
                         FI ;
mand-Hospital(hc) ←      IF hc = e-Hospital-coll THEN true
                         ELSE
                          Hospital-HospitalId-ex(head(hc)) and
                          mand-Hospital(tail(hc))
                         FI ;
mand-Patient(pc) ←       IF pc = e-Patient-coll THEN true
                         ELSE
                          Patient-PatId-ex(head(pc)) and
                          Patient-Name-ex(head(pc)) and
                          Patient-Geschlecht-ex(head(pc)) and
                          Patient-CreationInfo-ex(head(pc)) and
                          mand-Patient(tail(pc))
                         FI ;
mand-Aufenthalt(ac) ←    IF ac = e-Aufenthalt-coll THEN true
                         ELSE
                          Aufenthalt-PatId-ex(head(ac)) and
                          Aufenthalt-AufenthaltsId-ex(head(ac)) and
                          Aufenthalt-Aufenthaltsinfo-ex(head(ac)) and
                          Aufenthalt-CreationInfo-ex(head(ac)) and
                          mand-Aufenthalt(tail(ac))
```

```
                              FI ;
mand-Benutzer(bc) ←           IF bc = e-Benutzer-coll THEN true
                              ELSE
                               Benutzer-Kennung-ex(head(bc)) and
                               mand-Benutzer(tail(bc))
                              FI ;
mand-Datenobjekt(dc) ←        IF dc = e-Datenobjekt-coll THEN true
                              ELSE
                               Sunrise-Datenobjekt-DatenId-ex(head(dc)) and
                               Sunrise-Datenobjekt-Parameterliste-ex(head(dc)) and
                               Sunrise-Datenobjekt-History-ex(head(dc)) and
                               Sunrise-Datenobjekt-CreationInfo-ex(head(dc)) and
                               mand-Datenobjekt(tail(dc))
                              FI ;
mand-Datenklasse(kc) ←        IF kc = e-Datenklasse-coll THEN true
                              ELSE
                               Datenklasse-Klassenname-ex(head(kc)) and
                               mand-Datenklasse(tail(kc))
                              FI ;
zw-part-Aufenthaltsbezug(dc,ab) ←
                              IF dc = e-Datenobjekt-coll THEN true
                              ELSE
                               is_in(~~ DatenId(head(dc)),ab) and
                               zw-part-Aufenthaltsbezug(tail(dc),ab)
                              FI ;
zw-part-Patientenbezug(ac,pb) ←
                              IF ac = e-Aufenthalt-coll THEN true
                              ELSE
                               is_in({(~~ PatId(head(ac)))/
                                       (~~ AufenthaltsId(head(ac)))},pb) and
                               zw-part-Patientenbezug(tail(ac),pb)
                              FI ;
zw-part-Hospitalbezug(pc,hb) ←
                              IF pc = e-Patient-coll THEN true
                              ELSE
                               is_in(~~ PatId(head(pc)),hb) and
                               zw-part-Hospitalbezug(tail(pc),hb)
                              FI ;
zw-part-Rootbezug(hc,rb) ←
                              IF hc = e-Hospital-coll THEN true
                              ELSE
                               is_in(~~ HospitalId(head(hc)),rb) and
                               zw-part-Rootbezug(tail(hc),rb)
                              FI ;
```

```
  zw-part-Datenklassenbezug(dc,kb) ←
                              IF dc = e-Datenobjekt-coll THEN true
                              ELSE
                               is_in(~~ DatenId(head(dc)),kb) and
                               zw-part-Datenklassenbezug(tail(dc),kb)
                              FI ;
ENDCREATE
```

Ende Modul IB-SCHEMATISCH

A.5 Zusammensetzen der Datenmodellebene

Die Einzelbestandteile der Datenmodellebene sind nun komplett und können gemäß Abbildung A.1 zusammengesetzt werden. Dies erfolgt im Modul DATENMODELLEBENE.

Es soll an dieser Stelle nochmals darauf hingewiesen werden, daß die Einzelbestandteile DB, ENTITY-KOMPONENTE und RELATIONSHIP-KOMPONENTE nur deshalb nicht zu einem Modul DATENBANKEBENE zusammengefaßt werden konnten, weil die OBSCURE-Moduldatenbank die umfangreiche Signatur dieses Moduls nicht aufnehmen konnte. Damit das Zusammenfügen aller Teile im Modul DATENMODELLEBENE möglich wird, müssen am Ende dieses Moduls sämtliche, von der Spezifikation der Aktivitätenebene nicht benötigte Operationen und Sorten vergessen werden. Nur durch diese große FORGET-Konstruktion wurde ein Ablegen des Moduls in der OBSCURE-Moduldatenbank möglich.

Am Ende des Moduls DATENMODELLEBENE werden außerdem einige sehr unleserliche Sortennamen umbenannt.

Modul: DATENMODELLEBENE
Autor: cb *Datum: 26. Januar 94*
Inhalt: Zusammensetzen der Datenmodellebene.

```
(
(
INCLUDE IB-SCHEMATISCH
X_COMPOSE
INCLUDE DB
X_COMPOSE
( INCLUDE ENTITY-KOMPONENTE
PLUS INCLUDE RELATIONSHIP-KOMPONENTE )
X_COMPOSE
INCLUDE ATTRIBUTEBENE
)
FORGET_ALL_BUT
```

OPNS

reachable : Kennung Aktuelles-Stack-Objekt-von →
SoDatenId-Aktuelles-Stack-Objekt-von
head : SoDatenId-Aktuelles-Stack-Objekt-von → DatenId
reachable : DatenId Default-Objekt → S1oDatenId-Default-Objekt
head : S1oDatenId-Default-Objekt → DatenId
reachable : DatenId Entstanden-aus → S1oDatenId-Entstanden-aus
tail : S1oDatenId-Entstanden-aus → S1oDatenId-Entstanden-aus
head : S1oDatenId-Entstanden-aus → DatenId
reachable_from : DatenId Datenklassenbezug →
SoKlassenname-Datenklassenbezug
head : SoKlassenname-Datenklassenbezug → Name
reachable_from : DatenId Aufenthaltsbezug →
SoPatId-x-AufenthaltsId-Aufenthaltsbezug
head : SoPatId-x-AufenthaltsId-Aufenthaltsbezug → PatId-x-AufenthaltsId
e-S1oDatenId-Entstanden-aus : → S1oDatenId-Entstanden-aus
_ = _ : S1oDatenId-Entstanden-aus S1oDatenId-Entstanden-aus → bool
reachable_from : DatenId Entstanden-aus → S2oDatenId-Entstanden-aus
head : S2oDatenId-Entstanden-aus → DatenId
tail : S2oDatenId-Entstanden-aus → S2oDatenId-Entstanden-aus
Datenname: Sunrise-Datenobjekt → A-Name
History : Sunrise-Datenobjekt → A-History
DatenId : Sunrise-Datenobjekt → A-DatenId
head : Datenobjekt-coll → Sunrise-Datenobjekt
tail : Datenobjekt-coll → Datenobjekt-coll
e-Datenobjekt-coll : → Datenobjekt-coll
set-Datenname: A-Name Sunrise-Datenobjekt → Sunrise-Datenobjekt
_ = _ : A-Name A-Name → bool
˜ _ : Name → A-Name
˜˜ _ : A-DatenId → DatenId
˜˜ _ : A-History → History
_ = _ : DatenId DatenId → bool
_ = _ : Db Db → bool
_ = _ : Datenobjekt-coll , Datenobjekt-coll → bool
get-Entity-Db : Db → Entity-Db
set-Entity-Db : Entity-Db Db → Db
get-Datenobjekt-coll : Entity-Db → Datenobjekt-coll
OK-schematisch : Db → bool
is-in-Stack-Objekt-von : Kennung Db → bool
is-in-Stack-Objekt-von : DatenId Db → bool
is-in-Global-fuer : DatenId Kennung Db → bool
is-in-Global-fuer : DatenId Db → bool
is-in-Lokal-fuer : DatenId Kennung Db → bool
is-in-Lokal-fuer : DatenId , Db → bool

is-in-fst-Entstanden-aus : DatenId Db → bool
is-in-fst-Default-Objekt : DatenId Db → bool
is-in-snd-Default-Objekt : DatenId Db → bool
is-in-Aufenthaltsbezug : DatenId PatId-x-AufenthaltsId Db → bool
is-in-Patientenbezug : PatId-x-AufenthaltsId PatId Db → bool
is-in-Db : Sunrise-Datenobjekt Db → bool
get-Aktuelles-Stack-Objekt-von : Db → Aktuelles-Stack-Objekt-von
get-Default-Objekt : Db → Default-Objekt
get-Entstanden-aus : Db → Entstanden-aus
get-Datenklassenbezug : Db → Datenklassenbezug
get-Aufenthaltsbezug : Db → Aufenthaltsbezug
est-Global-fuer : Db DatenId Kennung → Db
est-Lokal-fuer : Db DatenId Kennung → Db
est-Entstanden-aus : Db DatenId DatenId → Db
est-Aktuelles-Stack-Objekt-von : Db DatenId Kennung → Db
est-Stack-Objekt-von : Db DatenId Kennung → Db
est-Default-Objekt : Db DatenId DatenId → Db
rel-Default-Objekt : Db DatenId DatenId → Db
rel-Lokal-fuer : Db DatenId Kennung → Db
rel-Aktuelles-Stack-Objekt-von : Db DatenId Kennung → Db
rel-Stack-Objekt-von : Db DatenId Kennung → Db
rel-Entstanden-aus : Db DatenId DatenId → Db
rel-Datenklassenbezug : Db DatenId Name → Db
rel-Aufenthaltsbezug : Db DatenId PatId-x-AufenthaltsId → Db
get-Sunrise-Datenobjekt : DatenId Db → Sunrise-Datenobjekt
update-Sunrise-Datenobjekt : Sunrise-Datenobjekt Db → Db
del-Sunrise-Datenobjekt : Sunrise-Datenobjekt Db → Db

Für die Interpretersitzung — d. h. zum Aufbau einer Ausgangsdatenbank — werden zusätzlich die folgenden Operationen benötigt.

create-Root : A-RootId → Root
create-Hospital : A-HospitalId A-Name A-Adresse A-CreationInfo → Hospital
create-Benutzer : A-Name A-Name A-Adresse A-Datum A-Kennung → Benutzer
create-Sunrise-Datenobjekt : A-DatenId A-Name A-Parameterliste
A-History A-Binaerdaten A-Roi A-CreationInfo → Sunrise-Datenobjekt
create-Datenklasse : A-Name → Datenklasse
create-Patient : A-PatId A-Name A-Name A-Name A-Datum A-Ort A-Geschlecht
A-CreationInfo → Patient
create-Aufenthalt : A-PatId A-AufenthaltsId A-Familienstand
A-Nationalitaet A-Beruf A-Religion A-Diagnose A-Adresse
A-Aufenthaltsinfo A-CreationInfo → Aufenthalt
˜ _ : Kennung → A-Kennung
˜ _ : Beruf → A-Beruf
˜ _ : Diagnose → A-Diagnose

˜ _ : Parameterliste → A-Parameterliste
˜ _ : Addresse → A-Adresse
˜ _ : CreationInfo → A-CreationInfo
˜ _ : Aufenthaltsinfo → A-Aufenthaltsinfo
˜ _ : History → A-History
˜ _ : RootId → A-RootId
˜ _ : HospitalId → A-HospitalId
˜ _ : PatId → A-PatId
˜ _ : Familienstand → A-Familienstand
˜ _ : AufenthaltsId → A-AufenthaltsId
˜ _ : DatenId → A-DatenId
˜ _ : Religion → A-Religion
˜ _ : Binaerdaten → A-Binaerdaten
˜ _ : Datum → A-Datum
˜ _ : Geschlecht → A-Geschlecht
˜ _ : Ort → A-Ort
˜ _ : Nationalitaet → A-Nationalitaet
˜ _ : Roi → A-Roi
religion _ : integer → Religion
creationinfo _ : integer → CreationInfo
nation _ : integer → Nationalitaet
stand _ : integer → Familienstand
adresse _ : integer → Addresse
aufinfo _ : integer → Aufenthaltsinfo
roi _ : integer → Roi
geschl _ : integer → Geschlecht
aufid _ : integer → AufenthaltsId
paraliste _ : integer → Parameterliste
ort _ : integer → Ort
diagn _ : integer → Diagnose
rootid _ : integer → RootId
datenid _ : integer → DatenId
hospid _ : integer → HospitalId
bindaten _ : integer → Binaerdaten
kennung _ : integer → Kennung
beruf _ : integer → Beruf
name _ : integer → Name
datum _ : integer → Datum
patid _ : integer → PatId
history _ : integer → History
e-a-kennung : → A-Kennung
e-a-beruf : → A-Beruf
e-a-diagn : → A-Diagnose
e-a-paraliste : → A-Parameterliste

e-a-adresse : → A-Adresse
e-a-name : → A-Name
e-a-creationinfo : → A-CreationInfo
e-a-aufinfo : → A-Aufenthaltsinfo
e-a-history : → A-History
e-a-rootid : → A-RootId
e-a-hospid : → A-HospitalId
e-a-patid : → A-PatId
e-a-stand : → A-Familienstand
e-a-aufid : → A-AufenthaltsId
e-a-datenid : → A-DatenId
e-a-religion : → A-Religion
e-a-bindaten : → A-Binaerdaten
e-a-datum : → A-Datum
e-a-geschl : → A-Geschlecht
e-a-ort : → A-Ort
e-a-nation : → A-Nationalitaet
e-a-roi : → A-Roi
get-Root-coll : Entity-Db → Root-coll
get-Hospital-coll : Entity-Db → Hospital-coll
get-Patient-coll : Entity-Db → Patient-coll
get-Aufenthalt-coll : Entity-Db → Aufenthalt-coll
get-Benutzer-coll : Entity-Db → Benutzer-coll
get-Datenklasse-coll : Entity-Db → Datenklasse-coll
get-Aufenthaltsbezug : Relationship-Db → Aufenthaltsbezug
get-Datenklassenbezug : Relationship-Db → Datenklassenbezug
get-Patientenbezug : Relationship-Db → Patientenbezug
get-Stack-Objekt-von : Relationship-Db → Stack-Objekt-von
get-Lokal-fuer : Relationship-Db → Lokal-fuer
get-Rootbezug : Relationship-Db → Rootbezug
get-Hospitalbezug : Relationship-Db → Hospitalbezug
get-Global-fuer : Relationship-Db → Global-fuer
est-Global-fuer : Global-fuer DatenId Kennung → Global-fuer
est-Lokal-fuer : Lokal-fuer DatenId Kennung → Lokal-fuer
est-Entstanden-aus : Entstanden-aus DatenId DatenId → Entstanden-aus
est-Aktuelles-Stack-Objekt-von : Aktuelles-Stack-Objekt-von DatenId
Kennung → Aktuelles-Stack-Objekt-von
est-Stack-Objekt-von : Stack-Objekt-von DatenId Kennung → Stack-Objekt-von
est-Aufenthaltsbezug : Aufenthaltsbezug DatenId
PatId-x-AufenthaltsId → Aufenthaltsbezug
est-Datenklassenbezug : Datenklassenbezug DatenId Name → Datenklassenbezug
est-Patientenbezug : Patientenbezug PatId-x-AufenthaltsId PatId → Patientenbezug
est-Default-Objekt : Default-Objekt DatenId DatenId → Default-Objekt
est-Hospitalbezug : Hospitalbezug PatId HospitalId → Hospitalbezug

est-Rootbezug : Rootbezug HospitalId RootId → Rootbezug
ins : Benutzer Benutzer-coll → Benutzer-coll
ins : Sunrise-Datenobjekt Datenobjekt-coll → Datenobjekt-coll
ins : Datenklasse Datenklasse-coll → Datenklasse-coll
ins : Hospital Hospital-coll → Hospital-coll
ins : Patient Patient-coll → Patient-coll
ins : Root Root-coll → Root-coll
ins : Aufenthalt Aufenthalt-coll → Aufenthalt-coll
{_/_} : PatId , AufenthaltsId → PatId-x-AufenthaltsId
{_/_/_/_/_/_/_}: Benutzer-coll Datenklasse-coll Datenobjekt-coll
Aufenthalt-coll Patient-coll Hospital-coll Root-coll → Entity-Db
{_/_/_/_/_/_/_/_/_/_/_/_} : Rootbezug
Hospitalbezug Patientenbezug Aufenthaltsbezug Lokal-fuer
Global-fuer Stack-Objekt-von Aktuelles-Stack-Objekt-von
Datenklassenbezug Default-Objekt Entstanden-aus → Relationship-Db
{_/_} : Entity-Db Relationship-Db → Db
e-Db : → Db
get-Relationship-Db : Db → Relationship-Db
)
E_RENAME
SORTS SoDatenId-Aktuelles-Stack-Objekt-von S1oDatenId-Default-Objekt
S1oDatenId-Entstanden-aus S2oDatenId-Entstanden-aus
SoPatId-x-AufenthaltsId-Aufenthaltsbezug
SoKlassenname-Datenklassenbezug
OPNS e-S1oDatenId-Entstanden-aus **
AS
SORTS S1oDatenId S2oDatenId S3oDatenId S4oDatenId
SoPatId-x-AufenthaltsId SoName

OPNS e-S3oDatenId

Ende Modul DATENMODELLEBENE

A.6 Standardmodule

Von dem OBSCURE-System werden jedem Systembenutzer Standardmodule zur Verfügung gestellt. In dieser Spezifikation werden folgende Standardmodule verwendet:

- TUPEL-n : TUPEL-Spezifikation für unterschiedliche n, siehe Abschnitt A.6.1.
- LIST : Spezifikation von Listen, siehe Abschnitt A.6.2.

- MLIST : Spzifikation von Monolisten. Monolisten sind Listen ohne doppeltes Auftreten von Elementen. In dieser Arbeit werden sie anstelle von Mengen verwendet. Die Gründe für die Verwendung von Monolisten — diese wurden schon in der Saarbrücker Spezifikation zum KORSO-HDMS-A Fallbeispiel (siehe [Hec93], [Aut93] und [Ben93c]) anstelle von Mengen verwendet —, werden in [Hec93], Seite 45 ff. diskutiert. In Abschnitt A.6.3 wird die Spezifikation von Monolisten vorgestellt.

- MAKE_BINARY_RELATION : Spezifikation binärer Relationen. Diese Spezifikation ist relativ umfangreich und wird deshalb nicht vorgestellt. Als Spezialfall (wegen des Sortenclashproblems) wird ferner die Spezifikation MK_SPECIAL_BINARY_RELATION verwendet, auch dieser Modul wird nicht vorgestellt. Die Signatur des Moduls MAKE_BINARY_RELATION wurde bereits auf Seite 128 aufgelistet.

Im folgenden werden die Standardmodule TUPEL-2, MLIST und LIST präsentiert.

A.6.1 Tupelspezifikationen

Mehrmals wird in der Gesamtspezifikation auf Tupelmodule (Module TUPEL-n) zurückgegriffen. Konkret benötigt werden Tupelmodule in :

- Modul KEY (siehe Seite 119)
- Modul MK_n-ATTR-ENTITY, für n=1..10 (siehe Seite 120)
- Modul ENTITY-KOMPONENTE (siehe Seite 126)
- Modul MAKE_BINARY_RELATION (wird hier nicht vorgestellt)
- Modul RELATIONSHIP-KOMPONENTE (siehe Seite 133)
- Modul RAW_DB (siehe Seite 135)
- Modul SONDERMODUL (siehe Seite 62)

Das OBSCURE-System stellt dem Spezifizierer Tupelspezifikationen in der Standardmoduldatenbank bereit. Konkret existieren dort Module TUPEL-n für n=1..10 . Auf diese kann also in dieser Spezifikation zurückgegriffen werden.

Modul: TUPEL-2
Autor: Standardmodul
Inhalt: Spezifikation von Paaren (2-Tupel).

IMPORTS
SORTS
Sort1
Sort2

```
 OPNS
   _ = _ : Sort1 Sort1 → bool
   _ = _ : Sort2 Sort2 → bool
CREATE
 SORTS
   Zwei_Tupel,
 OPNS
   {_/_}      : Sort1 Sort2              → Zwei_Tupel
   get_fst    : Zwei_Tupel               → Sort1
   get_snd    : Zwei_Tupel               → Sort2
   set_fst    : Sort1 Zwei_Tupel         → Zwei_Tupel
   set_snd    : Sort2 Zwei_Tupel         → Zwei_Tupel
   _ = _      : Zwei_Tupel Zwei_Tupel    → bool
SEMANTICS
 CONSTRS
   {_/_}      : Sort1 Sort2              → Zwei_Tupel
 VARS
   var_Zwei_T upel : Zwei_Tupel
   var_Sort1, var1_Sort1 : Sort1
   var_Sort2, var1_Sort2 : Sort2
 PROGRAMS
   get_fst(var_Zwei_Tupel) ←
       CASE var_Zwei_Tupel OF
               {var_Sort1/var_Sort2} : var_Sort1;
       ESAC ;

   get_snd(var_Zwei_Tupel) ←
       CASE var_Zwei_Tupel OF
               {var_Sort1/var_Sort2} : var_Sort2;
       ESAC ;

   set_fst(var_Sort1,var_Zwei_Tupel ) ←
       CASE var_Zwei_Tupel OF
               {var1_Sort1/var_Sort2} : {var_Sort1/var_Sort2};
       ESAC ;

   set_snd(var_Sort2,var_Zwei_Tupel ) ←
       CASE var_Zwei_Tupel OF
               {var_Sort1/var1_Sort2} : {var_Sort1/var_Sort2};
       ESAC ;

ENDCREATE

PARAMS
( SORTS Sort1 , Sort2 )
[ SORTS Zwei_Tupel
```

OPNS{_/_} **,**
get_fst **,**
get_snd **,**
set_fst **,**
set_snd **]

Ende Modul TUPEL-2

A.6.2 Listen

Durch den parametrisierten Modul LIST werden Listen über einer beliebigen Elementsorte spezifiziert.

Modul: LIST
Autor: Standardmodul
Inhalt: Spezifikation von Listen.

IMPORTS
SORTS El
OPNS _ = _ : El El → bool
CREATE
SORTS List
OPNS
e : → List
_ ^ _ : El List → List
_ ^ _ : List El → List
_ ^ _ : List List → List
l : El → List
head : List → El
tail : List → List
last : List → El
del : El List → List
allbutlast : List → List
_ is_prefix_of _ : List List → bool
_ is-in _ : El List → bool
_ = _ : List List → bool
SEMANTICS
CONSTRS
e : → List
_ ^ _ : El List → List
VARS
el, el', el1, el2 : El
list, list', list", list1, list2, list1', list2' : List

PROGRAMS

```
list ^ el ← list ^ (el ^ e) ;
l(el) ← el ^ e ;
list1 ^ list2 ←     CASE list1 OF
                      e : list2 ;
                      el ^ list': el ^ (list' ^ list2) ;
                    ESAC ;
head(list) ←        CASE list OF
                      e : ERROR(El) ;
                      el ^ list': el ;
                     ESAC ;
tail(list) ←        CASE list OF
                      e : e ;
                      el ^ list': list' ;
                    ESAC ;
last(list) ←        CASE list OF
                      e : ERROR(El) ;
                      el ^ list':     CASE list' OF e : el ;
                                      ELSE last(list')
                                      ESAC ;
                    ESAC ;
del(el,list) ←      CASE list OF
                      e : e ;
                      el1 ^ list':
                        IF el = el1
                        THEN list'
                        ELSE el1 ^ (del(el,list'))
                        FI
                    ESAC ;
allbutlast(list) ←
                    CASE list OF
                      e : e ;
                      el ^ list':
                        CASE list' OF
                          e : e ;
                          el' ^ list":
                            CASE list" OF e : el ^ e ;
                            ELSE el ^ (el' ^ allbutlast(list"))
                            ESAC
                        ESAC ;
                    ESAC ;
list1 is_prefix_of list2 ←
                    CASE list1 OF
                      e : true ;
```

```
                        el1 ^ list1':
                          CASE list2 OF
                            e : false ;
                            el2 ^ list2' :
                              (el1 = el2) and (list1' is_prefix_of list2') ;
                          ESAC ;
                      ESAC ;
el is-in list ←
                      CASE list OF
                        e : false ;
                        el' ^ list':
                          IF el' = el
                          THEN true
                          ELSE el is-in list'
                          FI ;
                      ESAC ;
```

ENDCREATE

PARAMS
(**SORTS** El)
[**SORTS** List **OPNS** e : → List, l : El → List]

Ende Modul LIST

A.6.3 Monolisten

Mit Hilfe des parametrisierten Moduls MLIST können Monolisten über einem beliebigem Elementsorte spezifiziert werden. Weil Monolisten anstelle von Mengen verwendet werden, stellt die Spezifikation insbesondere solche Operationen bereit, die man typischerweise über Mengen erwarten würde. Ferner existieren typische Listen-Operationen wie z. B. *head* und *tail*. Ein wesentlicher Aspekt der Verwendung von Monolisten ist, daß die *choose*-Operation leicht über die *head*-Operation definiert werden kann. Bei *normalen* Mengen kann die *choose*-Operation nicht ohne weiteres spezifiziert werden.

Modul: MLIST
Autor: Standardmodul
Inhalt: Spezifikation von Monolisten (Listen ohne doppeltes Auftreten von Elementen) über einer beliebigen Elementsorte.

IMPORTS
SORTS
El

OPNS

_ = _	: El El	→ bool

CREATE

SORTS

MList

OPNS

empty-mlist	:	→ MList
_ ^ _	: El MList	→ MList
_ = _	: MList MList	→ bool
mk-mlist	: MList	→ MList
mlist	: El	→ MList
_ is-in _	: El MList	→ bool
ins	: El MList	→ MList
del	: El MList	→ MList
head	: MList	→ El
tail	: MList	→ MList
last	: MList	→ El
allbutlast	: MList	→ MList
_ is_prefix_of _	: MList MList	→ bool
choose	: MList	→ El
#	: MList	→ integer
_ u _	: MList MList	→ MList
_ n _	: MList MList	→ MList
_ c _	: MList MList	→ bool
_ - _	: MList MList	→ MList

SEMANTICS

CONSTRS

empty-mlist	:	→ MList
_ ^ _	: El MList	→ MList

VARS

el, el', el1, el2 : El

list, list', list", list1, list2, list1', list2' : MList

PROGRAMS

```
mk-mlist(list)   ← IF list = empty-mlist
                   THEN list
                   ELSE
                     IF head(list) is-in tail(list)
                     THEN head(list) ^ mk-mlist(del(head(list),tail(list)))
                     ELSE head(list) ^ mk-mlist(tail(list))
                     FI
                   FI ;
mlist(el)        ← el ^ empty-mlist;
el is-in list    ← CASE list OF
```

```
                                empty-mlist: false;
                                el' ^ list': (el = el') or (el is-in list')
                        ESAC ;
ins(el,list)        ← IF el is-in list
                        THEN list
                        ELSE el ^ list
                        FI;
del(el,list)        ← CASE list OF
                                empty-mlist: empty-mlist;
                                el1 ^ list':
                                    IF el = el1
                                    THEN list'
                                    ELSE el1 ^ (del(el,list'))
                                    FI
                        ESAC ;
head(list)          ← CASE list OF
                                empty-mlist: ERROR(El);
                                el ^ list': el;
                        ESAC ;
tail(list)          ← CASE list OF
                                empty-mlist: empty-mlist;
                                el ^ list': list';
                        ESAC ;
last(list)          ← CASE list OF
                                empty-mlist: ERROR(El);
                                el ^ list':     CASE list' OF
                                                        empty-mlist: el;
                                                ELSE    last(list')
                                                ESAC ;
                        ESAC ;
allbutlast(list) ←
    CASE list OF
            empty-mlist: empty-mlist;
            el ^ list':
                CASE list' OF
                        empty-mlist: empty-mlist;
                        el' ^ list":
                            CASE list" OF
                                    empty-mlist: el ^ empty-mlist;
                            ELSE  el ^ (el' ^ allbutlast(list"))
                            ESAC
                ESAC ;
    ESAC ;
list1 is_prefix_of list2 ←
```

```
    CASE list1 OF
            empty-mlist: true;
            el1 ^ list1':
                CASE list2 OF
                        empty-mlist: false;
                        el2 ^ list2' : (el1 = el2) and
                        (list1' is_prefix_of list2');
                ESAC ;
    ESAC ;
choose(list)        ← head(list);
#(list)             ← CASE list OF
                            empty-mlist: 0;
                            el ^ list': 1 + #(list')
                      ESAC ;
list u list'        ← CASE list OF
                            empty-mlist: list';
                            el ^ list": list" u (ins(el,list'))
                      ESAC ;
list n list'        ← CASE list OF
                            empty-mlist: empty-mlist;
                            el ^ list":     IF el is-in list'
                                            THEN el ^ (list" n list')
                                            ELSE list" n list'
                                            FI
                      ESAC ;
list c list'        ← list = (list n list');
list - list'        ← CASE list OF
                            empty-mlist: empty-mlist;
                            el ^ list":     IF el is-in list'
                                            THEN list" - list'
                                            ELSE el ^ (list" - list')
                                            FI
                      ESAC ;

ENDCREATE
FORGET
 OPNS _ ^ _ : El MList → MList

SUBSET OF MList BY
VARS list : MList ;
     (mk-mlist(list) = list) == true ;
ENDSUBSET

PARAMS
  ( SORTS El)
```

[**SORTS** MList **OPNS** empty-mlist: → MList, mlist : El → MList]

Ende Modul MLIST

A.6.4 Statistik zur Spezifikation

Die Gesamtspezifikation umfaßt insgesamt **75** Module.

In dieser Arbeit vorgestellt wurden davon **30**, nämlich:
AKTIVITAETENEBENE,
ATTR,
ATTRIBUTEBENE,
ATTRIBUTSORTEN,
DATENKLASSENBEZUG,
DATENMODELLEBENE,
DATENOBJEKT-COLLECTION,
DB,
DOMAINSORTEN,
DS,
ENTITIES,
ENTITY-COLLECTIONS,
ENTITY-KOMPONENTE,
FEHLERMELDUNG,
IB-SCHEMATISCH,
KEY,
LIST,
MK_7-ATTR-ENTITY,
MK_ENTITY_COLLECTION,
MK_RELATIONSHIP,
MLIST,
OK-PRAEDIKAT,
RAW_DB,
RELATIONSHIP-KOMPONENTE,
RELATIONSHIPS,
SCHNITTSTELLE,
SONDERMODUL,
SUNRISE-DATENOBJEKT,
TUPEL-2,
WSAV.

Nicht vorgestellt wurden **45** Module, nämlich:
AKTUELLES-STACK-OBJEKT-VON,
ALLES,

AUFENTHALT-COLLECTION,
AUFENTHALT,
AUFENTHALTSBEZUG,
BENUTZER,
BENUTZER-COLLECTION,
DATENKLASSE-COLLECTION,
DATENKLASSE,
DEFAULT-OBJEKT,
ENTSTANDEN-AUS,
GLOBAL-FUER,
HOSPITAL-COLLECTION,
HOSPITAL,
HOSPITALBEZUG,
LOKAL-FUER,
MAKE_BINARY_RELATION,
MK_1-ATTR-ENTITY,
MK_10-ATTR-ENTITY,
MK_4-ATTR-ENTITY,
MK_5-ATTR-ENTITY,
MK_8-ATTR-ENTITY,
MK_SPECIAL_RELATIONSHIP,
NEXTAUFENTHALTSID,
NEXTDATENID,
NEXTHOSPID,
NEXTKENNUNG,
NEXTNAME,
NEXTOPERATIONS,
NEXTPATID,
NEXTROOTID,
PATIENT-COLLECTION,
PATIENT,
PATIENTENBEZUG,
ROOT-COLLECTION,
ROOT,
ROOTBEZUG,
STACK-OBJEKT-VON,
TUPEL-1,
TUPEL-10,
TUPEL-11,
TUPEL-4,
TUPEL-5,
TUPEL-7,
TUPEL-8.

Insgesamt enthält die Spezifikation **4346** Zeilen, **12943** Worte und **157512** Character, Kommentare wurden dabei nicht berücksichtigt.

Anhang B

Auszug einer Rapid-Prototyping-Sitzung

In diesem Abschnitt wird ein Mitschnitt der Interpretersitzung vorgestellt, die mit dem Interpreter des OBSCURE-Systems (siehe [Sto91]) durchgeführt wurde. Ein Teil dieses Mitschnitts wurde in Abschnitt 3.5 bereits graphisch und informal erläutert. Die Auflistung ab Seite 167 stellt die Konstruktorterme zu den Entities vor, die in Abbildung 3.3 auftreten. Leider sind die Terme, die vom OBSCURE-Interpreter ausgegeben werden, wenig aussagekräftig. Nicht nur für einen Leser, der die Spezifikation der Datenmodellebene (Kapitel A) nicht ausführlich nachvollzogen hat, sind diese womöglich absolut unverständlich. Dies liegt vor allem daran, daß die Domainsorten (z. B. *Name* und *Datum*) nicht wirklich spezifiziert wurden, sondern aus einer Umbenennung der ganzen Zahlen hervorgehen. Dieser Aspekt wurde in Abschnitt A.2.1 näher diskutiert. Um die schlechte Lesbarkeit des Mitschnitts der Interpretersitzung etwas zu verbessern, wird in dieser Auflistung zu jeder Entity zunächst ein Bezeichner angegeben — diese Bezeichner, z. B. D10, entsprechen denjenigen aus Abbildung A.1 —, dann folgt der (unleserliche) Konstruktorterm, der diese Entity repräsentiert, und zuletzt wird eine Beschreibung angegeben, in der die abstrakten Attribute der Form a *d n* (mit *n* ist eine Ganze Zahl) ersetzt werden durch Bezeichnungen, wie sie in einer realistischen Einsatzsituation des Systems auftreten könnten. Beim späteren Studieren der Konstruktorterme zum Ausgangs- und Resultatzustand der Operationsanwendung von WSAV kann diese Auflistung nützlich sein.

Bevor diese Auflistung erfolgt, soll zum besseren Verständnis der Konstruktorterme zunächst ein Beispiel betrachtet werden: Angenommen man möchte eine Entity zum Entitytyp *Hospital* kreieren. Dazu kann man sich der Operation *Create-Hospital* bedienen. Dieser müssen vier Attributwerte übergeben werden (siehe auch Abbildung 2.15): Eine Hospital-Identifikationsnummer (der Sorte *A-HospitalId*), ein Hospitalname (der Sorte *A-Name*), eine Adresse (der Sorte *A-Adresse*) und eine Creation-Information (der Sorte *A-CreationInfo*). Alle diese Attributwerte müssen, gemäß der Integritätsbedingungen zum Entitytyp Hospital ungleich dem leeren Attributeintrag (Konstanten e-a-...) sein. Wie aber kann ein solcher Attri-

butwert generiert werden? Dazu ist es notwendig, einen Träger der Domainsorte, die der entsprechenden Attributsorte zugrundeliegt, zu beschreiben und diesen durch Anwendung der Liftingoperation ˜ zu einem Attributwert zu liften. Weil die Spezifikation der Domainsorten für diese Fallstudie aber nicht unbedingt erforderlich war, wurde aus Umfangsgründen darauf verzichtet. Damit ein Rapid-Prototyping aber dennoch möglich ist, wurden die Domainsorten durch eine Umbenennung der ganzen Zahlen spezifiziert (siehe Seite 111). Das heißt, ein Träger der Domainsorte *Name*, kann wie folgt beschrieben werden: *name 1*, wobei *1* ein Term der Sorte *integer* ist und *name* die Operation, die einen Träger der Sorte *integer* erhebt (umbenennt) zu einem Träger der Sorte *Name* (siehe Seite 115 bzw. 116). Der Konstruktorterm zum Term *name 1* lautet *d 1*. Ein weiterer, vom Namen *d 1* verschiedener Name, wird z. B. durch den Term *name 7* beschrieben. Dessen Konstruktorterm lautet *d 7*. Wie bereits erwähnt kann durch Anwendung der Liftingoperation ˜ ein Träger einer Domainsorte zu einem Träger der entsprechenden Attributsorte geliftet werden. Der Term *˜ (name 7)* repräsentiert also einen Attributwert. Vom Interpreter wird dieser ausgewertet zum Konstruktorterm a *d 7* der *A-Name*. Eine Entity des Typs *Hospital* wird dementsprechend durch einen Term *create-Hospital(˜ (hospid 1), ˜ (name 7), ˜ (adresse 1), ˜ (creationinfo 1))* beschrieben. Leider wird dieser noch recht verständliche Eingabeterm vom Interpreter zum unübersichtlich Konstruktorterm {a *d 1*/a *d 7*/a *d 1*/a *d 1*} ausgewertet. Unbedingt zu beachten ist: Die drei Konstruktorterme a *d 1* an der ersten, dritten und vierten Stelle dieses 4-Tupels haben unterschiedliche Sorten und repräsentieren auch Träger unterschiedlicher Sorten, die absolut nichts miteinander zu tun haben. Genau dies ist der Grund, warum die Interpretersitzung so schwer nachvollziehbar ist. Wären die Domainsorten ausführlich spezifiziert worden, würde dieser Effekt nicht in dieser extremen Form auftreten.

Es erfolgt nun die Auflistung der Konstruktorterme zu den Entitytypen, die durch die eingangs erwähnten, intuitiveren Informationen ergänzt werden.

Entities des Typs Root

R0: {a d 0} *ersetze durch*
{r0}

Entities des Typs Hospital

H1: {a d 1/a d 7/a d 1/a d 1} *ersetze durch*
{h1/St.Christoph/Klarenthal/8.4.90,roland}

Entities des Typs Patient

P3 {a d 2 /a d 21/a d 22/empty/a d 2/a d 2/a d 2/a d 2} *ersetze durch*
{p2/Hufschmid/Holger/-/24.12.1962/Völkingen/männlich /1.2.91,-roland}

P4 {a d 3/a d 31/a d 32/empty/a d 3/a d 3/a d 3/a d 3} *ersetze durch*
{p3/Zeyer/Jörg/-/1.4.1950/Dudweiler/männlich/1.3.94, roland}

P5 {a d 4/a d 41/a d 42/empty/a d 4/a d 4/a d 4/a d 4} *ersetze durch*
{p4/Heckler/Ramses/-/28.6.1964/Klarenthal/männlich/2.3.94,roland}

Entities des Typs Aufenthalt

A6 {a d 2/a d 5/a d 5/a d 5/a d 5/a d 5/a d 5/a d 5/a d 5/a d 5}
ersetze durch
{p2/a5/ledig/deutsch/Student/ev/Faulitis/Ubbenstr.4,Völklingen/
Station 4,Zi.-Nr.402/1.2.91,roland}

A7 {a d 2/a d 6/a d 6/a d 6/a d 6/a d 6/a d 6/a d 6/a d 6/a d 6}
ersetze durch
{p2/a6/verheiratet/deutsch/Mathematiker/rk/Abaditis/Karlstr.15,
Völklingen/Station 3,Zi.-Nr.331/2.3.94,roland}

A8 {a d 3/a d 7/a d 7/a d 7/a d 7/a d 7/a d 7/a d 7/a d 7/a d 7}
ersetze durch
{p3/a7/ledig/saarl./Rhytmiker/rk/Depressionen/Zappastr.45,Dud-
weiler/Station 2,Zi.-Nr.204/3.3.94,roland}

A9 {a d 4/a d 8/a d 8/a d 8/a d 8/a d 8/a d 8/a d 8/a d 8/a d 8}
ersetze durch
{p4/a8/ledig/saarl./Philosoph/rk/Spezifikatistis/ Pyramidenweg
2,Klarenthal/Station 2,Zi.-Nr.204/4.3.94,roland}

Entities des Typs Sunrise-Datenobjekt

D10 {a d 9/a d 9/a d 9/a d 9/a d 9/empty/a d 9} *ersetze durch*
{d9/Hufschmid Schädel 1/p2,p3/ Import-Röntgenbild von R-
Station1/bindaten-9/-/10.2.91,roland}

D11 {a d 10/a d 10/a d 10/a d 10/a d 10/empty/a d 10} *ersetze durch*
{d10/Hufschmid Schädel 2/p4,p5/Import-Röntgenbild von R-Station2/bindaten-10/-/5.3.94,roland}

D12 {a d 11/empty/a d 11/a d 11/empty/empty/a d 11} *ersetze durch*
{d11/-/p5,p8/Graustufenfilter (auf d02)/-/-/6.3.94, roland}

D13 {a d 12/empty/a d 12/a d 12/empty/empty/a d 12} *ersetze durch*
{d12/-/p5,p8,p4,p2/Medianfilter,Graustufenfilter (auf d02) /-/-/6.3.94,roland}

D14 {a d 13/a d 13/a d 13/a d 13/a d 13/a d 13/a d 13} *ersetze durch*
{d13/Zeyer Blut-Anlalyse 1/p1/Import-Spektrum von S-Station1/ bindaten-13/1403,1003/6.3.94,chris}

D15 {a d 14/a d 14/a d 14/a d 14/a d 14/empty/a d 14} *ersetze durch*
{d14/Heckler Blut-Analyse 1/p3/Import-Spektrum von S-Station2/ bindaten-14/-/7.3.94,chris}

D16 {a d 15 /a d 15/a d 15/a d 15/empty/empty/a d 15} *ersetze durch*
{d15/Heckler Blut-Analyse 2/p3,p13/Fouriertransformation (auf d06)/-/-/7.3.94,chris}

D17 {a d 16/a d 16/a d 16/a d 16/a d 16/empty/a d 16} *ersetze durch*
{d16/Heckler Blut-Analyse 3/p1,p2/Import-Spektrum von S-Station2/bindaten-16/-/7.3.94,chris}

D18 {a d 17/empty/a d 17/a d 17/empty/empty/a d 17} *ersetze durch*
{d17/-/p14,p1/Fouriertransformation (d08)/-/-/7.3.94,chris}

Entities des Typs Datenklasse

P19 {a d 18} *ersetze durch*
{Image}

P20 {a d 19} *ersetze durch*
{Spektrum}

Entities des Typs Benutzer

P1 {a d 201/a d 202/a d 20/a d 20/a d 20} *ersetze durch*
{Benzmüller/Christoph/Waldstr.4,Saarbrücken/8.9.1968/chris}

P2 {a d 211/a d 212/a d 21/a d 21/a d 21} *ersetze durch*
{Brill/Roland/Freiburgerstr.12,Saarbrücken/refer2.1965/roland}

Zur Interpretersitzung wird vorausgesetzt, daß die Datei rapid-prototyping.B, in der der Ausgangszustand zur Operationsanwendung (siehe auch Abbildung 3.3) durch einen Term beschrieben wird, folgenden Inhalt hat (vor den Doppelpunkten wird jeweils eine Interpreter-Variable angegeben, dahinter folgt der zugeordnete Term):

```
R0 : create-Root(~(rootid 0)) ;

H1 : create-Hospital(~(hospid 1), ~(name 7), ~(adresse 1),
~(creationinfo 1)) ;

P2 : create-Patient(~(patid 2), ~(name (10 + 10 + 1)),
~(name (10 + 10 + 2)), e-a-name, ~(datum 2), ~(ort 2),
~(geschl 2), ~(creationinfo 2)) ;
P3 : create-Patient(~(patid 3), ~(name (10 + 10 + 10 + 1)),
~(name (10 + 10 + 10 + 2)), e-a-name, ~(datum 3), ~(ort 3),
~(geschl 3), ~(creationinfo 3)) ;
P4 : create-Patient(~(patid 4), ~(name (10 + 10 + 10 + 10 + 1)),
~(name (10 + 10 + 10 + 10 + 2)), e-a-name, ~(datum 4), ~(ort 4),
~(geschl 4), ~(creationinfo 4)) ;

A5 : create-Aufenthalt(~(patid 2), ~(aufid 5), ~(stand 5), ~(nation 5),
~(beruf 5), ~(religion 5), ~(diagn 5), ~(adresse 5), ~(aufinfo 5),
~(creationinfo 5)) ;
A6 : create-Aufenthalt(~(patid 2), ~(aufid 6), ~(stand 6), ~(nation 6),
~(beruf 6), ~(religion 6), ~(diagn 6), ~(adresse 6), ~(aufinfo 6),
~(creationinfo 6)) ;
A7 : create-Aufenthalt(~(patid 3), ~(aufid 7), ~(stand 7), ~(nation 7),
~(beruf 7), ~(religion 7), ~(diagn 7), ~(adresse 7), ~(aufinfo 7),
~(creationinfo 7)) ;
A8 : create-Aufenthalt(~(patid 4), ~(aufid 8), ~(stand 8), ~(nation 8),
~(beruf 8), ~(religion 8), ~(diagn 8), ~(adresse 8), ~(aufinfo 8),
~(creationinfo 8)) ;

D9 : create-Sunrise-Datenobjekt(~(datenid 9), ~(name 9), ~(paraliste 9),
~(history 9), ~(bindaten 9), e-a-roi, ~(creationinfo 9)) ;
```

```
D10 : create-Sunrise-Datenobjekt(~(datenid 10), ~(name 10),
~(paraliste 10), ~(history 10), ~(bindaten 10), e-a-roi,
~(creationinfo 10)) ;
D11 : create-Sunrise-Datenobjekt(~(datenid (10 + 1)), e-a-name,
~(paraliste (10 + 1)), ~(history (10 + 1)), e-a-bindaten, e-a-roi,
~(creationinfo (10 + 1))) ;
D12 : create-Sunrise-Datenobjekt(~(datenid (10 + 2)), e-a-name,
~(paraliste (10 + 2)), ~(history (10 + 2)), e-a-bindaten, e-a-roi,
~(creationinfo (10 + 2))) ;
D13 : create-Sunrise-Datenobjekt(~(datenid (10 + 3)), ~(name (10 + 3)),
~(paraliste (10 + 3)), ~(history (10 + 3)), ~(bindaten (10 + 3)),
~(roi (10 + 3)), ~(creationinfo (10 + 3))) ;
D14 : create-Sunrise-Datenobjekt(~(datenid (10 + 4)), ~(name (10 + 4)),
~(paraliste (10 + 4)), ~(history (10 + 4)), ~ (bindaten (10 + 4)), e-a-roi,
~(creationinfo (10 + 4))) ;
D15 : create-Sunrise-Datenobjekt(~(datenid (10 + 5)), ~(name (10 + 5)),
~(paraliste (10 + 5)), ~(history (10 + 5)), e-a-bindaten, e-a-roi,
~(creationinfo (10 + 5))) ;
D16 : create-Sunrise-Datenobjekt(~(datenid (10 + 6)), ~(name (10 + 6)),
~(paraliste (10 + 6)), ~(history (10 + 6)), ~(bindaten (10 + 6)), e-a-roi,
~(creationinfo (10 + 6))) ;
D17 : create-Sunrise-Datenobjekt(~(datenid (10 + 7)), e-a-name,
~(paraliste (10 + 7)), ~(history (10 + 7)), e-a-bindaten, e-a-roi,
~(creationinfo (10 + 7))) ;

K18 : create-Datenklasse(~(name (10 + 8))) ;
K19 : create-Datenklasse(~(name (10 + 9))) ;

B20 : create-Benutzer(~(name ((2 * 10 * 10) + 1)),
~(name ((2 * 10 * 10) + 2)), ~(adresse (10 + 10)), ~(datum (10 + 10)),
~(kennung (10 + 10))) ;
B21 : create-Benutzer(~(name ((2 * 10 * 10) + 10 + 1)),
~(name ((2 * 10 * 10) + 10 + 2)), ~(adresse (10 + 10 + 1)),
~(datum (10 + 10 + 1)), ~(kennung (10 + 10 + 1))) ;

rc : ins(R0,get-Root-coll(get-Entity-Db(e-Db))) ;

hc : ins(H1,get-Hospital-coll(get-Entity-Db(e-Db))) ;

pc : ins(P2,get-Patient-coll(get-Entity-Db(e-Db))) ;
pc : ins(P3,pc) ;
pc : ins(P4,pc) ;

ac : ins(A5,get-Aufenthalt-coll(get-Entity-Db(e-Db))) ;
ac : ins(A6,ac) ;
ac : ins(A7,ac) ;
ac : ins(A8,ac) ;

dc : ins(D9,get-Datenobjekt-coll(get-Entity-Db(e-Db))) ;
dc : ins(D10,dc) ;
```

```
dc : ins(D11,dc) ;
dc : ins(D12,dc) ;
dc : ins(D13,dc) ;
dc : ins(D14,dc) ;
dc : ins(D15,dc) ;
dc : ins(D16,dc) ;
dc : ins(D17,dc) ;

kc : ins(K18,get-Datenklasse-coll(get-Entity-Db(e-Db))) ;
kc : ins(K19,kc) ;

bc : ins(B20,get-Benutzer-coll(get-Entity-Db(e-Db))) ;
bc : ins(B21,bc) ;

EDB : {bc/kc/dc/ac/pc/hc/rc} ;



rb : est-Rootbezug(get-Rootbezug(get-Relationship-Db(e-Db)), hospid 1,
rootid 0) ;

hb : est-Hospitalbezug(get-Hospitalbezug(get-Relationship-Db(e-Db)),
patid 2, hospid 1) ;
hb : est-Hospitalbezug(hb, patid 3, hospid 1) ;
hb : est-Hospitalbezug(hb, patid 4, hospid 1) ;

pb : est-Patientenbezug(get-Patientenbezug(get-Relationship-Db(e-Db)),
{(patid 2)/(aufid 5)}, patid 2) ;
pb : est-Patientenbezug(pb, {(patid 2)/(aufid 6)}, patid 2) ;
pb : est-Patientenbezug(pb, {(patid 3)/(aufid 7)}, patid 3) ;
pb : est-Patientenbezug(pb, {(patid 4)/(aufid 8)}, patid 4) ;

ab : est-Aufenthaltsbezug(get-Aufenthaltsbezug(get-Relationship-Db(e-Db)),
datenid 9, {(patid 2)/(aufid 5)}) ;
ab : est-Aufenthaltsbezug(ab, datenid 10, {(patid 2)/(aufid 6)}) ;
ab : est-Aufenthaltsbezug(ab, datenid (10 + 1), {(patid 2)/(aufid 6)}) ;
ab : est-Aufenthaltsbezug(ab, datenid (10 + 2), {(patid 2)/(aufid 6)}) ;
ab : est-Aufenthaltsbezug(ab, datenid (10 + 3), {(patid 3)/(aufid 7)}) ;
ab : est-Aufenthaltsbezug(ab, datenid (10 + 4), {(patid 4)/(aufid 8)}) ;
ab : est-Aufenthaltsbezug(ab, datenid (10 + 5), {(patid 4)/(aufid 8)}) ;
ab : est-Aufenthaltsbezug(ab, datenid (10 + 6), {(patid 4)/(aufid 8)}) ;
ab : est-Aufenthaltsbezug(ab, datenid (10 + 7), {(patid 4)/(aufid 8)}) ;

kb : est-Datenklassenbezug(get-Datenklassenbezug(get-Relationship-Db(e-Db)),
datenid 9, name (10 + 8)) ;
kb : est-Datenklassenbezug(kb, datenid 10, name (10 + 8)) ;
kb : est-Datenklassenbezug(kb, datenid (10 + 1), name (10 + 8)) ;
kb : est-Datenklassenbezug(kb, datenid (10 + 2), name (10 + 8)) ;
kb : est-Datenklassenbezug(kb, datenid (10 + 3), name (10 + 9)) ;
kb : est-Datenklassenbezug(kb, datenid (10 + 4), name (10 + 9)) ;
```

```
kb : est-Datenklassenbezug(kb, datenid (10 + 5), name (10 + 9)) ;
kb : est-Datenklassenbezug(kb, datenid (10 + 6), name (10 + 9)) ;
kb : est-Datenklassenbezug(kb, datenid (10 + 7), name (10 + 9)) ;

eab : est-Entstanden-aus(get-Entstanden-aus(e-Db), datenid (10 + 1),
datenid 10) ;
eab : est-Entstanden-aus(eab, datenid (10 + 2), datenid (10 + 1)) ;
eab : est-Entstanden-aus(eab, datenid (10 + 5), datenid (10 + 4)) ;
eab : est-Entstanden-aus(eab, datenid (10 + 7), datenid (10 + 6)) ;

gfb : est-Global-fuer(get-Global-fuer(get-Relationship-Db(e-Db)), datenid 9,
kennung (10 + 10)) ;
gfb : est-Global-fuer(gfb, datenid 10, kennung (10 + 10)) ;
gfb : est-Global-fuer(gfb, datenid (10 + 3), kennung (10 + 10 + 1)) ;
gfb : est-Global-fuer(gfb, datenid (10 + 4), kennung (10 + 10 + 1)) ;
gfb : est-Global-fuer(gfb, datenid (10 + 5), kennung (10 + 10 + 1)) ;
gfb : est-Global-fuer(gfb, datenid (10 + 6), kennung (10 + 10 + 1)) ;

lfb : est-Lokal-fuer(get-Lokal-fuer(get-Relationship-Db(e-Db)),
datenid (10 + 1), kennung (10 + 10)) ;
lfb : est-Lokal-fuer(lfb, datenid (10 + 2), kennung (10 + 10)) ;
lfb : est-Lokal-fuer(lfb, datenid (10 + 7), kennung (10 + 10 + 1)) ;

sob : est-Stack-Objekt-von(get-Stack-Objekt-von(get-Relationship-Db(e-Db)),
datenid 10, kennung (10 + 10)) ;
sob : est-Stack-Objekt-von(sob, datenid (10 + 5), kennung (10 + 10 + 1)) ;
sob : est-Stack-Objekt-von(sob, datenid (10 + 6), kennung (10 + 10 + 1)) ;

asob : est-Aktuelles-Stack-Objekt-von(get-Aktuelles-Stack-Objekt-von(e-Db),
datenid 10, kennung (10 + 10)) ;
asob : est-Aktuelles-Stack-Objekt-von(asob, datenid (10 + 5),
kennung (10 + 10 + 1)) ;

dob : est-Default-Objekt(get-Default-Objekt(e-Db), datenid (10 + 2),
datenid 10) ;
dob : est-Default-Objekt(dob, datenid (10 + 5), datenid (10 + 5)) ;
dob : est-Default-Objekt(dob, datenid (10 + 7), datenid (10 + 6)) ;

RDB : {rb/hb/pb/ab/lfb/gfb/sob/asob/kb/dob/eab} ;


DB : {EDB/RDB} ;
```

Nun wird der Mitschnitt der Interpretersitzung vorgestellt. Teilweise werden am rechten Rand durch einige Stichworte erläuternde Hinweise gegeben (siehe z. B. Seite 177). Konkret werden die Bezeichner der Entities, deren Typ und die Relationshiptypen aufgelistet.

Erläuternde Kommentare im laufenden Mitschnitt sind in der Schriftart *Italic* abgedruckt. <>: ist das Eingabeprompt des OBSCURE-Interpreters.

<>: c ALLES

Die Spezifikation wurde in den Interpreter eingelesen.

<>: R rapid-prototyping.B

Der Zustand der Ausgangsdatenbank wurde in der Datei rapid-prototyping.B durch den Term zur Interpreter-Variablen 'DB' beschrieben (Beachte: Dieser Term enthält selbst wieder Interpreter-Variablen, die einen Eingabeterm repräsentieren usw.). Dieser Ausgangszustand wurde durch den Interpreter-Befehl 'R rapid-prototyping.B' nun in den Interpreter eingelesen.

<>: P OK(DB)

Zunächst soll überprüft werden, ob die Ausgangsdatenbank auch den Integritätsbedingungen genügt. Dazu wird auf die Ausgangsdatenbank — in rapid-prototyping.B wird dieser der Name 'DB' zugewiesen — das OK-Prädikat angewendet. Der Befehl 'P OK(DB)' ('P' steht für 'parse') fordert den Interpreter zum parsen des Terms 'OK(DB)' auf.

<>: f
true

Der Befehl 'f' (für 'force') hat das Auswerten des aktuell eingelesenen Terms zur Folge. Die Ausgangsdatenbank erfüllt also die Integritätsbedingungen, sie ist eine integre Datenbank.

<>: P kennung(10+10)

Die Operation WSAV *soll auf diese Ausgangsdatenbank anwendet werden. Dies fordert neben 'DB' als weitere Argumente die Kennung eines Benutzers und einen (lokal) noch nicht vergebenen Namen. Als Benutzer wird B20 gewählt, dessen Kennung wird durch den Term 'kennung(20)' beschrieben. Leider kann die Integer-Zahl '20' nur in der Form '10+10' eingegeben werden.*

<>: A chris

Durch den Befehl 'A chris' ('A' steht für 'assign') wird der aktuelle Term 'kennung(10+10)' der Interpreter-Variablen 'chris' zugewiesen.

<>: f
d 20

Ausgewertet wird 'kennung(10+10)' zu 'd 20', wobei 'd' der Umbenennungskonstruktor ist, der die ganzen Zahlen zu Trägern der Sorte 'Kennung' erhebt.

<>: P kennung(10+10+1)

Ein weiterer Term der Sorte 'Kennung' wird geparst,

<>: A roland

der Interpreter-Variable 'roland' zugewiesen

<>: f
d 21

und ausgewertet zu 'd 21'.

<>: P name(10*10)

Nun wird ein Term der Sorte 'Name' geparst,

<>: A Bild-neu

der Interpreter-Variablen 'Bild-neu' zugewiesen

<>: f
d 100

und ausgewertet zu 'd 100'. Damit sind die Vorarbeiten zur Anwendung der Operation WSAV *beendet.*

<>: D v DB

Bevor WSAV *angewendet wird, soll der Konstruktorterm zur Variablen 'DB' (d. h. zum Term, der der Interpretervariablen 'DB' zugewiesen wurde), der die Ausgangsdatenbank repräsentiert, vorgestellt werden. Dieser kann dann mit dem resultierenden Konstruktorterm der Operationsanwendung verglichen werden. Mit dem Befehl 'D v DB' ('D' steht für 'Display') wird der Interpreter aufgefortert, den Konstruktorterm zur Variablen 'DB' anzuzeigen. Wie bereits erwähnt, finden sich am rechten Seitenrand hilfreiche Erläuterungen zu einzelnen Subtermen. Eine intuitivere Beschreibung der Konstruktorterme zu den einzelnen Entitytypen wurde durch die Auflistung ab Seite 167 bereits vorgestellt.*

DB:
term: { EDB / RDB }
sort: Db
value:

	Erläuterungen
{	
{	
{ a d 211 / a d 212 / a d 21 / a d 21 / a d 21 } ^	B21
{ a d 201 / a d 202 / a d 20 / a d 20 / a d 20 } ^	B20
empty-mlist	Benutzer
/	
{ a d 19 } ^	K19
{ a d 18 } ^	K18
empty-mlist	Datenklasse
/	
{ a d 17 / empty / a d 17 / a d 17 / empty / empty / a d 17 } ^	D17
{ a d 16 / a d 16 / a d 16 / a d 16 / a d 16 / empty / a d 16 } ^	D16
{ a d 15 / a d 15 / a d 15 / a d 15 / empty / empty / a d 15 } ^	D15
{ a d 14 / a d 14 / a d 14 / a d 14 / a d 14 / empty / a d 14 } ^	D14
{ a d 13 / a d 13 / a d 13 / a d 13 / a d 13 / a d 13 / a d 13 } ^	D13
{ a d 12 / empty / a d 12 / a d 12 / empty / empty / a d 12 } ^	D12
{ a d 11 / empty / a d 11 / a d 11 / empty / empty / a d 11 } ^	D11
{ a d 10 / a d 10 / a d 10 / a d 10 / a d 10 / empty / a d 10 } ^	D10

```
    { a d 9 / a d 9 / a d 9 / a d 9 / a d 9 / empty / a d 9 } ^                    D9
    empty-mlist                                                                   S.-Datenobjekt
  /
    { a d 4 / a d 8 / a d 8 / a d 8 / a d 8 / a d 8 / a d 8 / a d 8 / a d 8 } ^   A8
    { a d 3 / a d 7 / a d 7 / a d 7 / a d 7 / a d 7 / a d 7 / a d 7 / a d 7 } ^   A7
    { a d 2 / a d 6 / a d 6 / a d 6 / a d 6 / a d 6 / a d 6 / a d 6 / a d 6 } ^   A6
    { a d 2 / a d 5 / a d 5 / a d 5 / a d 5 / a d 5 / a d 5 / a d 5 / a d 5 } ^   A5
    empty-mlist
  /
    { a d 4 / a d 41 / a d 42 / empty / a d 4 / a d 4 / a d 4 / a d 4 } ^          P4
    { a d 3 / a d 31 / a d 32 / empty / a d 3 / a d 3 / a d 3 / a d 3 } ^         P3
    { a d 2 / a d 21 / a d 22 / empty / a d 2 / a d 2 / a d 2 / a d 2 } ^         P2
    empty-mlist                                                                   Aufenthalt
  /
    { a d 1 / a d 7 / a d 1 / a d 1 } ^                                           H1
    empty-mlist                                                                   Hospital
  /
    { a d 0 } ^                                                                   R0
    empty-mlist                                                                   Root
  }
/
  {
    { d 1 / d 0 } ^ empty-mlist                                                   Rootbezug
  /
    { d 4 / d 1 } ^ { d 3 / d 1 } ^ { d 2 / d 1 } ^ empty-mlist                   Hospitalbezug
  /
    { { d 4 / d 8 } / d 4 } ^ { { d 3 / d 7 } / d 3 } ^

    { { d 2 / d 6 } / d 2 } ^ { { d 2 / d 5 } / d 2 } ^ empty-mlist               Patientenbezug
  /
    { d 17 / { d 4 / d 8 } } ^ { d 16 / { d 4 / d 8 } } ^

    { d 15 / { d 4 / d 8 } } ^ { d 14 / { d 4 / d 8 } } ^

    { d 13 / { d 3 / d 7 } } ^ { d 12 / { d 2 / d 6 } } ^

    { d 11 / { d 2 / d 6 } } ^ { d 10 / { d 2 / d 6 } } ^

    { d 9 / { d 2 / d 5 } } ^ empty-mlist                                         Aufenthaltsbezug

  /
    { d 17 / d 21 } ^ { d 12 / d 20 } ^ { d 11 / d 20 } ^ empty-mlist             Lokal-fuer
  /
    { d 16 / d 21 } ^ { d 15 / d 21 } ^ { d 14 / d 21 } ^
    { d 13 / d 21 } ^ { d 10 / d 20 } ^ { d 9 / d 20 } ^ empty-mlist              Global-fuer
  /
    { d 16 / d 21 } ^ { d 15 / d 21 } ^ { d 10 / d 20 } ^ empty-mlist             Stack-Objekt-von
  /
    { d 15 / d 21 } ^ { d 10 / d 20 } ^ empty-mlist                               Aktuelles-Stack-
                                                                                  Objekt-von
```

```
  /
    { d 17 / d 19 } ^ { d 16 / d 19 } ^ { d 15 / d 19 } ^

    { d 14 / d 19 } ^ { d 13 / d 19 } ^ { d 12 / d 18 } ^

    { d 11 / d 18 } ^ { d 10 / d 18 } ^ { d 9 / d 18 } ^ empty-mlist          Datenklassenbezug
  /
    { d 17 / d 16 } ^ { d 15 / d 15 } ^ { d 12 / d 10 } ^ empty-mlist         Default-Objekt
  /
    { d 17 / d 16 } ^ { d 15 / d 14 } ^ { d 12 / d 11 } ^

    { d 11 / d 10 } ^ empty-mlist                                            Entstanden-aus
  }
}
<>: P wsav(DB,chris,Bild-neu)
```

Nun wird WSAV *auf die Argumente 'DB', 'chris' (die Kennung von Benutzer B20) und 'Bild-neu' (ein neuer Datenobjektname) angewendet. Diese Anwendung entspricht dem in Abschnitt 3.5 durch die Abbildungen 3.3 und 3.4 diskutierten Beispiel einer Operationsanwendung. Nach dem Parsen wird der Term ausgewertet.*

```
<>: f
{
{
  {
    { a d 211 / a d 212 / a d 21 / a d 21 / a d 21 } ^                       B21
    { a d 201 / a d 202 / a d 20 / a d 20 / a d 20 } ^                       B20
    empty-mlist                                                              Benutzer
  /
    { a d 19 } ^                                                             K19
    { a d 18 } ^                                                             K18
    empty-mlist                                                              Datenklasse
  /
    { a d 17 / empty / a d 17 / a d 17 / empty / empty / a d 17 } ^           D17
    { a d 16 / a d 16 / a d 16 / a d 16 / a d 16 / empty / a d 16 } ^         D16
    { a d 15 / a d 15 / a d 15 / a d 15 / empty / empty / a d 15 } ^          D15
    { a d 14 / a d 14 / a d 14 / a d 14 / a d 14 / empty / a d 14 } ^         D14
    { a d 13 / a d 13 / a d 13 / a d 13 / a d 13 / a d 13 / a d 13 } ^       D13
    { a d 12 / a d 100 / a d 12 / a d 12 / empty / empty / a d 12 } ^        D12
    { a d 10 / a d 10 / a d 10 / a d 10 / a d 10 / empty / a d 10 } ^        D10
    { a d 9 / a d 9 / a d 9 / a d 9 / a d 9 / empty / a d 9 } ^              D9
    empty-mlist                                                              S.-Datenobjekt
  /
    { a d 4 / a d 8 / a d 8 / a d 8 / a d 8 / a d 8 / a d 8 / a d 8 / a d 8 } ^   A8
    { a d 3 / a d 7 / a d 7 / a d 7 / a d 7 / a d 7 / a d 7 / a d 7 / a d 7 } ^   A7
    { a d 2 / a d 6 / a d 6 / a d 6 / a d 6 / a d 6 / a d 6 / a d 6 / a d 6 } ^   A6
    { a d 2 / a d 5 / a d 5 / a d 5 / a d 5 / a d 5 / a d 5 / a d 5 / a d 5 } ^   A5
    empty-mlist                                                              Aufenthalt
```

```
  /
    { a d 4 / a d 41 / a d 42 / empty / a d 4 / a d 4 / a d 4 / a d 4 } ^        P4
    { a d 3 / a d 31 / a d 32 / empty / a d 3 / a d 3 / a d 3 / a d 3 } ^       P3
    { a d 2 / a d 21 / a d 22 / empty / a d 2 / a d 2 / a d 2 / a d 2 } ^       P2
    empty-mlist                                                                 Patient
  /
    { a d 1 / a d 7 / a d 1 / a d 1 } ^                                         H1
    empty-mlist                                                                 Hospital
  /
    { a d 0 } ^                                                                 R0
    empty-mlist                                                                 Root
  }
/
  {
    { d 1 / d 0 } ^ empty-mlist                                                 Rootbezug
  /
    { d 4 / d 1 } ^ { d 3 / d 1 } ^ { d 2 / d 1 } ^ empty-mlist                 Hospitalbezug
  /
    { { d 4 / d 8 } / d 4 } ^ { { d 3 / d 7 } / d 3 } ^ { { d 2 / d 6 } / d 2 } ^

    { { d 2 / d 5 } / d 2 } ^ empty-mlist                                       Patientenbezug
  /
    { d 17 / { d 4 / d 8 } } ^ { d 16 / { d 4 / d 8 } } ^ { d 15 / { d 4 / d 8 } } ^

    { d 14 / { d 4 / d 8 } } ^ { d 13 / { d 3 / d 7 } } ^ { d 12 / { d 2 / d 6 } } ^

    { d 10 / { d 2 / d 6 } } ^ { d 9 / { d 2 / d 5 } } ^ empty-mlist            Aufenthaltsbezug
  /
    { d 17 / d 21 } ^ empty-mlist                                               Lokal-fuer
  /
    { d 12 / d 20 } ^ { d 16 /d 21 } ^ { d 15 / d 21 } ^ { d 14 / d 21 } ^

    { d 13 / d 21 } ^ { d 10 / d 20 } ^ { d 9 / d 20 } ^ empty-mlist            Global-fuer
  /
    { d 12 / d 20 } ^ { d 16 / d 21 } ^ { d 15 / d 21 } ^ empty-mlist           Stack-Objekt-von
  /
    { d 12 / d 20 } ^ { d 15 / d 21 } ^ empty-mlist                             Aktuelles-Stack-
                                                                                Objekt-von
  /
    { d 17 / d 19 } ^ { d 16 / d 19 } ^ { d 15 / d 19 } ^

    { d 14 / d 19 } ^ { d 13 / d 19 } ^ { d 12 / d 18 } ^

    { d 10 / d 18 } ^ { d 9 / d 18 } ^ empty-mlist                              Datenklassenbezug
  /
    { d 12 / d 12 } ^ { d 17 / d 16 } ^ { d 15 / d 15 } ^ empty-mlist           Default-Objekt
  /
    { d 12 / d 10 } ^ { d 17 /d 16 } ^ { d 15 / d 14 } ^ empty-mlist            Entstanden-aus
  }
}
```

/ e }

<>: A DB2xF

Dem soeben ausgewerteten Term wird der Name 'DB2xF' zugewiesen. Man beachte, daß gemäß der Zielsorte der Operation Wsav *(siehe Spezifikation der Operation* Wsav *auf Seite 31) dieser Term ein Paar ist, bestehend aus einer Datenbank und einer Fehlermeldung — in diesem Fall die leere Fehlermeldung, repräsentiert durch 'e'.*

<>: P fst(DB2xF)

'fst' selektiert die erste Komponente des Resultatterms.

<>: A DB2

Dieser wird der Interpretervariablen 'DB2' zugewiesen.

<>: P OK(DB2)

Es kann nun überprüft werden, ob dieser Resultatterm die Integritätsbedingungen, ebenfalls erfüllt, also ob die Anwendung von Wsav *auf die Ausgangsdatenbank zu einer integren Datenbank führte.*

<>: f
true

Die aus der Operationsanwendung resultierende Datenbank erfüllt die Integritätsbedingungen.

Ein Auszug aus der weiteren Interpretersitzung wird nun unkommenentiert aufgelistet.

<>: P wsav(DB2,chris,Bild-neu)
<>: f
{ { { { a d 211 / a d 212 / a d 21 / a d 21 / a d 21 } ^ { a d 201 / a d 202 / a d 20 / a d 20 / a d 20 } ^ empty-mlist / { a d 19 } ^ { a d 18 } ^ empty-mlist / { a d 17 / empty / a d 17 / a d 17 / empty / empty / a d 17 } ^ { a d 16 / a d 16 / a d 16 / a d 16 / a d 16 / empty / a d 16 } ^ { a d 15 / a d 15 / a d 15 / a d 15 / empty / empty / a d 15 } ^ { a d 14 / a d 14 / a d 14 / a d 14 / a d 14 / empty / a d 14 } ^ { a d 13 / a d 13 / a d 13 / a d 13 / a d 13 / a d 13 / a d 13 } ^ { a d 12 / a d 100 / a d 12 / a d 12 / empty / empty / a d 12 } ^ { a d 10 / a d 10 / a d 10 / a d 10 / a d 10 / empty / a d 10 } ^ { a d 9 / a d 9 / a d 9 / a d 9 / a d 9 / empty / a d 9 } ^ empty-mlist / { a d 4 / a d 8 / a d 8 / a d 8 / a d 8 / a d 8 / a d 8 / a d 8 / a d 8 / a d 8 } ^ { a d 3 / a d 7 / a d 7 / a d 7 / a d 7 / a d 7 / a d 7 / a d 7 / a d 7 / a d 7 } ^ { a d 2 / a d 6 / a d 6 / a d 6 / a d 6 / a d 6 / a d 6 / a d 6 / a d 6 / a d 6 } ^ { a d 2 / a d 5 / a d 5 / a d 5 / a d 5 / a d 5 / a d 5 / a d 5 / a d 5 / a d 5 } ^ empty-mlist / { a d 4 / a d 41 / a d 42 / empty / a d 4 / a d 4 / a d 4 / a d 4 } ^ { a d 3 / a d 31 / a d 32 / empty / a d 3 / a d 3 / a d 3 / a d 3 } ^ { a d 2 / a d 21 / a d 22 / empty / a d 2 / a d 2 / a d 2 / a d 2 } ^ empty-mlist / { a d 1 / a d 7 / a d 1 / a d 1 } ^ empty-mlist / { a d 0 } ^ empty-mlist } / { { d 1 / d 0 } ^ empty-mlist / { d 4 / d 1 } ^ { d 3 / d 1 } ^ { d 2 / d 1 } ^ empty-mlist / { { d 4 / d 8 } / d 4 } ^ { { d 3 / d 7 } / d 3 } ^ { { d 2 / d 6 } / d 2 } ^ { { d 2 / d 5 } / d 2 } ^ empty-mlist / { d 17 / { d 4 / d 8 } } ^ { d 16 / { d 4 / d 8 } } ^ { d 15 / { d 4 / d 8 } } ^ { d 14 / { d 4 / d 8 } } ^ { d 13 / {

d 3 / d 7 } } ^ { d 12 / { d 2 / d 6 } } ^ { d 10 / { d 2 / d 6 } } ^ { d 9 / { d 2 / d 5 } } ^ empty-mlist / { d 17 / d 21 } ^ empty-mlist / { d 12 / d 20 } ^ { d 16 / d 21 } ^ { d 15 / d 21 } ^ { d 14 / d 21 } ^ { d 13 / d 21 } ^ { d 10 / d 20 } ^ { d 9 / d 20 } ^ empty-mlist / { d 12 / d 20 } ^ { d 16 / d 21 } ^ { d 15 / d 21 } ^ empty-mlist / { d 12 / d 20 } ^ { d 15 / d 21 } ^ empty-mlist / { d 17 / d 19 } ^ { d 16 / d 19 } ^ { d 15 / d 19 } ^ { d 14 / d 19 } ^ { d 13 / d 19 } ^ { d 12 / d 18 } ^ { d 10 / d 18 } ^ { d 9 / d 18 } ^ empty-mlist / { d 12 / d 12 } ^ { d 17 / d 16 } ^ { d 15 / d 15 } ^ empty-mlist / { d 12 / d 10 } ^ { d 17 / d 16 } ^ { d 15 / d 14 } ^ empty-mlist } } / nichts-abzuspeichern ^ e }
<>: P wsav(DB,roland,Bild-neu)
<>: f
{ { { { a d 211 / a d 212 / a d 21 / a d 21 / a d 21 } ^ { a d 201 / a d 202 / a d 20 / a d 20 / a d 20 } ^ empty-mlist / { a d 19 } ^ { a d 18 } ^ empty-mlist / { a d 17 / empty / a d 17 / a d 17 / empty / empty / a d 17 } ^ { a d 16 / a d 16 / a d 16 / a d 16 / a d 16 / empty / a d 16 } ^ { a d 15 / a d 15 / a d 15 / a d 15 / empty / empty / a d 15 } ^ { a d 14 / a d 14 / a d 14 / a d 14 / a d 14 / empty / a d 14 } ^ { a d 13 / a d 13 / a d 13 / a d 13 / a d 13 / a d 13 / a d 13 } ^ { a d 12 / empty / a d 12 / a d 12 / empty / empty / a d 12 } ^ { a d 11 / empty / a d 11 / a d 11 / empty / empty / a d 11 } ^ { a d 10 / a d 10 / a d 10 / a d 10 / a d 10 / empty / a d 10 } ^ { a d 9 / a d 9 / a d 9 / a d 9 / a d 9 / empty / a d 9 } ^ empty-mlist / { a d 4 / a d 8 / a d 8 / a d 8 / a d 8 / a d 8 / a d 8 / a d 8 / a d 8 / a d 8 } ^ { a d 3 / a d 7 / a d 7 / a d 7 / a d 7 / a d 7 / a d 7 / a d 7 / a d 7 / a d 7 } ^ { a d 2 / a d 6 / a d 6 / a d 6 / a d 6 / a d 6 / a d 6 / a d 6 / a d 6 / a d 6 } ^ { a d 2 / a d 5 / a d 5 / a d 5 / a d 5 / a d 5 / a d 5 / a d 5 / a d 5 / a d 5 } ^ empty-mlist / { a d 4 / a d 41 / a d 42 / empty / a d 4 / a d 4 / a d 4 / a d 4 } ^ { a d 3 / a d 31 / a d 32 / empty / a d 3 / a d 3 / a d 3 / a d 3 } ^ { a d 2 / a d 21 / a d 22 / empty / a d 2 / a d 2 / a d 2 / a d 2 / a d 2 } ^ empty-mlist / { a d 1 / a d 7 / a d 1 / a d 1 } ^ empty-mlist / { a d 0 } ^ empty-mlist } / { { d 1 / d 0 } ^ empty-mlist / { d 4 / d 1 } ^ { d 3 / d 1 } ^ { d 2 / d 1 } ^ empty-mlist / { { d 4 / d 8 } / d 4 } ^ { { d 3 / d 7 } / d 3 } ^ { { d 2 / d 6 } / d 2 } ^ { { d 2 / d 5 } / d 2 } ^ empty-mlist / { d 17 / { d 4 / d 8 } } ^ { d 16 / { d 4 / d 8 } } ^ { d 15 / { d 4 / d 8 } } ^ { d 14 / { d 4 / d 8 } } ^ { d 13 / { d 3 / d 7 } } ^ { d 12 / { d 2 / d 6 } } ^ { d 11 / { d 2 / d 6 } } ^ { d 10 / { d 2 / d 6 } } ^ { d 9 / { d 2 / d 5 } } ^ empty-mlist / { d 17 / d 21 } ^ { d 12 / d 20 } ^ { d 11 / d 20 } ^ empty-mlist / { d 16 / d 21 } ^ { d 15 / d 21 } ^ { d 14 / d 21 } ^ { d 13 / d 21 } ^ { d 10 / d 20 } ^ { d 9 / d 20 } ^ empty-mlist / { d 16 / d 21 } ^ { d 15 / d 21 } ^ { d 10 / d 20 } ^ empty-mlist / { d 15 / d 21 } ^ { d 10 / d 20 } ^ empty-mlist / { d 17 / d 19 } ^ { d 16 / d 19 } ^ { d 15 / d 19 } ^ { d 14 / d 19 } ^ { d 13 / d 19 } ^ { d 12 / d 18 } ^ { d 11 / d 18 } ^ { d 10 / d 18 } ^ { d 9 / d 18 } ^ empty-mlist / { d 17 / d 16 } ^ { d 15 / d 15 } ^ { d 12 / d 10 } ^ empty-mlist / { d 17 / d 16 } ^ { d 15 / d 14 } ^ { d 12 / d 11 } ^ { d 11 / d 10 } ^ empty-mlist } } / nichts-abzuspeichern ^ e }
<>: A DB1xF
<>: P fst(DB1xF)
<>: A DB1
<>: P DB = DB1
<>: f
true

Anhang C

Visualisierung der Spezifikation

Die Struktur der Gesamtspezifikation kann mit dem OBSCURE-Visualisierungwerkzeug VIRUS (siehe [RM92a], [RM92b] und [RM92c]) visualisert werden. Diese Visualiserung wurde dem hinteren Einband dieser Arbeit lose beigefügt. (Die mangelnde Qualität beruht darauf, daß bisher keine direkte Ausgabe einer Visualisierung zu einem Drucker möglich ist, sondern nur auf einen Bildschirm. Die Bildschirmausgabe wurde durch Anwendung des Programms *xv* mühsam und unter Qualitätsverlust zur Ausgabe an einen Drucker aufbereitet.)

Dargestellt ist die Verschachtelungsstruktur der Gesamtspezifikation. Je ein benanntes Rechteck symbolisiert einen benannten Modul der Spezifikation (Ausnahme: Rechtecke mit der Beschriftung 'ATOMAR' repräsentieren unbenannte, atomare Module). Im rechten oberen Bildrand dieser Darstellung werden die verwendeten Symbole erläutert. Es ist deutlich zu erkennen, daß die Spezifikation der Aktivitätenebene weitaus kleiner ist, als die schematisch entwickelte Spezifikation der Datenmodellebene.

Index

Literaturverzeichnis

[ABH92] Serge Autexier, Christoph Benzmüller, and Ramses A. Heckler. Das Fallbeispiel UNIX: Dokumentation einer UNIX–Filesystem–Spezifikation mit OWEB. Interner Bericht (WP92/36), 1992. Universität des Saarlandes.

[AFHL92] Abdelwaheb Ayari, Stefan Friedrich, Ramses A. Heckler, and Jacques Loeckx. Das Fallbeispiel LEX. Interner Bericht (WP 92/39), 1992. Universität des Saarlandes.

[Aut93] Serge Autexier. HDMS–A und OBSCURE in KORSO — Die Funktionale Essenz von HDMS–A aus Sicht der algorithmischen Spezifikationsmethode — TEIL 2: Spezifikation des Datenmodells. Technischer Bericht A/05/93, Universität des Saarlandes, 1993. / unter Mitarbeit von Christoph Benzmüller und Ramses A. Heckler / unter Beratung von Stefan Conrad und Rudi Hettler /.

[Bal92] Helmut Balzert. *CASE. Systeme und Werkzeuge.* B.I. Wissenschaftsverlag, 1992.

[BCC+93] P. Baur, E. Canver, J. Cleve, R. Drexler, R. Förster, P. Göhner, H. Hauff, D. Hutter, P. Kejwal, D. Loevenich, W. Reif, C. Sengler, W. Stephan, M. Ullmann, and A. Wolpers. VSE – Verification Support Environment. In *Proceedings of the VIS 93*, 1993.

[BCN92] C. Batini, S. Ceri, and S. B. Navathe. *Conceptual Database Design: An Entity-Relationship Approach.* Benjamin/Cummings, Redwood City, CA, 1992.

[Be94] M. Broy and S. Jähnichen (eds.). Korso, correct software by formal methods. To appear as a LNCS volume, 1994.

[Ben93a] C. Benzmüller. Konzeptvorschlag zur SUNRISE-Diplomarbeit. März 1993.

[Ben93b] C. Benzmüller. Zwischenbericht zur SUNRISE-Diplomarbeit mit konkretisierter Beschreibung des weiteren Vorgehens. November 1993.

[Ben93c] Christoph Benzmüller. HDMS–A und OBSCURE in KORSO— Die Funktionale Essenz von HDMS–A aus Sicht der algorithmischen Spezifikationsmethode — TEIL 3: Spezifikation der atomaren Funktionen. Technischer Bericht A/06/93, Universität des Saarlandes, 1993. / unter Mitarbeit von Serge Autexier und Ramses A. Heckler / unter Beratung von Stefan Conrad und Rudi Hettler /.

[BFG+91] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, and K. Stølen. The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. Version 0.3. TUM–I9140, Technische Universität München, 1991.

[BG80] R. M. Burstall and J. A. Goguen. The semantics of CLEAR, a specification language. *LNCS*, 86:232–232, 1980.

[BHHW86] S. Biundo, B. Hummel, D. Hutter, and C. Walther. The Karlsruhe Induction Theorem Proving System. pages 672 – 674. Springer Verlag, 1986.

[Bid91] M. Bidoit. Development of modular specifications by stepwise refinement using the PLUSS specification language. Rapportt de Recherche du LIENS LIENS–91–9, Ecole Normale Supèrieure, 1991.

[BJ94] Manfred Broy and Stefan Jähnichen, editors. *Korrekte Software durch formale Methoden – Abschlußbericht des BMFT–Verbundprojekts KORSO.* KORSO/BMFT, 94.

[Boe88] B. W. Boehm. *A spiral model of software development and enhancement.* IEEE Computer, 1988.

[BSS93] R. Brill, J. Stahl, and M. Staemmler. SUNRISE *manual pages.* Fraunhofer-Institut Biomedizinische Technik (IBMT), St. Ingbert, 1993.

[BSSG93] R. Brill, J. Stahl, M. Staemmler, and K. Gersonde. SUNRISE II — a versatile environment for medical image and data processing. In *Visualisierung in der Medizin (Tagungsband).* Universität Freiburg, März 1993.

[Bud93] D. Budgen. *Software Design.* Addison-Wesley, 1993.

[CES89] UK Systems Security Confidence Levels, CESG Memorandum No. 3. Communications-Electronics Security Group, United Kingdom, January 1989.

[Che89] P. P.-S. Chen. The entity-relationship model: Toward a unified view of data. In J. Mylopoulos and M. L. Brodie, editors, *Readings in Artificial Intelligence and Databases*, pages 98–111. Kaufmann, San Mateo, CA, 1989.

[CHL94] 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.

[CK91] Peter Pin-Shan Chen and Heinz-Dieter Knöll. *Der Entity–Relationship–Ansatz zum logischen Systementwurf.* B.I. Wissenschaftsverlag, 1991.

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

[DeM78] T. DeMarco. *Structured Analysis and System Specification.* New York: Yourdan Press, 1978.

[DTI89] DTI Commercial Security Centre Evaluation Levels Manual, V22. Department of Trade and Industry, United Kingdom, February 1989.

[EGL89] Ehrich, Gogolla, and Lipeck. *Algebraische Spezifikation abstrakter Datentypen.* Teubner, 1989.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1, Equations and Initial Semantics.* Springer, 1985.

[EM90] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2, Module Specifications and Constraints.* Springer, 1990.

[FHM+91] Jürgen Fuchs, Annette Hoffmann, Liane Meiss, Joachim Philippi, Michael Stolz, Markus Wolf, and Jörg Zeyer. The *OBSCURE* Manual, Part I: Editing and Rapid Prototyping, Februar 1991. Universität des Saarlandes.

[FW83] P. Freeman and A. J. Wasserman, editors. *Software Design Techniques.* Silver Spring: IEEE Computer Society Press, 1983. 4th Edition.

[GL89] J. Grant and T.-W. Ling. Database representation and manipulation using entity-relationship database logic. In Z. W. Ras, editor, *Methodologies for Intelligent Systems, 4: Proc. of the Fourth International Symposium on Methodologies for Intelligent Systems*, pages 102–109. North-Holland, New York, 1989.

[GM86] N. Gehani and A. D. McGettrick, editors. *Software Specification Techniques.* International Computer Science Series, Addison-Wesley, 1986.

[Gog93] Martin Gogolla. An extended Entity-Relationship Modell. *Springer LNCS*, 1993.

[Hec93] Ramses A. Heckler. HDMS–A und OBSCURE in KORSO — Die Funktionale Essenz von HDMS–A aus Sicht der algorithmischen Spezifikationsmethode — TEIL 1: Einführung und Anmerkungen. Technischer Bericht A/04/93, Universität des Saarlandes, 1993. / unter Mitarbeit von Serge Autexier und Christoph Benzmüller / unter Beratung von Stefan Conrad und Rudi Hettler /.

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

[HNSE87] U. Hohenstein, L. Neugebauer, G. Saake, and H.-D. Ehrich. Three-level specification using an extended entity-relationship model. In *Proc. Informationsbedarfsermittlung und -analyse für den Entwurf von Informationssystemen.* Springer, 1987.

[Hof89] A. Hoffmann. Schnittstellenbeschreibung zwischen dem Benutzer und dem OBSCURE-System unter emacs. Interner Bericht (WP 14/88), Januar 1989. Universität des Saarlandes.

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

[Huf94] H. Hufschmidt. Spezifikation von Zugriffsrechten in OBSCURE. Interner Bericht (WP 93/44), Februar 1994. Universität des Saarlandes.

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

[ITS91] Kriterien für die Bewertung der Sicherheit von Systemen der Informationstechnik (ITSEC): Vorläufige Form der harmonisierten Kriterien. Amt für amtliche Veröffentlichungen der Europäischen Gemeinschaft, Brüssel, Luxemburg, Version 1.2 vom 28. Juni 1991.

[Jac83] M. Jackson. *System Development.* Englewood Cliffs: Prentice Hall, 1983.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice Hall, 1990.

[JWS94] Cai Jiamei, Markus Wolf, and Stefan Schlobach. The Translation from OBSCURE into ML. Interner Bericht (WP 93/43), Februar 1994. Universität des Saarlandes.

[Knu81] D. Knuth. *The Art of Computer Programming, Vol. 1,2,3.* Reading: Addison-Wesley, 1973,1973,1981.

[Lat92] Bernhard Latz. OBSCURE und INKA. Interner Bericht (WP 92/32), März 1992. Universität des Saarlandes.

[Leh90] Thomas Lehmann. A notion of implementation for the specification language OBSCURE. Interner Bericht (WP 90/27), Juni 1990. Universität des Saarlandes.

[Lev93] P. Leven. Spezifikation eines Algorithmus zur Zugriffskontrolle in OBSCURE. Interner Bericht (WP 93/46), Oktober 1993. Universität des Saarlandes.

[LEW] J. Loeckx, M. D. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. Wiley-Teubner. To appear 1995.

[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.

[Mee86] L. Meertens. Program specification and transformation. In *Proc. IFIP TC2 Working Conference on Program Specification and Transformation*. North-Holland, 1986.

[Mei89] L. Meiss. Schnittstellenbeschreibung zwischen der Semantischen Analyse und der Moduldatenbank. Interner Bericht (WP 13A/88), Januar 1989. Universität des Saarlandes.

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

[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.

[PST91] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z*. Prentice Hall, 1991.

[RM92a] Jochen Röhrig and Erik Mohr. Benutzerhandbuch VIRUS, VISCERAL-Projekt. Dokumentation zum Fortgeschrittenenpraktikum SS 1992, angefertigt am Lehrstuhl von Prof. Dr.-Ing. Loeckx, Universität des Saarlandes, 1992.

[RM92b] Jochen Röhrig and Erik Mohr. Projektdokumentationdokumentation VIRUS, VISCERAL-Projekt. Dokumentation zum Fortgeschrittenenpraktikum SS 1992, angefertigt am Lehrstuhl von Prof. Dr.-Ing. Loeckx, Universität des Saarlandes, 1992.

[RM92c] Jochen Röhrig and Erik Mohr. Systemdokumentation VIRUS, VISCERAL-Projekt. Dokumentation zum Fortgeschrittenenpraktikum SS 1992, angefertigt am Lehrstuhl von Prof. Dr.-Ing. Loeckx, Universität des Saarlandes, 1992.

[SB92] I. Schaumüller-Bichl. *Sicherheitsmanagement: Risikobewältigung in Informationstechnologischen Systemen.* BI-Wiss.-Verl., Mannheim, Leibzig, Wien, Zürich, 1992.

[SCS89] Catalogue de Critères Destinés à évaluer le Degré de Confiance des Systèmes d'Information. Service Central de la Sécurité des Systèmes d'Information, France, Juillett 1989.

[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.

[Sta93] J. Stahl. SUNRISE — *Entwurf und Implementation eines Software-Systems zur medizinischen Bild- und Meßdatenverarbeitung.* PhD thesis, Universität des Saarlandes, 1993.

[Sto91] Michael Stolz. Eine verzögerte Auswertung für algorithmische Spezifikationen. Die Theorie und ein Interpretierer fuer OBSCURE. Master's thesis, Universität des Saarlandes, 1991.

[SW83] D. T. Sannella and M. Wirsing. A kernel language for algebraic specification and implementation. *LNCS*, 158:413–427, 1983.

[TCS85] Trusted Computer Systems Evaluation Criteria. Deapartment of Defense, United States of Amerika, December 1985.

[Tre91] Ralf Treinen. Ein Kalkül für Algorithmische Spezifikationen. Master's thesis, Universität des Saarlandes, 1991.

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

[WH89] M. Wolf and M. Henz. Dokumentation des EXITUS-Fortgeschrittenenpraktikums SS 1989. Angefertigt am Lehrstuhl von Prof. Dr.-Ing. Loeckx, Leiter: S. Uhrig, Betreuer: A. Heckler. Universität des Saarlandes, 1989.

[Woo93] J. C. P. Woodcock. *Using Standard Z. Specification, Proof and Refinement.* Prentice Hall, 1993. 13-948472-8.

[YC79] E. Yordon and L. Constantine. *Structured Design.* Englewood Cliffs: Prentice Hall, 1979.

[You89] E. Yourdon. *Modern Structured Analysis.* Englewood Cliffs, N.J.: Yourdon Press, 1989.

[Zey89] J. Zeyer. Kontextbedingungen für OBSCURE. Interner Bericht (WP 89/13), Juni 1989. Universität des Saarlandes.

[Zey92] J. Zeyer. Erweiterung des OBSCURE-Systems. Interner Bericht (WP 92/31), Februar 1992. Universität des Saarlandes.

[ZSI89] Criteria for the Evaluation of Trusthworthiness of Information Technology (IT) Systems. German Information Security Agency (Bundesamt für Sicherheit in der Informationstechnik), Federal Repuplic of Germany, January 1989.

[ZSI90] Kriterien für die Entwicklung, Realisierung und Zulassung von Werkzeugen zur formalen Spezifikation und Verifikation (Anhang IT-Sicherheitskriterien). Zentralstelle für Sicherheit in der Informationstechnik (ZSI) im Auftrag der Bundesregierung, ZSI/BSI, Bonn, September 1990.