Forschungszentrum Karlsruhe
Technik und Umwelt
Wissenschaftliche Berichte
FZKA 5597

Entwurf eines Systems zur Erfassung und Weiterverarbeitung von Produktinformation bei der Herstellung von Mikrostrukturen

P. Wieland, C. Düpmeier, H. Eggert, K. P. Scherer

Institut für Angewandte Informatik

Juli 1995

Forschungszentrum Karlsruhe

Technik und Umwelt

Wissenschaftliche Berichte

FZKA 5597

Entwurf eines Systems zur Erfassung und Weiterverarbeitung von Produktinformation bei der Herstellung von Mikrostrukturen

P. Wieland*), C. Düpmeier, H. Eggert, K.P. Scherer

Institut für Angewandte Informatik

*) von der Fakultät für Maschinenbau der Universität Karlsruhe genehmigte Dissertation

Forschungszentrum Karlsruhe GmbH, Karlsruhe

1995

Forschungszentrum Karlsruhe GmbH
Postfach 3640, 76021 Karlsruhe

ISSN 0947-8620

Entwurf eines Systems zur Erfassung und Weiterverarbeitung von Produktinformation bei der Herstellung von Mikrostrukturen

Zusammenfassung

Das LIGA-Verfahren ist ein am Forschungszentrum Karlsruhe entwickeltes Verfahren zur Herstellung von Mikrostrukturen. Es wird heute am Institut für Mikrostrukturtechnik eingesetzt, um Mikrostrukturen herzustellen und dabei ständig erweitert und verbessert.

Um das komplexe LIGA-Verfahren mit seinen zahlreichen Einzelprozessen besser überschauen zu können, ist es entscheidend, ein System zur rechnergestützten Dokumentation des LIGA-Verfahrens zu realisieren. Dieses System ermöglicht die *Erfassung aller Prozeßschritte* des Verfahrens mit *allen Parametern, allen Materialdaten* sowie einer *Beschreibung der Ressourcen* (Ausrüstungsgegenstände).

Auch die Erfahrungen, die aus der Produktion von Mikrostrukturen gewonnen werden, stellen Fertigungswissen dar. Daher ist es notwendig, das System so zu konzipieren, daß auch die Sammlung von Information über *sämtliche* in der Vergangenheit *gefertigten Produkte* möglich ist.

In der vorliegenden Arbeit wird solch ein System, genannt *PRAXIS* (PRoduct And EXperimentation Information System) entworfen.

PRAXIS bietet die Möglichkeit, beliebige Werkzeuge für die Sammlung und Weiterverarbeitung der Information, sowie für die Unterstützung bzw. Durchführung der Prozeßschritte bzw. Aktivitäten einzubinden.

Diese Werkzeuge arbeiten auf einem integrierten Datenbestand, der von einem objektorientierten Datenbanksystem verwaltet wird.

Design of a system for collection and processing of product information during the manufacturing process of microstructures

Abstract

The LIGA-process (german acronym for Lithographie, Galvanik, Abformung) has been developed at the Institute for Microstructural Engineering (IMT) at the Karlsruhe Research Center for manufacture microstructures. Today, the IMT is employing the LIGA-technique to manufacture the mechanical parts of microsystems, meanwhile the technique is constantly being extended and improved.

In order to obtain improved surveillance over the complex LIGA-process with its numerous process steps, it is necessary to develop a computer based framework to summarize the overall LIGA-process. This system allows the collection of *all process steps* with *all parameters, all material information* and *all ressources* of the process.

Experience made during the production of microstructures represents manufacturing knowledge. Thus it is necessary to design the framework in such a way as to allow the collection of information about manufactured products.

This thesis contains the design of such a system called *PRAXIS* (PRoduct And EXperimentation Information System).

PRAXIS offers the project managers the possibility to embed arbitrary tools into the framework that grasp and process such information and support or perform the process steps or activities.

These tools use data integrated by an object-oriented database management system.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Die industrielle Fertigung ist heute durch zwei Merkmale geprägt: die zunehmende Komplexität der Fertigungsprozesse sowie der wachsende Automatisierungsgrad. Auf die Notwendigkeit, die dabei anfallende Information geeignet zu sammeln und zu verwalten, wurde mehrfach hingewiesen ([And93a],[GAP93], [Sch90],[And93b]).

Dies gilt auch für die LIGA-Technik[1] [Men93], ein am Forschungszentrum Karlsruhe entwickeltes Verfahren zur Herstellung von Mikrostrukturen. Das LIGA-Verfahren ist in seiner heutigen Realisierung noch wenig automatisiert. Es ist mit seinen über 250 Einzelprozessen jedoch derart komplex, daß auch hier die Notwendigkeit besteht, Information über den Prozeß in geeigneter Form rechnerunterstützt zu sammeln, zu verwalten und auszuwerten [Men93].

Das LIGA-Verfahren mit seinen Prozeßschritten bildet den Kern des Fertigungsprozesses, um den es in dieser Arbeit geht. Es wird durch Aktivitäten ergänzt, die durch die Erfassung und Verarbeitung von Information über die hergestellten Produkte notwendig werden.

Die Möglichkeiten einer strukturierte Sammlung von Information über den LIGA-Fertigungsprozeß wurde bereits in [Bra94] untersucht. Die Motivation dieser Arbeit war es allerdings nicht, ein System bereitzustellen, das eine Planung und Steuerung des Fertigungsprozesses erlaubt, sondern die strukturierte Sammlung von Fertigungswissen. So sind dort auch Prinzipien wie Parallelität und Synchronisation, die für eine effiziente Fertigungsplanung und -steuerung maßgebend sind, nicht berücksichtigt. Sie werden in der vorliegenden Arbeit aufgegriffen.

Fertigungswissen bedeutet nicht allein das Wissen über die Prozeßschritte des Fertigungsprozesses sowie die dabei relevanten Parameter. Auch die Erfahrun-

[1] LIGA steht für RöntgentiefenLIthographie, Galvanoformung und KunststoffAbformung, den Hauptschritten des LIGA-Verfahrens.

gen, die aus der Produktion von Mikrostrukturen gewonnen werden, stellen Fertigungswissen dar. Es muß somit ein Weg gefunden werden, um neben der Information über den Prozeß auch Information über die gefertigten Produkte zu sammeln, zu verwalten und auszuwerten.

Am Institut für Mikrostrukturtechnik (IMT) des Forschungszentrums Karlsruhe existiert ein System zur Sammlung und Auswertung von Information über Produkte, die mit Hilfe des LIGA-Fertigungsprozesses gefertigt werden. Auf Grundlage eines relationalen Datenbanksystems wurde ein Datenbankschema zur Archivierung von Produktinformation, die während der Produktion anfällt, konzipiert und umgesetzt. Darauf aufsetzend wurde eine Anwendung implementiert, die die Eingabe und bedingt auch eine Selektion von Daten ermöglicht. Eine Schwachstelle des bestehenden Systems ist die zu starke Einbeziehung von Prozeßinformation in das Produktmodell [WBS92a].

Als Folge daraus müssen das Produktmodell und die darauf aufsetzenden Anwendungen geändert werden, wenn sich die Prozeßinformation ändert. Da der Prozeß selbst ständig weiterentwickelt wird, verursacht diese Einbeziehung von Prozeßinformation in das Produktmodell eine permanente Anpassung von Produktmodell und Anwendungen. Diese ungünstigen Merkmale erzwingen eine flexiblere Modellierung.

Der Versuch, Prozeß- und Produktmodellierung stärker als bisher zu entkoppeln, um die angesprochenen Nachteile zu beseitigen, führt zu anderen Problemen ([WBS92b]), da Produkt- und Prozeßsemantik nicht disjunkt sind. Diese Probleme können, wie in dieser Arbeit gezeigt wird, durch eine objektorientierte Modellierung ([Mey88]) vermieden werden. Die objektorientierte Modellierung bietet sich auch deshalb an, weil objektorientierte Datenstrukturen die Vielzahl an zu verarbeitenden Datentypen[2], die bei der Herstellung von LIGA-Mikrostrukturen anfallen, adäquater beschreiben können.

Möglichkeiten für die Modellierung eines System zur Erfassung und Weiterverarbeitung von Produktinformation bei der Herstellung von Mikrostrukturen (**PR**oduct **A**nd E**X**perimentation **I**nformation **S**ystem, kurz ***PRAXIS***) werden in der vorliegenden Arbeit aufgezeigt.

Benutzt man ein System, das Information sowohl über den Prozeß als auch über die Produkte verwaltet, so liegt es nahe, dieses System auch zur Optimierung der Abläufe in der Produktion einzusetzen.

1.2 Die LIGA-Technik

Die LIGA-Technik ist ein Verfahren, das am Forschungszentrum Karlsruhe Anfang der 80er Jahre zur Isotopentrennung nach dem Trenndüsenverfahren entwickelt wurde [BM91a]. Nach ständigen Weiterentwicklungen und Verbesserun-

[2]Neben Zeichenketten und numerischen Werten müssen fortan auch Zeichnungen, Dokumente, geometrische Objekte und ähnlich komplexe Daten verwaltet werden.

gen werden heute mit Hilfe der LIGA-Technik Mikrokomponenten hergestellt, die Ausgangsbasis für die Fertigung kompletter Mikrosysteme sind.

Der schematische Ablauf des LIGA-Verfahrens ist in [BM91a] gezeigt. Nach dem CAD-Design wird mit Hilfe eines Elektronenstrahlschreibers (ESS) eine primäre Maske (Chrommaske) erzeugt, aus der eine Röntgenmaske gefertigt wird. Diese Röntgenmaske ist Ausgangsbasis für die sich anschließende Röntgentiefenlithographie, eines zentralen Prozeßschritts des LIGA-Verfahrens [Men93]. Dabei wird ein Resist, das mit der Röntgenmaske abgedeckt wurde, bestrahlt. Die Bestrahlung des Resists durch energiereiche, parallele Röntgenstrahlung ist eine Voraussetzung für die hohen Aspektverhältnisse, die Mikrostrukturen auszeichnen, die nach dem LIGA-Verfahren gefertigt sind. Die Röntgenstrahlung wird von einer Synchrotronstrahlenquelle erzeugt. Nach der Belichtung werden – je nach angewendetem Resist-Entwicklersystem – die belichteten oder unbelichteten Teile des Resists durch Ätzen entfernt. Die entstehenden Hohlräume der Resiststruktur werden galvanisch gefüllt, es entsteht ein Formeinsatz. Dieser Formeinsatz kann anschließend für die Herstellung der eigentlichen Mikrostrukturen aus Metall, Kunststoff oder Keramik verwendet werden.

Diese grobe Beschreibung des Ablaufs des LIGA-Verfahrens läßt sich weiter verfeinern, wobei man zu einer detaillierten Beschreibung der Einzelprozesse gelangt.

1.3 Das Ziel der Arbeit

Um das LIGA-Verfahren mit seinen Einzelprozessen besser überschauen zu können, ist es entscheidend, eine rechnergestützte Dokumentation des LIGA-Verfahrens zu realisieren. Dazu gehört die Erfassung *aller* Prozeßschritte des LIGA-Verfahrens mit *allen* Parametern, *allen* Materialdaten sowie einer Beschreibung der Ressourcen (Ausrüstungsgegenstände) in *PRAXIS*.

Da die Erfahrung, die mit der realen Fertigung von Produkten gewonnen wird Qualitätswissen darstellt, ist es notwendig, das System so zu konzipieren, daß auch Information über die gefertigten Produkte gesammelt und verwaltet werden kann. *PRAXIS* soll es ermöglichen, über *sämtliche* in der Vergangenheit hergestellte Produkte die *vollständige* Produktionshistorie mit der dabei entstandenen Produktinformation zugänglich zu machen.

Dazu wird die anfallende Informationsmenge zunächst von drei unterschiedlichen Sichtweisen aus analysiert:

- von der Prozeßsicht aus, d.h. dem LIGA-Fertigungsprozeß aus ganzheitlicher Sicht mit allen für den Prozeß möglichen Fertigungsalternativen,
- von der Produktsicht aus, d.h. mit aller Information, die bei der Produktion eines ganz bestimmten Produktes mit Hilfe des LIGA-Verfahrens anfällt,

- sowie von der Produktionssicht aus, d.h. einer die vorhandenen Produktsichten koordinierenden Sicht mit dem Ziel des Projektmanagements.

Diese Sichtweisen bilden die Schnittstellen, an denen der Anwender mit *PRAXIS* arbeitet. Der Anwender sieht – je nach durchzuführender Tätigkeit – eine dieser drei Systemsichten.

Für die Sammlung und Weiterverarbeitung von Produktinformation steht eine breite Palette an informationsverarbeitenden Werkzeugen zur Verfügung. Diese Palette wird ständig erweitert. Neue Werkzeuge kommen hinzu, bestehende werden verbessert. Erforderlich ist ein System, das leicht auf sich ändernde Anforderungen angepaßt werden kann. *PRAXIS* muß die Möglichkeit bietet, beliebige Werkzeuge zur Erfassung und Weiterverarbeitung von Information einzubinden. Es muß auch sichergestellt werden, daß die unterschiedlichsten Werkzeuge auf dem Datenbestand arbeiten können, ohne daß der Anwender sich um die Frage kümmern muß, in welcher Weise die Daten für die Werkzeuge bereitgestellt werden.

Will man arbeitsintensive und fehleranfällige Mehrfacherfassung und redundante Speicherung der Information über den LIGA-Prozeß sowie die gefertigten Produkte vermeiden, muß man die Daten in integrierter Form in einer Datenbank abgelegen und von einem Datenbanksystem verwalten lassen. Aus der Notwendigkeit der vorher erwähnten objektorientierten Datenstrukturierung muß dazu ein objektorientiertes Datenbanksystem verwendet werden.

Die Vorteile eines Systems zur Erfassung und Weiterverarbeitung von Information bei der Fertigung von LIGA-Mikrostrukturen sind folgende:

- Durch die Prozeßsicht bekommt man eine Dokumentation des LIGA-Fertigungsprozesses. Beschrieben werden alle möglichen Prozeßalternativen. Aus diesen Alternativen wird bei der Fertigung jedes Produkts eine für diesen speziellen Fall geeignete Fertigungsvariante gewählt.

- Durch die Produktsicht erhält man die Historie der Fertigungsabläufe jedes Produkts. Will man ein neues Produkt fertigen, kann man auf bereits in der Vergangenheit gemachte und dokumentierte Erfahrungen des Fertigungsablaufs ähnlicher Produkte zurückgreifen.

  Eine Dokumentation des Fertigungsablaufs für Produkte wird auch aus Gesichtspunkten der Qualitätssicherung immer bedeutender.

- Durch die Produktionssicht hat man die Möglichkeit, die Fertigung von Produkten im Fertigungsablauf zu planen und zu optimieren. Dies wird insbesondere im Hinblick auf eine industrielle Fertigung (Serienfertigung) interessant.

Das Ziel dieser Arbeit ist es, in einem ersten Schritt die Anforderungen an *PRAXIS* als einem System zur Erfassung und Weiterverarbeitung von Information

(mit dem Schwerpunkt der Produktinformation) bei der Herstellung von LIGA-Mikrostrukturen herauszuarbeiten. Dies geschieht getrennt nach dem oben beschriebenen Komponenten:

- Darstellung und Bearbeitung der Systemsichten.
- Einbindung von beliebigen Werkzeugen und Integration der Daten.
- Verwaltung der integrierten Daten durch ein objektorientiertes Datenbanksystem.

In einem zweiten Schritt wird *PRAXIS* modelliert.

1.4 Gliederung der Arbeit

Die Architektur von *PRAXIS* besteht aus drei Hauptkomponenten: einem Werkzeug zur Visualisierung der Systemsichten (je nach Aufgabe die Prozeß-, eine Produkt-, oder die Produktionssicht), einer Toolbox für die Anbindung von Werkzeugen, sowie einem objektorientierten Datenbanksystem.

Diese drei Komponenten mit denen an sie gestellten Anforderungen werden in den folgenden drei Kapiteln näher erläutert.

Kapitel 2 beschreibt die drei Systemsichten Prozeßsicht, Produktsicht und Produktionssicht.

Kapitel 3 betrachtet die erweiterbare Sammlung von Werkzeugen, mit denen während der Produktion gearbeitet und Information gewonnen und weiterverarbeitet wird.

Kapitel 4 geht auf das objektorientierte Datenbanksystem ein. Dieses soll das integrierte Datenmodell samt Daten sowie die Systemsichten verwalten.

Kapitel 5 vertieft die in den vorausgegangenen Kapiteln behandelten Konzepte durch die Einführung von Partialmodellen für den Entwurf und die Fertigung von Mikrostrukturen. Kapitel 5.1 beschreibt die Aktivitätenkette vom CAD-Entwurf einer Mikrostruktur bis zur Vermessung der gefertigten Masken und dem dazu benötigten Partialmodell der Geometrie. In Kapitel 5.2 wird das Partialmodell für Teile der Fertigungssicht vorgestellt.

In Kapitel 6 wird ein Ansatz der Architektur von *PRAXIS* vorgestellt, die es ermöglicht, sämtliche Produktinformation, die bei der Herstellung von Mikrostrukturen anfällt, zu erfassen und weiterzuverarbeiten.

Nachdem die Systemanforderungen in Kapitel 2 bis 4 dargelegt wurden, und die Partialmodelle eingeführt sind, enthält Kapitel 7 abschließend einen Ansatz zur objektorientierten Modellierung von *PRAXIS*.

Kapitel 8 enthält die Zusammenfassung der Arbeit sowie einen Ausblick.

Der Anhang besteht aus einer Beschreibung der Vorbedingungen aus dem CAD-Entwurf (Anhang A.1), der spezifischen Formate IGES (Anhang A.2) und GDSII (Anhang A.3) sowie der Funktionsweise der Konverter für diese Formate, die für das Partialmodell der Entwurfsschritte für Mikrostrukturen (Kapitel 5.1) von Bedeutung sind.

Anhang B enthält eine Beschreibung der Implementierung eines Werkzeugs zur Visualisierung des Partialmodells für die Fertigung von Mikrostrukturen ((Kapitel 5.2), sowie eine Beschreibung der dabei implementierten Klassen.

Anhang C besteht aus der Beschreibung Klassen als Ergebnis der objektorientierten Modellierung von *PRAXIS*. Zur Beschreibung wurde die Objektdefinitionssprache ODL (siehe Kapitel 4.2.2) verwendet.

Für die Modellierung von *PRAXIS* wurde eine eigene Sprache entwickelt. Anhang D schließt die Arbeit mit der Auflistung der Modellierungssprache von *PRAXIS* und einem Beispiel ab.

Kapitel 2

Die unterschiedlichen Systemsichten

Ein wesentlicher Teil eines jeden Systems ist die Schnittstelle zu dessen Benutzern bzw. Anwendern. Die Schnittstelle muß – evtl. für mehrere unterschiedliche Anwendergruppen getrennt – eine dem Anwender eingängige Sicht auf das System, die dort enthaltene Information sowie die dort ablaufenden Verarbeitungsvorgänge bieten. Für ein System zur Unterstützung der Informationsverarbeitung im Zusammenhang mit der LIGA-Technik (*PRAXIS*) wurden folgende drei Systemsichten ausgearbeitet:

- Die Prozeßsicht. Sie beschreibt den LIGA-Entwurfs-, Fertigungs- und Qualitätssicherungsprozeß aus ganzheitlicher Sicht, mit *allen* möglichen Fertigungsalternativen, *allen* Parametern, *allen* Materialdaten sowie den Ressourcen (Ausrüstungsgegenstände).

- Die Produktsicht. Sie enthält sämtliche Information, die bei der Produktion *eines ganz bestimmten* Produkts anfällt.

- Die Produktionssicht. Sie ist eine die vorhandenen Produktsichten koordinierenden Sicht mit dem Ziel des Projektmanagements.

Sie bilden im späteren System *PRAXIS* die Schnittstellen, an denen der Benutzer mit *PRAXIS* arbeitet. Diese drei Sichten werden im Anschluß erläutert. Kapitel 2.4 erläutert anschließend, wie diese drei Sichten zusammenhängen.

2.1 Die Prozeßsicht

Das LIGA-Verfahren mit seinen Prozeßschritten bildet die Ausgangsbasis für den Prozeß, um den es in dieser Arbeit geht. Darin enthalten sind Schritte zum Entwurf, der Fertigung sowie der Qualitätssicherung von Mikrostrukturen.

Der Begriff Prozeßschritt, d.h. ein Abschnitt des LIGA-Verfahrens, wird im folgenden durch den allgemeineren Begriff der Aktivität erfaßt.

> Eine **Aktivität** kann entweder die Durchführung eines Prozeßschritts des LIGA-Verfahrens für den Entwurf, die Fertigung oder die Qualitätssicherung einer Mikrostruktur sein, oder die Durchführung einer Tätigkeit, die im Zusammenhang mit einem Prozeßschritt als Vor- oder Nachbereitung dessen notwendig wird.

So wird z.B. bei der Durchführung des Entwurfsschritts *CAD-Entwurf* eine IGES-Datei ([IGE91]) erzeugt. Diese Datei muß, bevor sie von dem nachfolgenden Fertigungsschritt *Maskenerstellung mit dem Elektronenstrahlschreiber (ESS)* weiterverarbeitet werden kann, auf einen anderen Rechner portiert und anschließend in eine GDSII-Datei ([Cal84]) transformiert werden (Abbildung 2.3 auf Seite 12). Diese Schritte der Portierung und Transformation werden in der Prozeßsicht ebenfalls erfaßt und als Aktivitäten dargestellt.

Unter dem in dieser Arbeit zu modellierenden Prozeß (auch Fertigungs-, Herstellungs- oder Produktionsprozeß) verstehen wir eine Abfolge von Aktivitäten (ein sogenanntes Aktivitätennetz), die zu einem Produkt führen. Wie wir in Kapitel 2.2 sehen werden, muß es sich dabei nicht unbedingt um eine Mikrostruktur als Endprodukt handeln. Vielmehr werden auch die Zwischenprodukte erfaßt.

Um diesen Prozeß zu modellieren, müssen unterschiedliche Kennzeichen des Prozesses, wie

- Hierarchiebildung bei Aktivitäten
- Kausale Vorgänger- /Nachfolgerbeziehung von Aktivitäten
- Alternativaktivitäten
- Nebenläufigkeit von Aktivitäten
- Ein- und Ausgabeobjekte von Aktivitäten

berücksichtigt werden. Diese werden im folgenden anhand des in Abbildung 2.1 gezeigten Modells beschrieben.

Hierarchiebildung, Vorgänger-/Nachfolgerbeziehung und Alternativaktivitäten sind bereits in [Bra94] modelliert, neu in dem in dieser Arbeit vorgestellten Modell sind nebenläufige Aktivitäten, sowie die Berücksichtigung der Ein- und Ausgabeobjekte.

2.1.1 Hierarchie

Da ein Prozeß aus einer Vielzahl von Aktivitäten besteht, wird man leicht den Überblick verlieren, wenn es keine Möglichkeit gibt, Details zu verbergen und

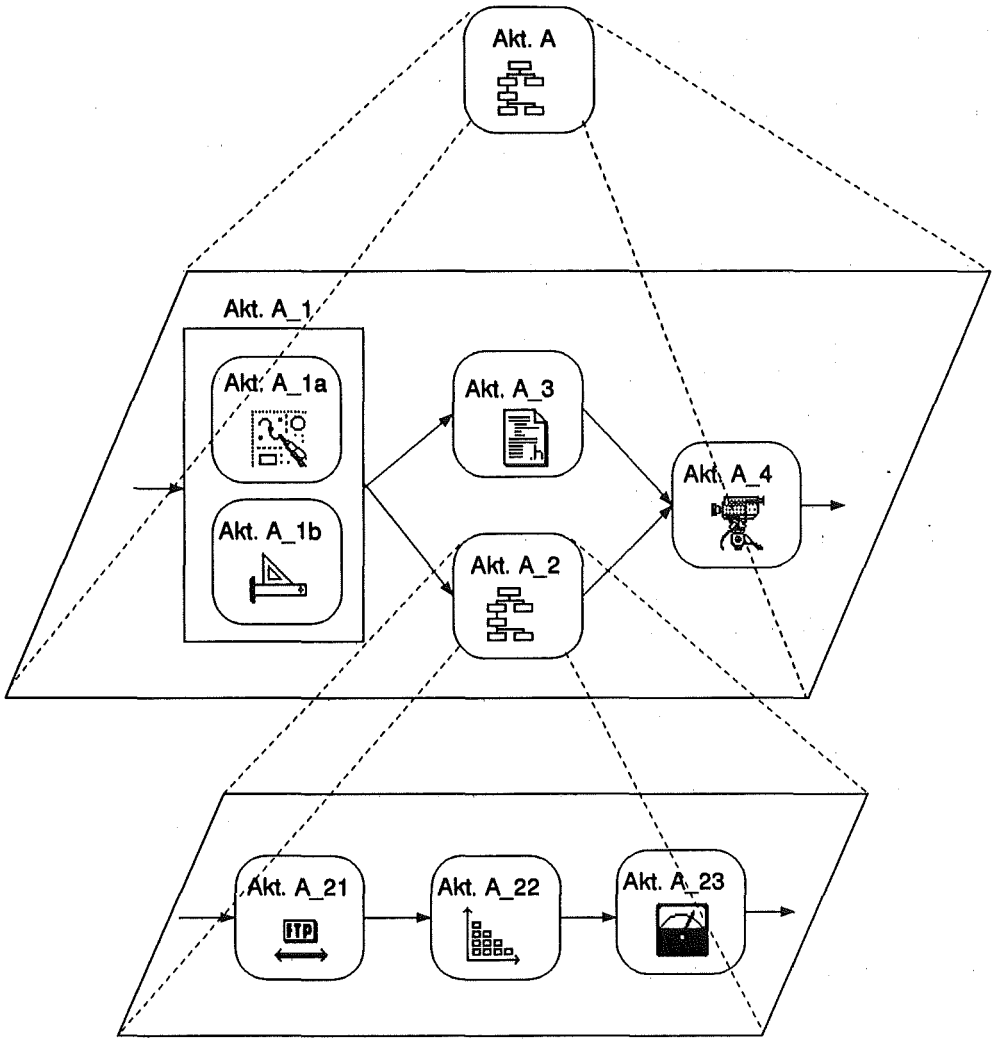


Abbildung 2.1: Darstellung des Prozeßmodells

nur bei Bedarf darzustellen. Die Einführung einer Hierarchie, bei der auf verschiedenen Hierarchiestufen ein unterschiedlicher Detaillierungsgrad dargestellt wird, bietet diese Möglichkeit. Aktivitäten eines Prozesses können auf unterschiedlichen Hierarchiestufen angesiedelt sein, d.h. es besteht die Möglichkeit, Aktivitäten zu einer Oberaktivität zusammenzufassen (Vergröberung), ebenso wie es möglich ist, eine Aktivität in Unteraktivitäten zu unterteilen (Verfeinerung).

In Abbildung 2.1 ist beispielsweise die Aktivität A unterteilt in die Unteraktivitäten $A_1, \ldots, A_4$. Die Aktivitäten A_{21}, A_{22} und A_{23} sind in der Oberaktivität A_2 zusammengefaßt.

Die Hierarchiestufe der Aktivität A ist demnach höher als die von $A_1, \ldots, A_4$. Wir reden – unabhängig von der Hierarchiestufe – ausschließlich von Aktivitäten.

Es bleibt festzuhalten, daß die Tiefe dieser Hierarchisierung nicht notwendigerweise überall gleich groß sein muß. Es ist durchaus möglich, daß zwar Aktivität A_2, wie in Abbildung 2.1 weiter verfeinert wird, während hingegen Aktivität A_4 auf der gleichen Ebene ihre unterste Hierarchiestufe erreicht hat.

2.1.2 Kausale Vorgänger-/Nachfolgerbeziehungen

Für die Durchführung eines Prozesses ist es notwendig, die Aktivitäten in einer definierten Reihenfolge ablaufen lassen zu können. D.h. man kann für eine Aktivität eine oder mehrere Aktivitäten bestimmen, die kausal davor bzw. danach ausgeführt werden.

Vorgänger-/Nachfolgerbeziehungen sind immer auf der gleichen Hierarchiestufe anzusiedeln, und dort ausschließlich in der Verfeinerung derselben Aktivität.

Aktivität A_{21} aus Abbildung 2.1 hat keinen Vorgänger, auch nicht Aktivität A_1, da sich die beiden Aktivitäten nicht auf der gleichen Hierarchiestufe befinden.

Aktivität A_{21} ist aber Verfeinerung von Aktivität A_2, welche als Vorgänger Aktivität A_1 besitzt. Also wird die zuletzt ausgeführte Aktivität einer möglichen Verfeinerung (z.B. A_{1x}) von Aktivität A_1 *vor* der Aktivität A_{21} ausgeführt. Dabei ist jedoch A_{1x} nicht als Vorgängeraktivität von Aktivität A_{21} modelliert, da Aktivität A_{1x} nicht aus der gleichen Aktivität heraus verfeinert wurde, wie Aktivität A_{21}.

2.1.3 Alternativaktivitäten

Da es möglich ist, bestimmte Fertigungsziele mit Hilfe von unterschiedlichen Aktivitäten (oder Aktivitätsnetzen) zu erreichen, können in der Prozeßbeschreibung Alternativen zu einzelnen Aktivitäten bzw. ganzen Aktivitätsnetzen enthalten sein. In der Prozeßsicht werden alle alternativen Aktivitäten erfaßt.

In Abbildung 2.1 besitzt die Aktivität A_1 zwei Alternativen, eine Aktivität A_{1a} und eine Aktivität A_{1b}.

Alternativaktivitäten können unabhängig voneinander weiter verfeinert werden.

2.1.4 Nebenläufigkeit von Aktivitäten

Oftmals ist es möglich, Aktivitäten zeitlich parallel auszuführen, falls sie voneinander unabhängig sind. In Abbildung 2.1 werden nach Abarbeitung von Aktivität A_1 (entweder in Form der Alternative A_{1a} oder der Alternative A_{1b}) zwei Aktivitäten parallel und unabhängig voneinander abgearbeitet: die Aktivitäten A_2 und A_3. Dieses Vorgehen erzwingt eine Synchronisation des Ablaufs. Mit der Ausführung von Aktivität A_4 darf erst begonnen werden, wenn beide Vorgängeraktivitäten A_2 und A_3 abgeschlossen sind.

2.1.5 Ein- und Ausgabeobjekte von Aktivitäten

Auf die Eigenschaft, daß zur Systembeschreibung und damit zur Systemmodellierung neben dem Erfassen der Aktivitäten auch das Erfassen der zugehörigen Daten eine wesentliche Bedeutung hat, weist schon [LSTK83] hin. Dort wird mit SADT (Structured Analysis and Design Technique [RS77]) eine Technik vorgestellt, um sowohl ein Tätigkeitsmodell als auch ein Datenmodell zu beschreiben. In der Terminologie des Tätigkeitsmodells werden Eingangsdaten, unter Einbeziehung von Kontrolldaten und Hilfsmechanismen, zu Ausgangsdaten verarbeitet.

Eine an SADT angelehnte Technik kann auch hier verwendet werden (Abbildung 2.2).

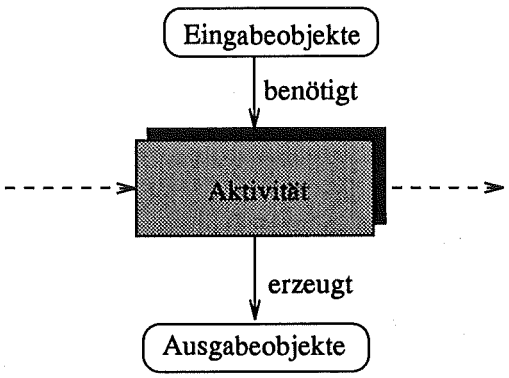


Abbildung 2.2: Zusammenhang zwischen Aktivitäten und Ein- bzw. Ausgabeobjekten

Eine Aktivität benötigt zu Ausführung bestimmte Eingabeobjekte. Es wird hier bewußt keine Beschränkung auf Daten vorgenommen, da z.B. für eine Aktivität *Vermessung* ein reales Objekt z.B. in Form einer Mikrostruktur zur Verfügung stehen muß. Es handelt sich bei den Eingaben einer Aktivität nicht ausschließlich um Daten, sondern ganz allgemein um Eingabeobjekte. Analoges gilt für Ausgabeobjekte, die bei der Ausführung von Aktivitäten erzeugt werden können.

Dadurch, daß ein Ausgabeobjekt einer Aktivität zum Eingabeobjekt einer nachfolgenden Aktivität werden kann, erhält man einen dem Datenfluß analogen Objektfluß durch das System.

Die Durchführung einer Aktivität ist in der Regel an Ressourcen (Ausrüstungsgegenständen) gebunden. Die getrennte Modellierung dieser Ressourcen ist für die Produktionssicht (Kapitel 2.3) relevant, die sich unter anderem die Koordination und optimale Ausnutzung dieser Ressourcen zum Ziel setzt.

Für die Prozeßsicht können ausschließlich rein klassifizierende Angaben über den Datentyp der Objekte gemacht werden. In der Prozeßsicht ist z.B. festzulegen, daß eine Aktivität *CAD-Entwurf* als Eingabeobjekt eine Spezifikation vom Datentyp *Dokument* hat und als Ausgabeobjekt eine IGES-Datei, sowie eine

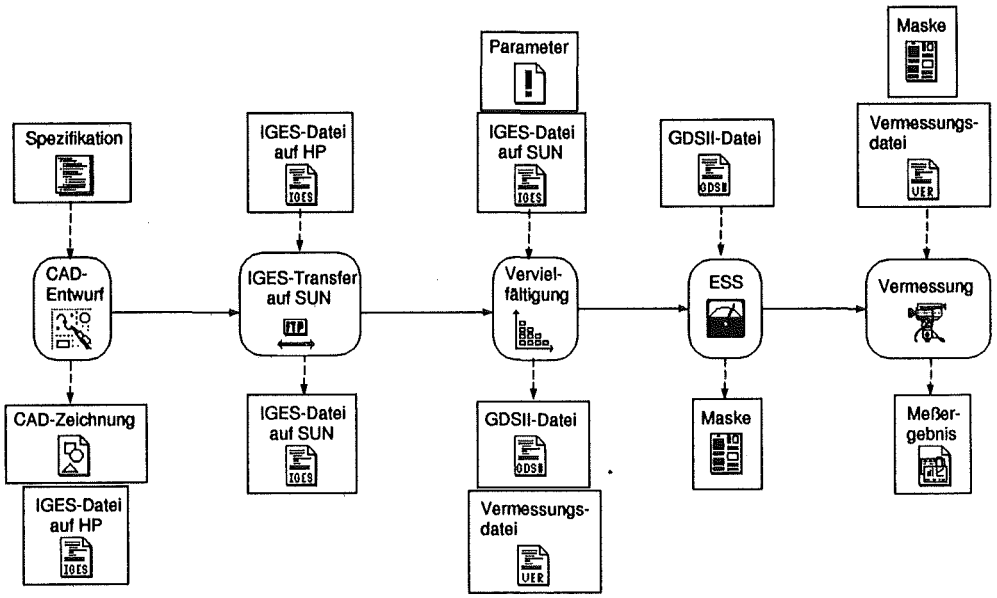


Abbildung 2.3: Prozeßsicht einer Aktivitätenkette

CAD-Zeichnung von Datentyp *Datei* erzeugt (siehe Abbildung 2.3).

Information über die Ausprägungen der einzelnen Objekte, d.h. welche Spezifikation in einen speziellen CAD-Entwurf einfließt oder welche IGES-Datei dabei erzeugt wird, ist produktspezifisch. Aus diesem Grund kann die Klassifizierung der Objekte erst in der Produktsicht instantiiert werden.

2.2 Die Produktsicht

In dem hier vorgestellten Modell ist ein Produkt ein Objekt, das einen Prozeß durchläuft. Der Prozeß wirkt durch seine Aktivitäten auf das Produkt, bis dieses seine endgültige Gestalt annimmt. Dabei ist wichtig herauszuheben, daß unter Produkt nicht nur das Endprodukt verstanden wird, sondern auch sämtliche Zwischenprodukte auf dem Weg zum Endprodukt[1]. Ein Produkt ist demnach ein Objekt, das durch die Abarbeitung eines *beliebigen Ausschnitts* eines Prozeßnetzes entsteht.

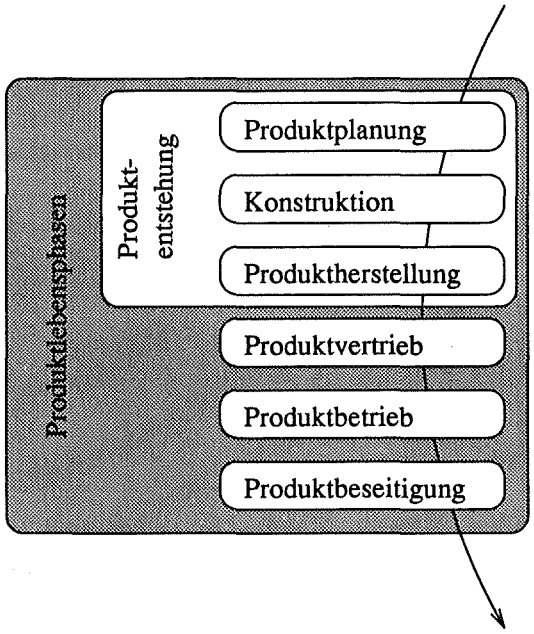


Abbildung 2.4: Produktlebenszyklus (aus [GAP93])

Ein Produkt durchläuft bestimmte Lebensphasen, wie dies in Abbildung 2.4 dargestellt ist. Für die Modellierungsanforderungen in dieser Arbeit werden ausschließlich die Phasen *Produktplanung, Konstruktion* und *Produktherstellung*, d.h. die Phasen der sogenannten Produktentstehung betrachtet. Alle nachfolgenden Phasen wie Vertrieb, Betrieb und Beseitigung des Produkts werden hier nicht berücksichtigt. Da im folgenden kein Unterschied gemacht wird, in welcher Phase des Produktlebenszyklus man sich befindet, sondern in allen Phasen davon ausgegangen wird, daß irgendwelche Aktivitäten durchgeführt werden, ist die nachträgliche Einbeziehung der noch fehlenden Phasen im Sinne des hier vorgestellten Modells möglich.

Die in Kapitel 2.1 betrachtete Prozeßsicht ist nicht produktabhängig. In ihr waren demnach sämtliche für den Prozeß denkbaren Aktivitäten mit allen Alternativen enthalten. Will man jedoch ein Produkt modellieren, so benötigt man

[1] Unter Produkt wird also nicht nur das gefertigte Endprodukt, z.B. eine Mikrostruktur oder gar ein komplettes Mikrosystem verstanden, sondern auch alle Zwischenprodukte wie Zwischenmasken, Arbeitsmasken u.ä.

nur eine Untermenge der in der Prozeßsicht enthaltenen Aktivitäten, nämlich genau die Aktivitäten, die zur Herstellung genau dieses Produkts dienen. Die Aktivitäten, die im Produktlebenszyklus eines Produkts enthalten sind, sind produktspezifisch.

Folgende Konzepte sind in der Produktsicht zu finden:

- Hierarchiebildung der an der Produktion beteiligten Aktivitäten
- Kausale Vorgänger- /Nachfolgerbeziehung der an der Produktion beteiligten Aktivitäten
- Nebenläufigkeit der an der Produktion beteiligten Aktivitäten
- Ein- und Ausgabeobjekte der an der Produktion beteiligten Aktivitäten
- Versionen von Produktdaten
- Zustand und Wiederholung von Aktivitäten
- Zustand des Produkts

2.2.1 Hierarchie

Die Aktivitäten zur Herstellung eines Produkts können – wie von der Prozeßsicht her bekannt – auf unterschiedlichen Hierarchiestufen erfaßt werden. Die so entstehende Hierarchie bietet Sichten unterschiedlicher Granularität auf die Herstellung des Produkts.

2.2.2 Kausale Vorgänger-/Nachfolgerbeziehungen

Auch bei der Herstellung eines speziellen Produkts haben Aktivitäten – wie die Beziehungen bei der Prozeßsicht – kausale Vorgänger- und Nachfolgerbeziehungen zu anderen Aktivitäten. Semantisch gelten die gleichen Einschränkungen wie bei der Prozeßsicht.

2.2.3 Alternativaktivitäten

Die Modellierung von Alternativen für bestimmte Aktivitäten spielen für die Produktsicht – im Gegensatz zu Alternativen in der Prozeßsicht – keine Rolle. Für ein Produkt interessieren nur die Aktivitäten, die ein Produkt im Laufe seines Lebenszyklus tatsächlich durchlaufen hat, nicht jedoch die Aktivitäten, die prinzipiell alternativ dazu auch hätten durchlaufen werden können. Diese Information ist der zur Produktion dieses Produkts gültigen Prozeßsicht zu entnehmen.

2.2.4 Nebenläufigkeit von Aktivitäten

Nebenläufigkeit von Aktivitäten sind für die Produktsicht – wie auch Nebenläufigkeiten in der Prozeßsicht – ebenfalls von Bedeutung. Die Produktsicht wird, wie später zu sehen ist, in das modellierte System eingebettet und dient als Vorgehensmodell bei der Produktherstellung. Die Möglichkeit einer Modellierung von unabhängig voneinander ausführbaren Aktivitäten ist maßgebend für eine zeitoptimale Fertigung des Produkts.

2.2.5 Ein- und Ausgabeobjekte von Aktivitäten

Zusätzlich zu der produktspezifischen Sequenz an Aktivitäten treten bei der Produktsicht die Ausprägungen der von jeder Aktivität für genau dieses Produkt verwendeten und erzeugten Objekte auf. Konnte man bei der Prozeßsicht nur festlegen, von welchem *Typ* Ein- und Ausgabeobjekte, verwendete Materialien und benötigte Ressourcen sind, so werden in der Produktsicht für jedes Produkt spezifisch die entsprechenden *Ausprägungen* sichtbar.

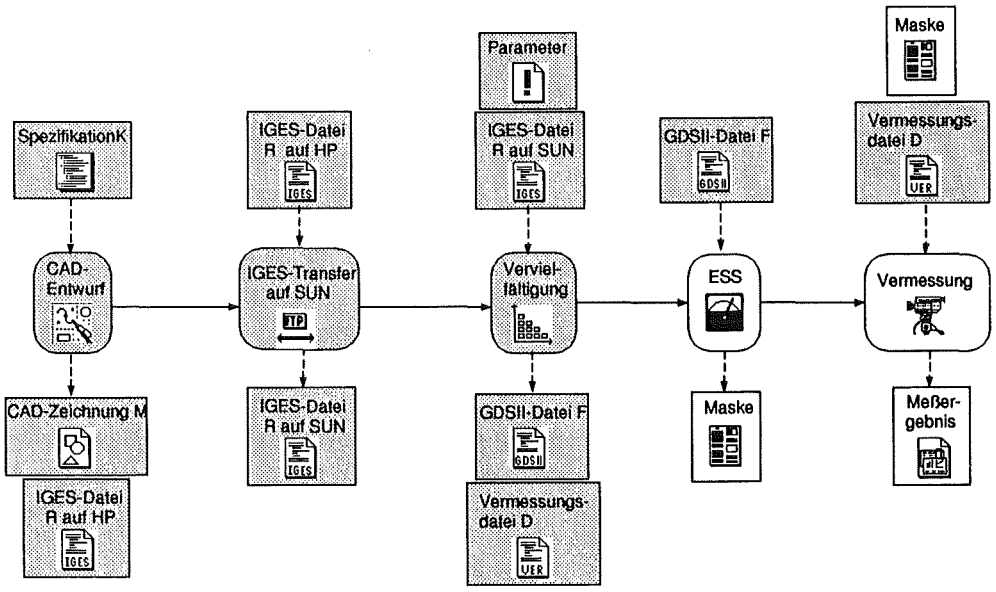


Abbildung 2.5: Produktsicht einer Aktivitätenkette

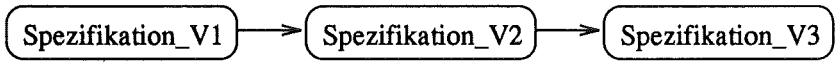
Die in Abbildung 2.5 dargestellte Produktsicht unterscheidet sich von der entsprechenden Prozeßsicht (Abbildung 2.3) dadurch, daß Eingabe- und Ausgabeobjekte reale Objekte und damit Ausprägungen der zuvor festgelegten Klassifizierung sind. Abbildung 2.5 zeigt z.B. anstelle der Klassifizierung *Spezifikation* die dem jeweilige Produkt zugrundeliegende Instanz (z.B. *Spezifikation K*).

2.2.6 Versionen von Produktdaten

Verändert man Daten, z.B. in einer Datenbank, so resultiert daraus i.a. ein Überschreiben der alten Daten. Will man das vermeiden, z.B. weil der Verlauf eines evolutionären Vorgangs wie die Erstellung einer Spezifikation von Interesse ist, so muß man die Verwaltung von verschiedenen Ausprägungen desselben Objekts in Form von Versionen unterstützen.

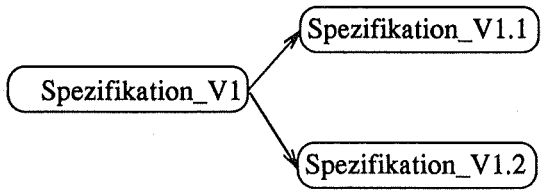
Nach [AASW94] kann man folgende 3 Arten der Versionierung unterscheiden:

1. *Lineare Versionierung*, bei denen Objektmodifikationen gespeichert werden, die in einem bestimmten Zeitraum durchgeführt werden. Interessiert beispielsweise das Aussehen einer Spezifikation zu diskreten Zeitpunkten während ihrer Entwicklung, so wird man diese als lineare Version ablegen:



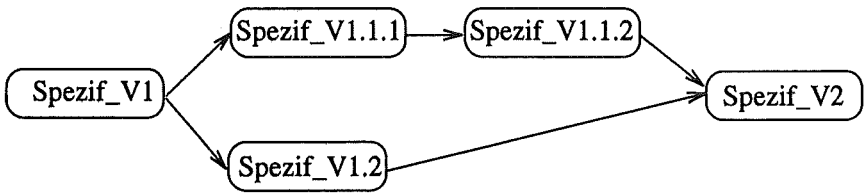
   Es entstehen nacheinander die Versionen V_1, V_2 und V_3 der Spezifikation.

2. Insbesondere Entwurfsanwendungen benötigen die Unterstützung von *Varianten*, d.h. zwei unterschiedliche Varianten von Objekten haben das gleiche Objekt als Ursprung.



   Es ist beispielsweise denkbar, daß eine Spezifikation ab einem bestimmten Zeitpunkt von zwei Beauftragten unabhängig voneinander verändert wird. Die Spezifikation der Version V_1 wird in Form der beiden Varianten $V_{1.1}$ und $V_{1.2}$ weiterverarbeitet.

3. Die ausschließliche Unterstützung von Varianten reicht ohne eine Möglichkeit, die Varianten eines Objektes wieder zu einer neuen Instanz zu kombinieren, oft nicht aus. Dieses Vorgehen der Versionierung wird als *Konfiguration* bezeichnet.



   Erst die Verwaltung von Konfigurationen unterstützt das sogenannte *Concurrent Engineering*, d.h. das Erstellen von Varianten, das unabhängige

Weiterverarbeiten jeder Variante (z.B. durch unterschiedliche Benutzer) und das Zusammenfügen dieser Varianten zu einer neuen Instanz.

Die Spezifikation der Version V_1 wird in Form der beiden Varianten $V_{1.1}$ und $V_{1.2}$ weiterverarbeitet. Die Variante $V_{1.1}$ hat dabei wiederum Versionen gebildet, die Versionen $V_{1.1.1}$ und $V_{1.1.2}$. Die Varianten $V_{1.1.2}$ und $V_{1.2}$ werden zur neuen Instanz V_2 zusammengefügt.

Hierbei ist zu beachten, daß der automatische Abgleich von Varianten und das Erstellen einer Instanz eine sehr komplexe Aufgabe darstellt[2].

2.2.7 Zustand und Wiederholung von Aktivitäten

Eine Aktivität kann bezüglich eines zu fertigenden Produkts einen von mehreren möglichen Zuständen einnehmen. Die Aktivität kann

- noch nicht begonnen sein,
- sich zur Zeit in Ausführung befinden, oder
- bereits erfolgreich ausgeführt, d.h. beendet sein.

Zusätzlich kann es möglich sein, die gleiche Aktivität mehrfach, aber z.B. mit anderen Parametern o.ä. durchzuführen. Das ist nur dann möglich, wenn die Aktivität die Eingabeobjekte nicht irreversibel verändert. Beispielsweise kann die Vermessung eines Maskenblanks wiederholt werden, wenn bei der ersten Ausführung das falsche Objektiv für das Vermessungssystem gewählt wurde.

2.2.8 Der Zustand des Produkts

Der Zustand des einzelnen Produkts muß modelliert werden, um ihn später in der Produktsicht darstellen zu können. In der Beispielsequenz von Abbildung 2.5 geschieht das z.B. durch Hervorheben der bereits ausgeführten Aktivitäten, sowie der existenten Objekte (Eingabe- und Ausgabeobjekte). Der Zustand des Produkts ist sofort ablesbar, die nächste auszuführende Aktivität ist die Herstellung einer Maske mit Hilfe des Elektronenstrahlschreibers (ESS).

Die Vielzahl der vorhandene Produktsichten (auf jedes Produkt existiert eine eigene Produktsicht) macht eine zusätzliche Sicht notwendig, die diese Sichten koordiniert, und zusätzliche Dienste auf der Grundlage der Vereinigung aller Produktsichten zur Verfügung stellt. Solch eine koordinierende Sicht wird mit der Produktionssicht eingeführt.

[2] Zur Verwaltung von Konfigurationen bieten manche Datenbanksystemen einen Check-Out/Check-In Mechanismus an. Eine Kopie des Objekts wird aus der Datenbank geholt, lokal als Variante geändert (oft in einer privaten Datenbank), und nach Abschluß der Änderungen wieder zurückgestellt. Bei diesem Vorgang ist sie evtl. mit anderen Varianten zu einer neuen Instanz abzugleichen.

2.3 Die Produktionssicht

Da in der Regel mehrere Produkte parallel gefertigt werden, kommt es zwangsläufig vor, daß in unterschiedlichen Fertigungslinien dieselben Ressourcen (Ausrüstungsgegenstände) verwenden werden sollen, obwohl aus technischen Gründen eine Mehrfachbenutzung nicht möglich ist (z.B. gleichzeitige Benutzung des selben Ofens mit unterschiedlicher Temperatur).

Ebenso kann es bei bestimmten Ressourcen (z.B. einem chemischen Bad) ökonomisch sinnvoll sein, die Abläufe der unterschiedlichen Fertigungslinien so zu koordinieren, daß sie von unterschiedlichen Produkten gleichzeitig oder sequentiell (unmittelbar im Anschluß) belegt werden.

Zur Lösung solcher Fragen ist eine Produktionssicht notwendig, die die Gesamtheit der Produktsichten im Hinblick auf eine optimale Verteilung der Ressourcen verwaltet. Es werden – durch den Zweck der Produktionssicht bedingt – nur die zur Ausführung einer Aktivität benötigten Komponenten, d.h. Eingangsobjekte, Ressourcen (Ausrüstungsgegenstände) und Materialien erfaßt. Ausgabeobjekte sind keine Voraussetzung für die Ausführung einer Aktivität[3] und werden nicht explizit erfaßt. Die Produktionssicht erfaßt demnach nur einen Ausschnitt der Produktsichten, da die Ausgabeobjekte, die in den Produktsichten enthalten sind, in der Produktionssicht nicht interessieren (siehe Abbildung 2.6 im Vergleich zu Abbildung 2.5). Außerdem werden in der Produktionssicht nicht alle Produktsichten berücksichtigt, sondern nur die, deren Produkt sich gerade in der Produktion befindet.

> Die Produktionssicht ist also eine Teilsicht auf die Obermenge aller Produktsichten für diejenigen Produkte, deren Produktion noch nicht abgeschlossen ist. Es werden auch die Produke erfaßt, deren Produktion bereits modelliert, aber noch nicht begonnen wurde.

Die Produktionssicht stellt folgende Voraussetzungen an die beteiligten Produktsichten:

- Die Produktsichten haben alle eine identische Version der Prozeßsicht als gemeinsame Grundlage, da sonst keine gemeinsame Basis vorhanden ist, auf die sich eine Produktionssicht aufbauen läßt.
- Gleiche Aktivitäten werden bei sämtlichen Produkten als identisch angenommen. Dies gilt auch dann, wenn die Aktivitäten mit unterschiedlichen Parameterwerten aufgerufen werden. Wenn beispielsweise eine Aktivität *CAD-Entwurf* für Produkt *A* von einem anderen Bearbeiter (unterschiedlicher Parameterwert) wie für Produkt *B* ausgeführt wird, so werden dennoch die Aktivitäten *CAD-Entwurf* für beide Produkte als äquivalent angenommen und in die Produktionssicht übernommen. Diese Vorausset-

[3] Dies ändert sich, wenn diese Ausgangsobjekte wieder Eingabeobjekte für nachfolgende Aktivitäten sind, aber dann werden sie als solche erfaßt.

zung wird getroffen, da der Unterschied eines Parameterwertes für die Aufgabe, die die Produktionssicht erfüllt, irrelevant ist.

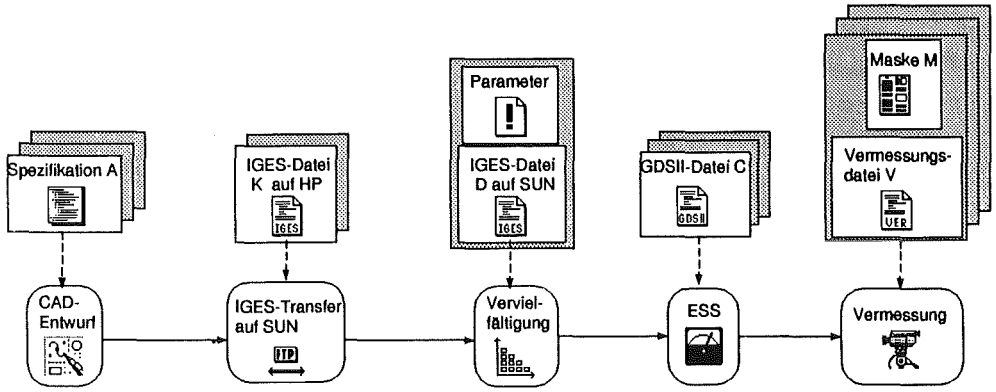


Abbildung 2.6: Produktionssicht einer Aktivitätenkette

Die Produktionssicht erfaßt folgende Eigenschaften der Produktsichten:

- Darstellung aller an einer Fertigung beteiligten Aktivitäten in unterschiedlichen Hierarchiestufen.
- Kausale Vorgänger-/Nachfolgerbeziehung dieser Aktivitäten.
- Alternativaktivitäten, wenn diese Aktivität an irgendeiner Produktion beteiligt ist.
- Nebenläufigkeiten
- Eingabeobjekte in Aktivitäten. Da für eine Aktivität (z.B. den *CAD-Entwurf*) mehrere Eingabeobjekte (d.h. verschiedene Spezifikationen) möglich werden, werden die entsprechenden Datentypen aus der Prozeßsicht in der Produktionssicht mehrere Ausprägungen besitzen.

2.4 Der Zusammenhang der Sichten

Da die Prozeßsicht alle potentiellen Fertigungswege enthält, bildet sie die Grundlage für die Produktsichten. Jede Produktsicht wird diejenigen Aktivitäten des Prozeßmodells enthalten, die sie für die Fertigung des von ihr repräsentierten Produkts benötigt. Zusätzlich wird die Produktsicht an Stelle der Typschablonen für Eingabe- und Ausgabeobjekte reale Objekte verwalten, sofern die entsprechenden Aktivitäten bereits ausgeführt wurden. Aus allen Produktsichten, deren repräsentiertes Produkt sich noch in der Fertigung befindet, wird die überlagerte Produktionssicht gebildet, die jedoch nur die für sie relevanten Aspekte berücksichtigt. Der Zusammenhang von Prozeß- Produkt- und Produktionssicht läßt sich demnach wie in Abbildung 2.7 gezeigt darstellen.

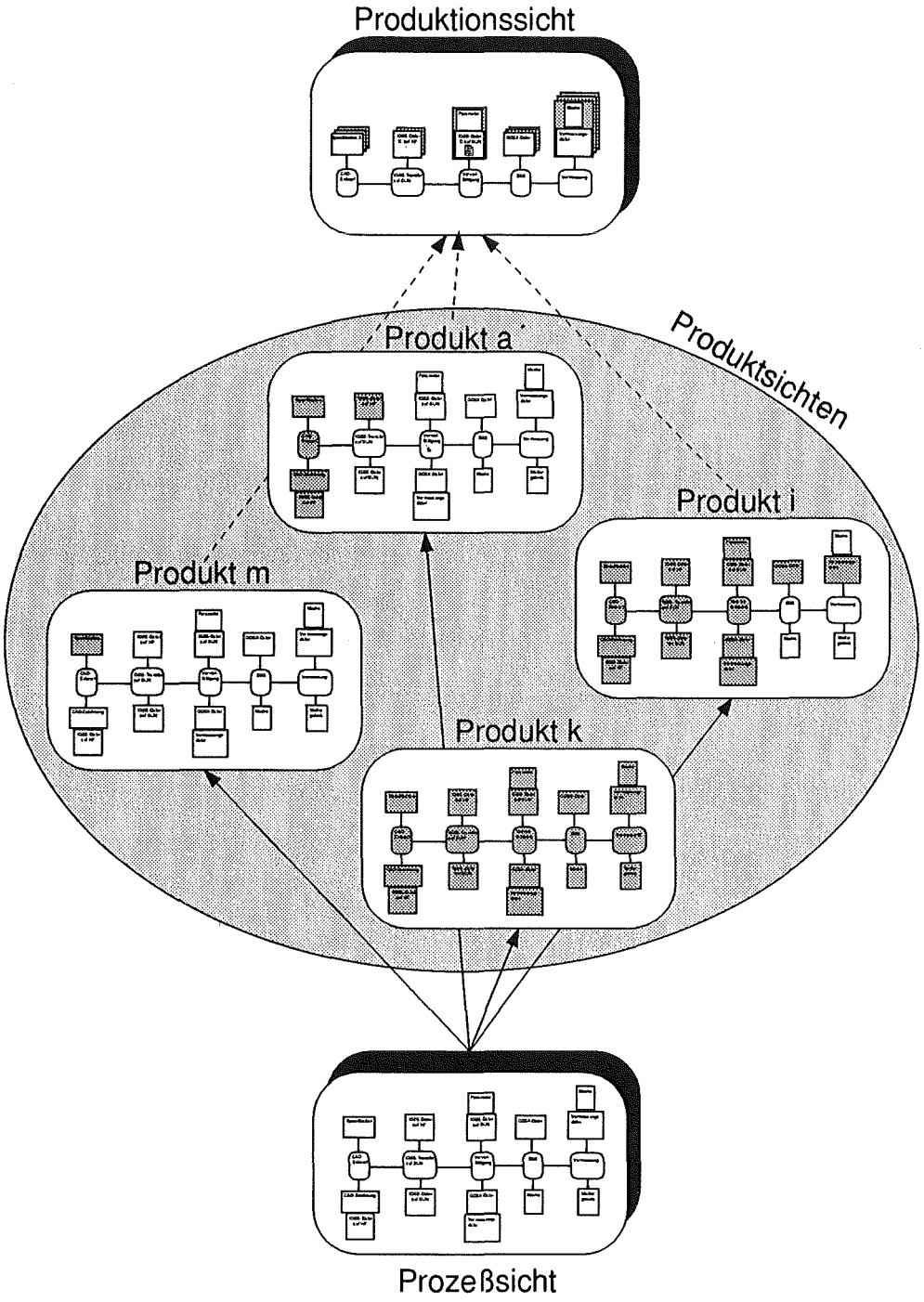


Abbildung 2.7: Zusammenwirken der unterschiedlichen Sichten

Alle Produktsichten sind auf der Grundlage der Prozeßsicht erstellt (durchgezogene Pfeile), jedoch gehen nur die Produktsichten der noch nicht fertiggestellten Produkte (Produkt m, a und i) in das Produktionsmodell ein (gestrichelte Pfeile). Das bereits fertiggestellte Produkt k (alle Aktivitäten hinterlegt und damit ausgeführt) wird in der Produktionssicht nicht berücksichtigt.

2.5 Gegenüberstellung von Merkmalen der einzelnen Sichten

In folgender Tabelle sind zusammenfassend die Merkmale der einzelnen Sichten Prozeßsicht, Produktsicht und Produktionssicht nochmals abschließend gegenübergestellt.

Merkmal	Prozeß-sicht	Produkt-sicht	Produktions-sicht
Hierarchie	+	+	+
kausale Vorgänger-/Nachfolgerbeziehung	+	+	+
Alternativaktivitäten	+	–[a]	+
alle möglichen Aktivitäten	+	–	+[b]
Nebenläufigkeit von Aktivitäten	+	+	+
Eingabeobjekte von Aktivitäten	+[c]	+	+
Ausgabeobjekte von Aktivitäten	+[c]	+	–
Versionen von Produktdaten	–	+	–
Ressourcen	+[d]	+	+
Materialien	+[d]	+	+
Synchronisation	–	+	+
Zustand und Wiederholung von Aktivitäten	–	+	–
Produktionsfortschritt	–	+	+

[a] Für bereits bearbeitete Aktivitäten werden keine Alternativaktivitäten erfaßt, für in der Zukunft noch zu bearbeitende Aktivitäten ist die Erfassung von Alternativaktivitäten jedoch denkbar.

[b] Es wird nur die Vereinigung der Aktivitäten erfaßt, die an der Produktion eines sich noch in der Fertigung befindlichen Produkts beteiligt sind.

[c] Erfaßt werden ausschließlich Datentypen (Objekttypen), keine realen Daten (Objekte).

[d] Hier werden die potentiell möglichen Ressourcen bzw. Materialien erfaßt.

Kapitel 3

Die Anbindung von Werkzeugen

Die im vorigen Kapitel vorgestellten Systemsichten von *PRAXIS* beschreiben die Grundlage für eine Benutzerinteraktion. Bei dieser Interaktion mit *PRAXIS* arbeitet der Benutzer mit einer Vielzahl von rechnergestützen Werkzeugen, sei es zum Durchführen von Aktivitäten (z.B. durch das Arbeiten mit einem CAD-System), oder zum Erfassen der anfallenden Produktinformation. Die Fragen, wie die den Benutzer unterstützenden Werkzeuge gestartet werden und woher die für die Werkzeuge notwendigen Daten kommen, bzw. wo die bei der Interaktion und den damit verknüpften Aktivitäten entstehenden Daten abgelegt werden, sollen im folgenden geklärt werden. Diese Fragen sollen im Idealfall weitgehend vom Anwender ferngehalten werden, so daß dieser sich auf das eigentliche Ziel, die Fertigstellung des Produkts konzentrieren kann. Die Betrachtung der Historie der Produktentwicklung zeigt Alternativen auf, wie Werkzeuge (in unterschiedlichen Integrationsstufen) zur Unterstützung der Produktentwicklung und Produktfertigung eingesetzt werden können.

3.1 Konventionelle Produktentwicklung

Betrachtet man die traditionellen Methoden zur Produktentwicklung, so ist diese von der Vorgehensweise geprägt, daß Projekte zur Entwicklung von Produkten in isolierte Komponenten unterteilt werden, welche anschließend getrennt voneinander unter Verwendung geeigneter Werkzeuge durchgeführt werden können. Danach können die Teilkomponenten in einer oder mehreren Stufen zum Gesamtprodukt zusammengefaßt werden, wie es in Abbildung 3.1 symbolisch dargestellt wird.

Die Produktivität kann durch die Verwendung von besseren Werkzeugen erhöht werden. Die Verwendung komplexerer Werkzeuge ermöglicht in der Regel die Bearbeitung komplexerer Teilkomponenten. Dennoch hat diese Vorgehensweise entscheidende Nachteile:

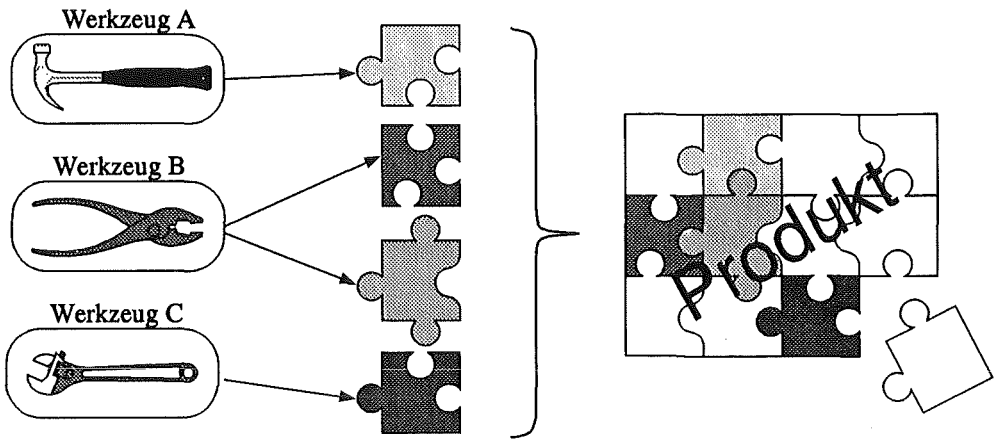


Abbildung 3.1: Konventionelle Produktentwicklung mit eigenständigen Werkzeugen

- Durch die Verwendung von unterschiedlichsten Werkzeugen gestaltet sich der Austausch von Daten zwischen diesen Werkzeugen als Problem. Der Aufwand, um Daten unterschiedlichen Formats ineinander zu überführen, ist beträchtlich. Außerdem ist ein u.U. erheblicher Konvertierungsaufwand erforderlich, wenn ein Werkzeug durch ein neues ersetzt wird. Die Schulung oder Einarbeitung der Anwender für die Benutzung dieser neuen Werkzeuge ist ebenfalls nicht zu vernachlässigen.

- Durch die Aufteilung der Produktentwicklung in isolierte Komponenten können Abhängigkeiten zwischen den Komponenten keine Berücksichtigung finden, was während der Integration oftmals zu erheblichen Problemen führt. Da die Integration erst relativ spät in der Produktentwicklung angesiedelt ist, werden die so auftauchenden Probleme zu spät sichtbar, was die Fertigstellung erheblich verzögert. Außerdem ist ein Redesign in dieser späten Projektphase der Integration beim Erkennen eines Designfehlers kaum mehr möglich.

Durch die Integration der verwendeten Werkzeuge sowie der Datenintegration kann ein Teil dieser Probleme beseitigt werden.

3.2 Integration der Werkzeuge

Durch die Integration von unterschiedlichen Werkzeugen sowie der Datenintegration (Abbildung 3.2) entsteht die Möglichkeit, die verwendeten Daten zwischen den Werkzeugen auszutauschen. Die aufwendige Mehrfacherfassung von Information ist hinfällig. Zusätzlich können die Schnittstellen für die Benutzer bei unterschiedlichen Werkzeugen konsistent realisiert werden.

Dennoch haben auch diese integrierten Lösungen ihre Schwachstellen:

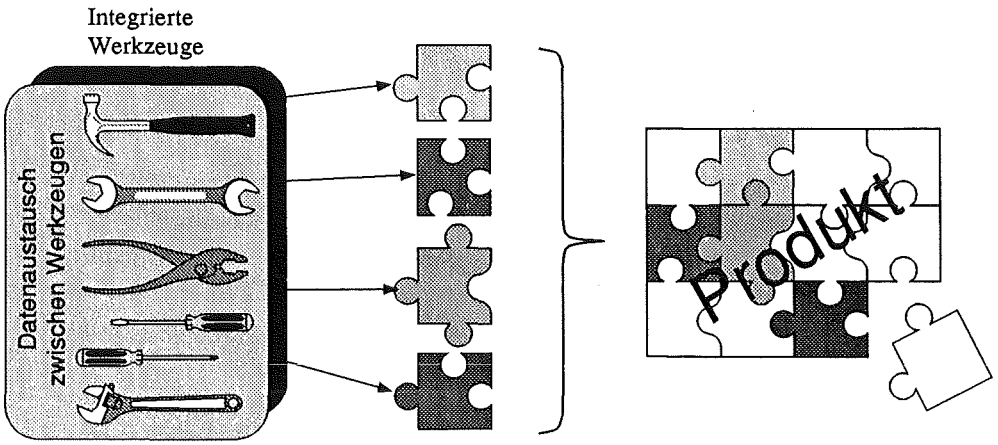


Abbildung 3.2: Produktentwicklung mit integrierten Werkzeugen

- Sie sind sehr starr in der Verwendung ihrer Werkzeuge, d.h. der Anwender ist auf die in diesen Paketen angebotenen Werkzeuge angewiesen. Er hat nicht die Möglichkeit Werkzeuge, die von der Werkzeugbox angeboten werden gegen Werkzeuge gleicher Funktionalität auszutauschen, z.B. um schon genutzte Werkzeuge auch weiterhin zu verwenden.

- Es ist nicht unbedingt zwingend, daß für Bearbeitung aller Teilaufgaben die jeweils besten verfügbaren Werkzeuge integriert werden. So sind oftmals firmenpolitische Gründe entscheidend für die Verwendung von Werkzeugen bei der anstehenden Integration.

Daraus leitet sich die Notwendigkeit ab, eine Art offenes und flexibles Baukastensystem zu verwenden, das es gestattet, Werkzeuge völlig frei in das System einzubinden und trotzdem einen Austausch von Daten zwischen den Werkzeugen auf einen bestimmten Niveau zu gestatten.

3.3 Das Framework

Ein Framework ist ein solches Baukastensystem. Es realisiert vom jeweiligen Anwendungsgebiet unabhängige Forderungen wie konsistente Datenhaltung, konsistente Benutzerschnittstelle, Kommunikationsmechanismen zwischen Werkzeugen, usw.

Ein entscheidender Vorteil eines Frameworks gegenüber einer herkömmlichen integrierten Lösung ist die Möglichkeit, durch die Integration von beliebigen Werkzeugen das Framework auf ein spezielles Anwendungsgebiet anzupassen. Diese Möglichkeit hat entscheidende Vorteile:

- Das Framework kann sukzessiv aufgebaut werden und auf diese Art mit dem Wissen über den Fertigungsprozeß mitwachsen. Dadurch läßt sich das

Problem eines immer komplexer werdenden Fertigungsprozesses besser handhaben.

- Durch die Möglichkeit, beliebige informationsverarbeitende Werkzeuge in das Framework zu integrieren, hat der Anwender entscheidende Vorteile:
  - Zum einen können Werkzeuge eingebunden werden, die schon längere Zeit als Insellösungen für die Unterstützung des Fertigungsprozesses verwendet werden. Diese Werkzeuge sind in ihrer Funktionsweise geläufig, d.h. eine Einarbeitung wird hinfällig, sie laufen stabil oder ihre Schwachstellen sind schon hinlänglich bekannt.
  - Zum anderen können firmenpolitisch unabhängig stets die besten zur Lösung einer Teilaufgabe verfügbaren Werkzeuge beschafft und eingebunden werden.
  - Falls kommerziell verfügbare Werkzeuge die Anforderungen des Anwenders nur unzureichend erfüllen, so kann dieser durch selbstentwickelte oder weiterentwickelte Werkzeuge und deren Integration in das Framework Abhilfe schaffen.

Ein Framework kapselt werkzeugspezifische Fragen durch die Integration der Werkzeuge vom Benutzer ab. Dazu zählen beispielsweise die Quelle der Eingabeobjekte und Ziel der Ausgabeobjekte, die Aufrufparameter der Werkzeuge und ähnliches.

Es ist zwar eine notwendige Voraussetzung für ein flexibles System, die Einbindung von beliebigen Werkzeugen zu ermöglichen, hinreichend für die Verbesserung der Produktivität ist diese Einbindungsmöglichkeit keineswegs. Entscheidend ist die Zuordnung der Werkzeuge zu den Aktivitäten. Durch die Definition der Aktivitäten, die zur Produktfertigung notwendig sind (prozeßspezifischer Teil der Produktsicht), und die nachfolgende Zuordnung der dabei verwendeten Werkzeuge wird eine gewisse Fertigungsmethodik definiert. Diese Methodik nimmt dem späteren Anwender die Entscheidung ab, welcher Fertigungsschritt für ein Produkt als nächstes durchgeführt werden muß, und welche informationstechnischen Werkzeuge zur Durchführung unterstützend angewendet werden sollen (siehe Abbildung 3.3). Die miteinander vernetzten Aktivitäten sind demnach eine Art Vorgehensmodell für die Fertigung eines Produkts.

Dieses Verfahren wird klarer, wenn man sich die unterschiedlichen Personenkreise betrachtet, die mit einem Framework arbeiten.

3.3.1 Anwendergruppen eines Frameworks

Man kann folgende Personengruppen unterscheiden, die mit einem Framework arbeiten:

- Es gibt eine Gruppe von Anwendern (z.B. Projektmanager), die den Projektablauf festlegen, d.h.

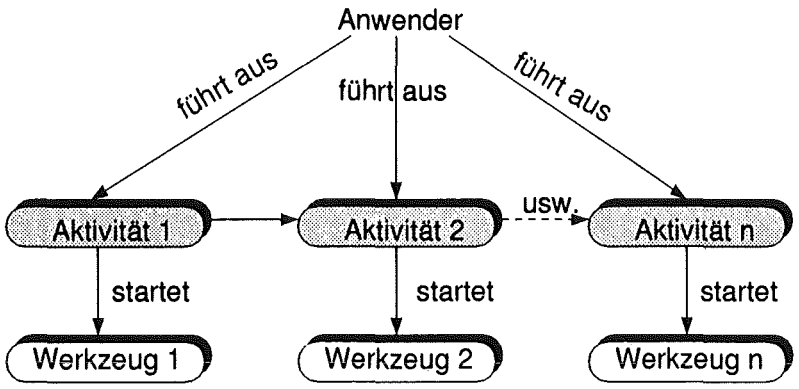


Abbildung 3.3: Zuordnung von Werkzeugen zu Aktivitäten

- die ein Projekt (z.B. die Fertigung eines Produkts) in ausführbare Aktivitäten unterteilen,
- jeder Aktivität Werkzeuge zuordnen, mit deren Hilfe die Aktivitäten durchgeführt werden können,
- den Aktivitäten die Gruppe von Personen zuordnet, die die Aktivitäten ausführen dürfen,

und die korrekte und planmäßige Ausführung des Projektablaufs überwacht.

- Es gibt die Gruppe von Anwendern, die die Aktivitäten ausführt und zwar gemäß den oben festgelegten Richtlinien. Sie werden auch den jeweiligen Werkzeugen die für jedes Produkt spezifischen Daten zuordnen, mit denen die Werkzeuge für genau jedes Produkt arbeiten.

- Und es gibt eine Gruppe für administrative Aufgaben innerhalb den Frameworks. Diese ist zuständig für die Installation, Konfiguration und Integration von neuen Werkzeugen. Diese Gruppe stellt überhaupt erst sicher, daß die ersten beiden Anwendergruppen problemlos mit dem System arbeiten können.

3.3.2 Das Problem der Datenintegration

Nachdem die Integration von Werkzeugen möglich ist, stellt sich die Frage der Integration der Daten, auf denen diese Werkzeuge arbeiten. Dazu gibt es verschiedene Lösungsansätze auf verschiedenen Integrationsebenen.

Black-Box-Integration

Die einfachste Möglichkeit, Daten zwischen Werkzeugen auszutauschen, ist der Weg über Dateien. Alle computergestützten Werkzeuge lesen zu Beginn eine

Eingabedatei, erhalten daher ihre Daten, arbeiten auf diesen Daten und verändern sie gegebenenfalls, um anschließend eine Ausgabedatei zu erzeugen. Verwaltet man nun diese Dateien, z.B. in einem gemeinsamen Dateisystem oder – etwas komfortabler in einem objektorientierten Datenbanksystem – und tauscht die Dateien zwischen den unterschiedlichen Werkzeugen aus, so hat man eine Integration der Daten auf primitivster Ebene. Man spricht in diesem Zusammenhang auch von Black-Box-Integration, die in Abbildung 3.4 dargestellt ist.

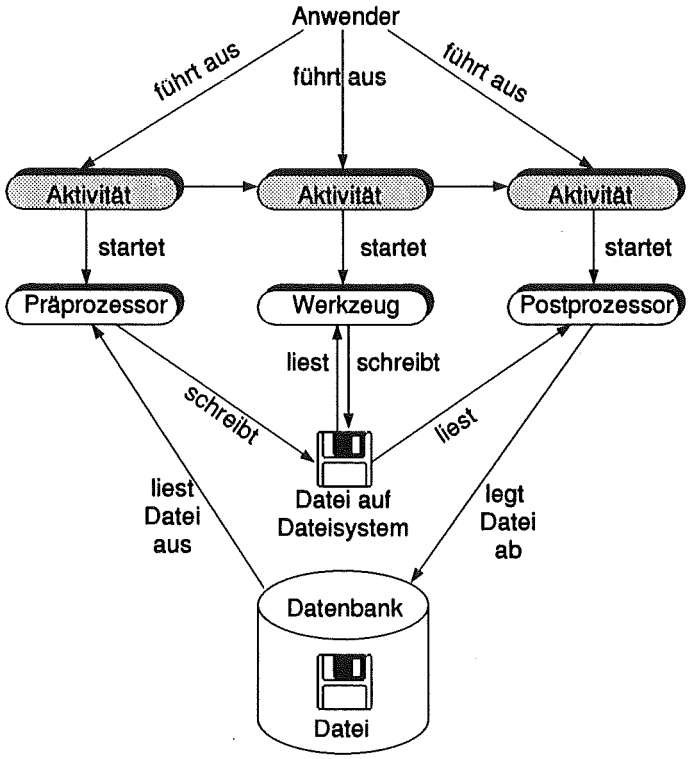


Abbildung 3.4: Black-Box-Integration

Das Werkzeug, das automatisch gestartet wird, wenn der Anwender eine Aktivität ausführt, liest bzw. schreibt eine Datei vom bzw. auf das Dateisystem (in Abbildung 3.4 ist das die mittlere Aktivität). Wird diese Datei, wie in Abbildung 3.4 dargestellt, in einer Datenbank gespeichert, so muß *vor* Ausführung der Aktivität die Datei von der Datenbank auf das Dateisystem transferiert werden. Dies geschieht, indem *vor* Ausführung der Aktivität eine andere Aktivität ausgeführt, die einen Präprozessor startet, der die Übertragung der Datei von der Datenbank auf das Dateisystem übernimmt (siehe Abbildung 3.4, linke Aktivität). Analog dazu muß *nach* Beendigung der Aktivität eine weitere Aktivität ausgeführt werden, die einen Postprozessor startet, der die Ergebnisdatei vom Dateisystem wieder in die Datenbank schreibt (siehe Abbildung 3.4, rechte Aktivität). Prä- und Postprozessoren werden also nur infolge unterschiedlicher Schnittstellen von Dateisystem und Datenbanksystem notwendig.

Der Hauptvorteil dieser Integrationsalternative ist, daß sie immer anwendbar ist, da alle informationsverarbeitenden Werkzeuge dateiorientiert arbeiten. Was

man tun muß, ist die Bereitstellung der Eingabedatei im richtigen Format beim Werkzeugstart, sowie die Speicherung der Ausgabedatei nach Beendigung des Werkzeugs.

Jedoch hat diese Vorgehensweise auch Nachteile: wenn unterschiedliche Werkzeuge verschiedene Dateiformate verwenden, dann müssen die Dateien zwischen diesen Formaten konvertiert werden. Da theoretisch[1] jedes Format in jedes andere überführt werden können muß, sind im ungünstigsten Falle für n unterschiedliche Dateiformate $n * (n - 1)$ Konverter zu implementieren, was mit steigender Anzahl unterstützter Dateiformate recht aufwendig wird. Zudem ist der Weg über eine Zwischendatei ineffizient und es liegt streng genommen keine Datenintegration vor.

White-Box-Integration über Zwischendatei

Will man die Daten auf einer höheren Ebene als der Dateiebene integrieren, muß man für sie einheitliche Datenstrukturen z.B. in Form eines Datenbankschemas in einer Datenbank zur Verfügung stellen.

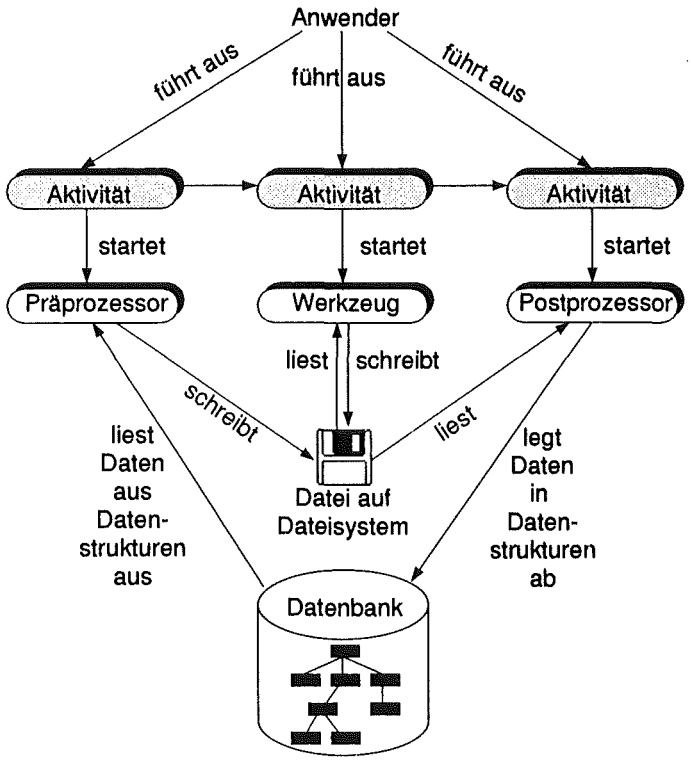


Abbildung 3.5: White-Box-Integration über Zwischendatei

Anschließend besteht für alle Werkzeuge die theoretische Möglichkeit, sich die benötigten Daten aus dieser Datenbank zu extrahieren sowie die erzeugten Da-

[1] Natürlich müssen nur Konverter zwischen Dateiformaten bereitgestellt werden, zwischen denen Information ausgetauscht werden soll.

ten an den entsprechenden Stellen wieder abzulegen. Problematisch dabei ist, daß dazu die Werkzeuge die entsprechenden Stellen der Schemata kennen müssen. Weil die Schemata aber individuell erzeugt werden, müssen theoretisch die Werkzeuge individuelle Datenquellen und Datensenken zulassen. Da diese Forderung aber von den Werkzeugen in der Regel nicht erfüllbar ist, bleibt auch hier wieder der Ausweg über eine Zwischendatei. Sie wird aber – im Gegensatz zur Black-Box-Integration – nicht in dieser Form auch in der Datenbank gespeichert, sondern aus den Datenstrukturen mittels eines Präprozessors vor dem Starten des Werkzeugs erzeugt. Analog überträgt ein Postprozessor nach der Beendigung des Werkzeugs die Information der Ausgabedatei in die entsprechenden Datenstrukturen der Datenbank. Dieses Vorgehen zeigt Abbildung 3.5.

Die Vorteile sind offensichtlich: auch dieses Verfahren ist für alle kommerziellen Werkzeuge, die dateiorientiert arbeiten, anwendbar. Da in diesem Fall mit den Datenstrukturen gewissermaßen ein neutrales Datenformat zur Verfügung gestellt wird, verringert sich die Anzahl der benötigten Konverter (Präbzw. Postprozessoren) entsprechend. Will man n unterschiedliche Dateiformate unterstützen, so sind n Präprozessoren und n Postprozessoren zu implementieren. Insgesamt bedeuten $2n$ Konverter einen im Gegensatz zur Black-Box-Integration ($n * (n-1)$ Konverter) erheblich geringeren Aufwand.

Was bleibt ist der nachteilige Weg über die Zwischendatei sowie der Aufwand der Datenkonvertierung. Diese Nachteile treten bei Anwendung der White-Box-Integration ohne Zwischendatei nicht mehr auf.

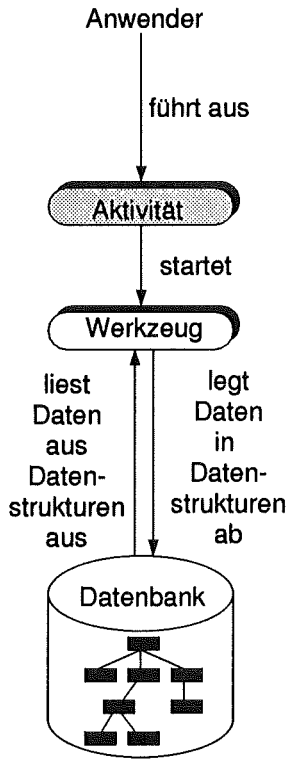


Abbildung 3.6: White-Box-Integration

White-Box-Integration

Eine Datenintegration auf höchster Stufe ist erst dann erreicht, wenn die Werkzeuge direkt auf die Daten d.h. über die Datenstrukturen der Datenbank zugreifen können, wie in Abbildung 3.6 dargestellt. In diesem Falle entfällt die Konvertierung zwischen Dateien unterschiedlichen Formates und den Daten aus definierten Datenstrukturen.

Entscheidender Nachteil dieser Integrationsalternative ist die Tatsache, daß sie ausschließlich für Werkzeuge anwendbar ist, deren Datenquellen und Datensenken individuell anpaßbar sind, was in der Regel nur auf selbstimplementierte Werkzeuge zutrifft.

Kapitel 4

Die Datenbank als Integrationsplattform

Bisher wurden zwei Informationskomponenten erarbeitet, die in *PRAXIS* verwaltet werden müssen: die Informationen der unterschiedlichen Sichten (die Prozeß-, die Produkt- und die Produktionssicht) sowie die zugehörigen Daten (z.B. Produktdaten). Für die Speicherung und Bereitstellung dieser Information sollte das gesamte System auf der Grundlage eines Datenbanksystems aufgebaut werden, da ein Datenbanksystem die dazu benötigte Funktionalitäten zur Verfügung stellt:

- Die Definition der Datenstrukturen auf der Grundlage eines festgelegten Datenmodells. Als Datenmodelle existieren das Netzwerkmodell [COD71], das hierarchische [Dat81], das relationale [Cod70] sowie verschiedene, leicht unterschiedliche objektorientierte Modelle [ABD 89].

- Die Bereitstellung einer Sprache, die die Definition und Manipulation der Datenstrukturen ermöglicht, eine sogenannte Data-Definition-Language (DDL).

- Die Bereitstellung einer Sprache, die die Manipulation der in den Datenstrukturen gespeicherten Daten erlaubt, eine sogenannte Data-Manipulation-Language (DML).

- Die Definition von Beziehungen (Referenzen) zwischen Daten (auf logischer Ebene, z.B. über Wertgleichheit unterschiedlicher Felder, oder auf physischer Ebene, z.B. durch Verzeigerung).

- Die Garantie der Datenpersistenz, d.h. daß gespeicherte Daten nicht verlorengehen können.

- Die Bereitstellung von Recovery-Mechanismen, d.h. Mechanismen, die unterschiedlichen Störfällen wie Systemausfälle, Schäden der Speichermedien usw. mit unterschiedlichen Vorsorgemaßnahmen begegnen.

- Die Bereitstellung eines Transaktionskonzeptes, das es erlaubt, Operationen auf den Daten so zusammenzufassen, daß immer die Einhaltung folgender 4 Grundregeln ([LS87]) garantiert wird:

  - Ununterbrechbarkeit, d.h. entweder werden alle Operationen der Transaktion ausgeführt und ihre Wirkungen sichtbar oder keine.
  - Konsistenzerhaltung, d.h. eine Transaktion überführt die Datenbank von einem konsistenten Zustand in einen anderen, ebenfalls konsistenten Zustand.
  - Dauerhaftigkeit der Ergebnisse.
  - Isolierter Ablauf, d.h. parallel ablaufenden Transaktionen nehmen keine (möglicherweise inkonsistente) Zwischenergebnisse einer anderen Transaktion wahr.

  Die Verwaltung von nebenläufigen Transaktionen auf der Datenbank geschieht durch die Bereitstellung von geeigneten Sperrmechanismen. Sie verhindern z.B., daß eine Transaktion die Ergebnisse einer anderen, parallel ablaufenden Transaktion überschreibt.

- Die Definition von Integritätsbedingungen auf den Daten und die Überwachung ihrer Einhaltung bzw. die Sicherstellung durch die Definition von Automatismen (z.B. Trigger).

- Die Bereitstellung von Mechanismen zur Autorisierung von Benutzern, d.h. der Gewährung von selektiven Zugriffsrechten auf die Daten.

Die Hauptunterscheidung der Datenbanksysteme wird anhand des Datenmodells getroffen, auf dem die Systeme basieren. Sie werden gemäß den Datenmodellen in Netzwerk-, hierarchische, relationale oder objektorientierte Datenbanksysteme unterteilt.

4.1 Relationale Datenbanksysteme

Seit ihrer Einführung Anfang der 80er Jahre haben relationale Datenbanksysteme die sich zuvor im Einsatz befindlichen hierarchischen und Netzwerkdatenbanksysteme immer mehr in den Hintergrund gedrängt. Ein Grund dafür war die Einfachheit des Datenmodells.

Ein Datenobjekt wird durch Eigenschaften definiert. Man stelle sich z.B. eine Chrommaske als Datenobjekt vor. Dieses Datenobjekt Chrommaske besitzt u.a. die Eigenschaften

- Maskennummer
- Name des Auftraggebers
- Herstellungsdatum
- Fläche (in mm^2)
- Dicke (in μm)

Eigenschaften dürfen im Relationenmodell nur atomare Werte besitzen, z.B. eine Zeichenkette für den Name des Auftraggebers oder einen Zahlenwert für die Dicke der Chrommaske. Auf diese Art erhält man eine flache Struktur, die sich leicht in einer oder mehreren Tabellen speichern läßt (siehe Abbildung 4.1). Eine solche Tabelle (eine sogenannte Relation) enthält dann eine Menge von Chrommaskenobjekten. Eine Chrommaske wird durch die Werte einer Zeile (ein sogenanntes Tupel) beschrieben. In einer Zeile ist für jede Eigenschafte (einem sogenannten Attribut) einer bestimmten Chrommaske ein Wert abgelegt (ein sogenannter Attributwert).

Dieses Struktur, eine Relation als Menge von Tupeln von Attributen, ist die einzige von relationalen Datenbanksystemen unterstützte Datenstruktur.

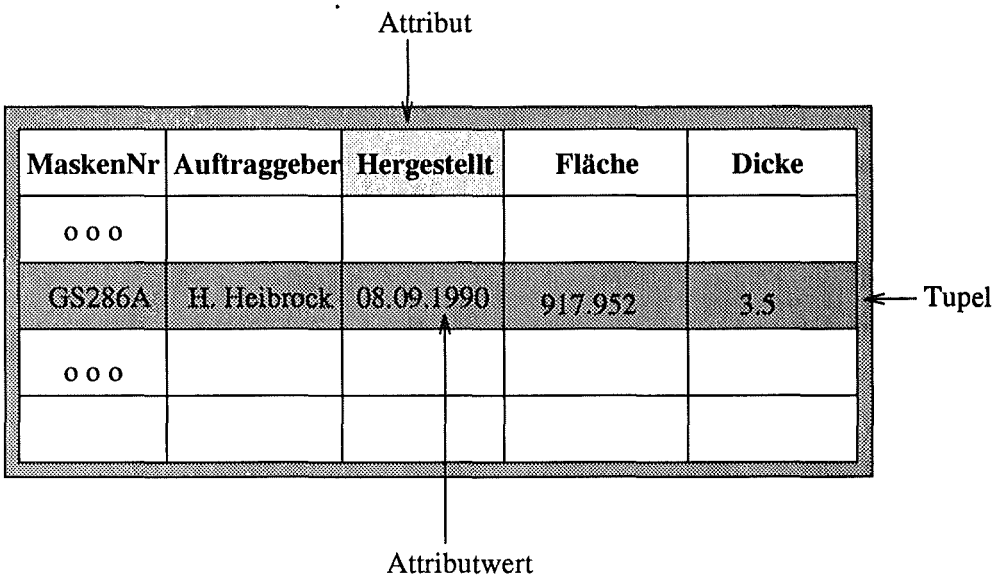


Abbildung 4.1: Eine Beispielrelation *Chrommaske*

Da sich mit Hilfe der Relationenalgebra Anfragen an das System über Daten geeignet optimieren lassen, sind die Systeme in der Lage, an sie gestellte Anfragen teilweise selbst zu optimieren. Durch die Bereitstellung einer deskriptiven Anfragesprache (z.B. SQL [MA]), bei der man die Daten beschreibt, die man als Ergebnis wünscht, nicht aber den Weg, wie man zu diesen Daten kommt, wurde die Bedienung der Datenbanksysteme zusätzlich erleichtert.

Allerdings haben sich bei relationalen Datenbanksystemen im Laufe ihres Einsatzes auch einige Schwachstellen gezeigt:

- Die Identifizierung eines Datensatzes geschieht durch dessen Schlüssel, der in der Regel als eine Kombination aus den Werten verschiedener Attribute (der sogenannten Schlüsselattribute) gebildet wird. Die Identität eines Datensatzes ist demnach wertbasiert, was zur Folge hat, daß sich die Identität eines Datensatzes ändert, wenn man die Werte der Schlüsselattribute ändert. Diese Verhaltensweise möchte man i.d.R. ausschließen.

- Die Einfachheit des Datenmodells hat sich als Nachteil bei der Modellierung komplexer, verschachtelter und vernetzter Datenstrukturen erwiesen.

- Während die Relationenalgebra mengenbasiert arbeitet, sind die Programmiersprachen, die die Daten anschließend weiterverarbeiten satzbasiert aufgebaut. Dadurch entstehen an der Schnittstelle von Datenbanksystem und Programmiersprache Probleme beim Datenaustausch[1].

- Zur Vermeidung von redundanter Datenhaltung wendet man beim Datenbankentwurf sogenannte Normalisierungsregeln [Ken83] an. Diese haben zur Folge, daß logisch zusammengehörenden Daten oftmals über viele verschiedene Relationen verteilt werden, aus welchen sie beim Zugriff auch wieder zusammengesetzt werden müssen. Dazu sind in der Verarbeitung sehr aufwendige Verbundoperationen (Joins) notwendig, die bei den Systemen Effizienzprobleme verursachen können.

- Das Fehlen der Möglichkeit an das System rekursiven Anfragen zu stellen, zwingt oft zu umständlicher Anfrageformulierung oder macht bestimmte Anfragen nicht möglich.

- Das Wissen über die Operationen auf den Daten wird verteilt über alle Anwendungsprogramme, die auf den Daten operieren. Damit sind diese Operationen, z.B. als Fehlerquelle im Fehlerfall, schwer überschaubar.

Diese Nachteile führten in Bereich der Datenbankforschung dazu, alternative Datenmodelle zu unterstützen (wie z.B. das objektorientierte Datenmodell), oder existierende relationale Systeme um fehlenden Funktionalitäten zu erweitern ([Cat91]).

4.2 Objektorientierte Datenbanksysteme

Objektorientierte Datenbanksysteme bieten Lösungen für einige der entscheidenden Nachteile von relationalen Datenbanksystemen an. Obwohl erst Mitte der 80er Jahre mit der Forschung auf dem Gebiet der objektorientierten Datenbanken begonnen wurde, sind heute – weniger als 10 Jahre später – bereits mehr als 20 kommerzielle Systeme auf dem Markt erhältlich, die sich als solche bezeichnen. Zur Zeit laufen Anstrengungen, die Welt der objekt-orientierten Datenbanksysteme, zumindest in Teilbereichen, zu standardisieren, beispielsweise für Anfragesprachen.

Die Object Database Management Group (ODMG) ist ein Konsortium von ODBMS-Herstellern, das die Standardisierung von Definition und Zugriff auf Objektdatenbanken mit dem Ziel der Portabilität von ODBMS-Anwendungen

[1] Das Aufeinandertreffen von mengenbasierter und satzbasierter Verarbeitung wird als Impedance Mismatch [LS87] bezeichnet

anstrebt. Der Standard ODMG-93 der Object Database Management Group wird in [Cat93] beschrieben.

Durch Unterstützung einer gemeinsamen Schnittstelle erwarten die ODMG-Mitglieder eine signifikante Beschleunigung ihrer Bemühungen um eine Aufnahme von ODMG-93 in die Standards verschiedener anderer Organisationen (OMG, ANSI, ISO, STEP, PCTE und CFI). Alle führenden Hersteller von ODBMS haben sich verpflichtet, ihre Produkte ODMG konform auszurichten.

Die entscheidenden Punkte dieses Standards sollen hier wiedergegeben werden. Für eine vollständige Beschreibung siehe [Cat93].

Der Standard ODMG-93 läßt sich in vier Teile gliedern:

1. Objektmodell.
2. Sprache zur Definition von Objekten (Object Definition Language, ODL).
3. Deklarative Anfragesprache für Objekte (Object Query Language, OQL).
4. Sprachanbindung zur Objektmanipulation (Object Manipulation Language, OML).

Diese Teile werden im folgenden kurz beschrieben.

4.2.1 Objektmodell

Ein **Objekt** modelliert einen Gegenstand der realen Welt. Ein Objekt hat ganz allgemein bestimmte Eigenschaften und ein Verhalten.

Unter **Typen** versteht man die Zusammenfassung von Objekten mit gemeinsamen Eigenschaften und einem gemeinsamen Verhalten. Typen haben drei wesentliche Eigenschaften:

1. Sie sind in Ober- und Untertypen strukturierbar, wodurch die Vererbung von Eigenschaften und Verhalten ermöglicht wird.
2. Sie kennen auf irgendeine Weise alle zu ihr gehörenden Instanzen.
3. Sie haben eine oder mehrere Implementierungen für ihre Eigenschaften und ihr Verhalten.

Die Kombination aus einer Implementierung und einer Schnittstellendefinition eines Typs wird als **Klasse** bezeichnet.

Ein Objekt ist damit eine **Instanz** *genau* einer Klasse.

Das Verhalten eines Objekts wird durch die Menge der Operationen bestimmt, die auf dem Objekt ausgeführt werden kann. Eine solche Operation wird **Methode** genannt.

Die Eigenschaften eines Objekts können entweder durch **Attribute** oder durch **Beziehungen** zwischen Objekten beschrieben werden.

Jedes Objekt hat einen **Zustand**, d.h. eine Menge von Werten für die Attribute. Dieser Zustand ist gekapselt, d.h. nur über die **Methoden** zugreif- und veränderbar.

Im folgenden soll nur auf die wichtigsten Eigenschaften des Objektmodells eingegangen werden.

Objekt-Identität (unveränderbar)

Eine Besonderheit des objektorientierten Datenmodells ist die Möglichkeit, Objekte anhand ihrer Identität zu referenzieren. Das verlangt nach einer eindeutigen Objekt-Identität, die über die gesamten Lebensdauer des Objekts unverändert bleibt (manche Ansätze gehen davon aus, daß dies auch nach dem Entfernen eines Objekts aus der Datenbasis gültig bleibt, d.h. ein einmal verwendeter Identifikator wird auch nach dem Löschen des Objekts nicht neu vergeben).

Literale und komplexe Objekte

Literale sind die atomaren Bausteine, aus denen die Eigenschaften der Objekte aufgebaut sind. Die gängigen Literale sind

- Integer
- Character
- Boolean
- Float

Daneben existieren einige zusammengesetzte Literale wie Datum oder Zeit.

Das Konzept der objektorientierten Datenbanksysteme erlaubt es, komplexe Objekte durch die Anwendung von Konstruktoren wie

- Set: Menge von homogenen Objekten (ungeordnet, unique)
- Bag: Vielfachmenge von homogenen Objekten (ungeordnet, Duplikate)
- List: Sammlung von homogenen Objekten (geordnet)
- Array: Sammlung von homogenen Objekten (geordnet, indizierbar)

aus einfacheren Objekten zu erzeugen.

Beispiel:

```
Dokument: {
            Titel       : STRING;
            Erscheinung : DATE;
            Schluessel  : SET[STRING];
            Kapitel     : LIST[Chapter];
          }
```

Die Konstruktoren müssen orthogonal sein, d.h. jeder Konstruktor muß auf jedes Objekt anwendbar sein.

Es muß möglich sein, über Elemente der so konstruierten Objekte iterativ zugreifen zu können.

Operatoren, die auf komplexen Objekten arbeiten, müssen durch das System auf die einzelnen Komponenten weitergegeben werden (z.B. müssen beim Kopieren (deep copy) eines komplexen Objekts alle Komponenten kopiert werden.).

Beziehungen als Attribute

Eine Besonderheit im Attributbereich stellen die abgeleiteten Attribute dar, bei denen der Bildbereich durch eine Funktionsvorschrift berechnet wird. Diese wird anstelle des Datentyps bei der Definition des Attributs angegeben.

Beispiel:

```
Person: {   ...
            Geburtstag : DATE;
            Alter      : years( Aktuelles_Datum - Geburtstag );
        }
```

Beziehungen zwischen Attributen werde oft über inverse Attribute modelliert, falls die Information der dahinterstehenden Beziehung für beide Attribute identisch ist.

Beispiel: Ein Kapitel gehört immer zu genau einem Dokument, also ist die Beziehung *enthält* von Dokumenten und Kapiteln über die inversen Attribute *Kapitel* und *Doku* modellierbar.

```
Dokument: {
            Titel       : STRING;
            Erscheinung : DATE;
            Schluessel  : SET[STRING];
            Kapitel     : LIST[Chapter] <-> Doku;
          }

Chapter:  {
            Titel       : STRING;
            Nummer      : INTEGER;
            Doku        : Dokument <-> Kapitel;
```

}

Inverse Attribute werden vom System automatisch konsistent gehalten. Im obigen Beispiel wird also, sobald ein neues Kapitel definiert ist, und dafür der Name des Dokuments in das Attribut *Doku* eingetragen ist, dieses auch in die Kapitelliste des Objekts *Dokument* eingefügt.

Typ- oder Klassenhierarchien (Vererbung)

Durch die Möglichkeit, Typen oder Klassen hierarchisch aufzubauen, kann das Konzept der Vererbung verwirklicht werden, das die Wiederverwendbarkeit von Programmen durch die Tatsache erhöht, daß jedes Programm auf einer Ebene angreift, auf der es die maximale Anzahl von Objekten bearbeiten kann.

Besondere Transaktionen

Beim Transaktionskonzept für herkömmliche Datenbanksysteme geht man von der Annahme aus, daß Transaktionen sehr kurz sind, wie beispielsweise das Abbuchen eines Betrags von einem Konto und das Zubuchen dieses Betrags auf ein anderes. Dieses Konzept ist insbesondere für Objekten, die einem Entwurfsprozeß unterliegen (CAD-Zeichnungen, Dokumente, o.ä.), nicht mehr haltbar. Daher werden von manchen Systemen auch längere Transaktionen (Tage oder Wochen) unterstützt, während deren ein Objekt verändert werden kann.

Zudem ist eine strenge Sequentialisierung von Transaktionen nicht mehr haltbar, weshalb manche Systeme auch geschachtelte Transaktionen unterstützen.

4.2.2 Objekt Definition Language (ODL)

Die Anforderungen an die Sprache zur Definition der Objekte sind u.a.:

- Die Sprache sollte vollständig sein, d.h. alle semantischen Konstrukte des Objektmodells sollen beschreibbar sein.
- ODL sollte unabhängig von Programmiersprachen sein (ähnlich SQL [MA]).
- Für Optimierungen und spätere Anforderungen sollte ODL leicht erweiterbar sein.

Eine ausführliche Sprachbeschreibung von ODL in Backus-Naur-Form [ASU86] enthält [Cat93], Kapitel 3. ODL sollte auch in der Lage sein, Schemas von anderen Anwendungen wie STEP [N.N93a] (siehe dazu Kapitel 4.4.2) zu integrieren.

4.2.3 Objekt Query Language (OQL)

Die Sprache zum Extrahieren von Objekten aus der Datenbasis soll folgenden Anforderungen genügen:

- OQL soll einen deklarativen Zugang zu den Objekten ermöglichen, d.h. daß die gewünschten Objekte beschrieben werden und nicht der navigierende Zugriffsweg.
- Eine Zugriffsoptimierung, die durch die deklarative Natur von OQL notwendig ist, sollte leicht realisierbar sein.
- OQL sollte eine an SQL angelehnte Syntax besitzen. Andererseits sollte auch eine Syntax für die Anbindung an verschiedene Programmiersprachen (z.B. C++ und Smalltalk) zur Verfügung stehen.
- OQL hat keine expliziten Befehle zur Veränderung von Objekten.

Es gibt prinzipiell zwei Wege, wie man eine Sprache wie OQL nutzen kann: stand-alone oder eingebettet in eine Programmiersprache. Im ersten Fall muß OQL die Möglichkeit bieten, Objekte anhand ihres Namens bzw. der darauf zugänglichen Referenzen zu extrahieren. Im zweiten Fall handelt es sich bei OQL oft um eine funktionale Spracherweiterung der betreffenden Programmiersprache.

Eine ausführliche Sprachbeschreibung von OQL in Backus-Naur-Form [ASU86] enthält [Cat93], Kapitel 4.

4.2.4 Objekt Manipulation Language (OML)

Die Sprache zur Manipulation von Objekten soll u.a. folgenden Forderungen genügen:

- Typen für Datenbank- und Programmiersprache werden gleich behandelt, d.h. Instanzen von Klassen können persistent (in der Datenbasis) oder transistent (nur im Programm der Programmiersprache) sein.
- OML hat die selben syntaktischen und semantischen Regeln wie die Programmiersprache, in die sie eingebettet wird.

Daraus ergibt sich, daß die Sprache OML spezifisch ist für die Sprachanbindung. Es gibt demnach für unterschiedliche Programmiersprachen (z.B. C++ und Smalltalk) verschiedene OML. OML für C++ und Smalltalk sind in [Cat93] ausführlich beschrieben.

4.2.5 Übersichten kommerzieller objektorientierter Datenbanksysteme

In jüngster Zeit sind einige Vergleiche objektorientierter Datenbanksysteme erstellt worden. Erwähnenswert davon sind:

- Die Firma Barry & Associates hat die wohl umfassendste Sammlung an Eigenschaften unterschiedlicher Systeme (Gemstone, IDB Object Database, ITASCA, MATISSE, O2DB, Objectivity/DB, ObjectStore, ODBMS, ONTOS DB, OpenODB, POET, VERSANT, sowie Kala Persistent Data Server) zusammengetragen und veröffentlicht[Bar93].
- Das Forschungszentrum Informatik an der Universität Karlsruhe hat in einer Publikation [AASW94] neben einer Einführung in allgemeine Konzepte auch die gewählten Lösungen von fünf verschiedenen Systemen (Objectivity/DB, ObjectStore, ONTOS DB, OBST und VERSANT) miteinander verglichen.

Darüber hinaus enthält [Heu92] (Kapitel 10) eine knappe Übersicht kommerzieller Systeme.

4.3 Entwicklungszyklus von Informationssystemen

Die Ziele, die man beim Datenbankentwurf verfolgt, sind abhängig von den vom Datenbanksystem unterstützten Konzepten. War es z.B. bei relationalen Systemen ein Hauptziel, Anomalien durch Normalisierung [Ken83] zu vermeiden, so stehen bei objektorientierten Datenbanken Entwurfsziele im Vordergrund, die auch bei beliebiger anderer objektorientierter Software bestehen[2]. So ist beispielsweise die Normalisierung bei objektorientiertem Datenbankentwurf durch die Mächtigkeit des Datenmodells nicht nötig. Daher unterscheidet sich auch die klassische Vorgehensweise beim Datenbankentwurf aus den Zeiten der hierarchischen, Netzwerk- und relationalen Datenbanken recht stark von der bei objektorientierten Datenbanken verfolgten Vorgehensweise:

- Der klassischer Datenbankentwurf mit der dazugehörigen Anbindung an Anwendungsprogramme läßt sich in die in Abbildung 4.2 dargestellten Schritte unterteilen:

  1. Anforderungsanalyse, in der die Anforderungen an das Informationssystem (im vorliegenden Fall in bezug auf den LIGA-Fertigungsprozeß) analysiert werden. Die einzelnen Prozeßschritte werden untersucht, um die dafür relevanten Eigenschaften zu extrahieren.

[2] Z.B. Wiederverwendbarkeit, Verwendung von Vererbung zur Kompatibilität mit anderen, bereits vorhandenen Systemen, Kapselung, d.h. Entwurf von Attributen (für die Daten) und Methoden (für die Verarbeitung der Daten) usw.

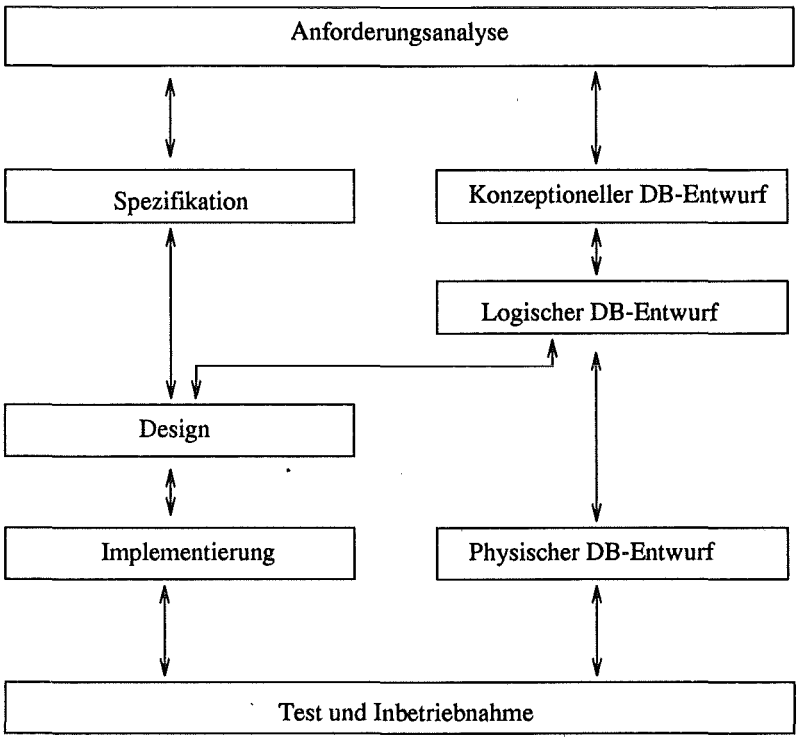


Abbildung 4.2: Klassische Vorgehensweise bei der Entwicklung von Datenbankschema und Anwendungsprogrammen

2. Anschließend folgen zwei Entwurfs- und Entwicklungsabläufe, die parallel und relativ unabhängig voneinander durchgeführt werden können:

   - Die Anwendungsprogramme werden spezifiziert und entworfen. Beim Entwurf der Programme wird auf das Resultat des logischen DB-Entwurfs[3] als Schnittstelle zwischen Anwendungsprogramm und Datenbank zurückgegriffen. Anschließend wird der Entwurf implementiert.
   - Der Datenbankentwurf seinerseits läßt sich weiter in folgende 3 Unterschritte unterteilen ([LS87]):

     (a) Konzeptueller Entwurf, der als Ergebnis die abzubildende Miniwelt semantisch modelliert, beispielsweise mit Hilfe eines Gegenstands-Beziehungsmodells (graphisch als ER-Diagramm, siehe[Che76]). Ein ER-Diagramm veranschaulicht die real abzubildende Umwelt durch Gegenstände (Entities), dargestellt durch Rechtecke, und deren Beziehungen (Relationships), dargestellt durch Rauten.

[3] Wird beispielsweise auf der Grundlage des relationalen Datenmodells entworfen, so liegen an dieser Stelle die Definitionen der Relationen und deren Attribute vor.

(b) Logischer Entwurf, bei dem das konzeptionelle Modell in ein Datenmodell[4] überführt wird. Dabei wird ggfs. eine Normalisierung durchgeführt.

(c) Physischer Entwurf, bei dem die Datenbasis mit Hilfe eines Datenbanksystems implementiert wird.

3. Abschließend folgen Test der Anwendungsprogramme, die auf der Datenbank aufsetzen, sowie Inbetriebnahme des Informationssystems.

- Der Datenbankentwurf für objektorientierte Datenbanken ist meist gleichzusetzen mit dem Entwurf der objektorientierten Anwendungsprogramme. Darauf, daß der Zyklus objektorientierter Systementwicklung nicht analog zu den klassischen Modellen wie dem Wasserfallmodell ([Fai85]) abläuft, hat bereits [HSE90] hingewiesen. Für die objektorientierte Systementwicklung gibt es eine Vielzahl von alternativen Methoden und Beschreibungsmöglichkeiten, z.B. [Boo91, CY91, Mey88, Wed92, BS93, CLF92]. Die dabei entstehende Beschreibung der Klassen wird meist direkt auf das Datenbankschema abgebildet. Es gibt also keine getrennte Entwicklung von Datenbankschema und Anwenderprogrammen, wie bei der klassischen Vorgehensweise.

4.4 Produktmodellierung

Will man Produktinformation über den gesamten Produktlebenszyklus sammeln und verwalten, so geschieht dies in einem sogenannten Produktmodell. Für das benötigte Produktmodell unterscheidet [Eul90] zwei Modellierungsalternativen:

- Die einfachere Modellierung ist, für jede Phase aus dem Produktlebenszyklus ein eigenes Teilmodell zu entwickeln (phasenbezogenes Produktmodell). Auf diese Art erhält man einen direkten Phasenbezug der Teilmodelle zu den Phasen des Zyklus, die Teilmodelle sind relativ einfach und die Datenmenge der einzelnen Teilmodelle ist überschaubar. Die Nachteile der unabhängigen Teilmodelle ist eine eventuelle Redundanz, wenn gleiche Daten in unterschiedlichen Teilmodellen auftreten. Ebenso muß man evtl. eine Datentransformation vornehmen, wenn man Daten nach einem Phasenübergang weiterverwenden will.

- Alternativ dazu kann man ein zentrales, einheitliches Datenmodell entwickeln, das phasenunabhängig sämtliche Produktdaten aufnimmt. In diesem Zusammenhang spricht man auch von kohärentem Produktmodell. Datentransformationen werden damit hinfällig, man hat jedoch die Nachteile von relativ komplexen Datenstrukturen und großen Datenmengen.

[4] An dieser Stelle handelt es sich bei dem Datenmodell um eines der geläufigen Modelle wie z.B. das relationale, das hierarchische oder das Netzwerkdatenmodell.

Die Vermeidung der Datentransformation als Vorteil eines kohärenten Produktmodells führt zu vielfältigen Ansätzen, ein solches kohärentes, integriertes Produktmodell zu schaffen. Um die überaus komplexen Datenstrukturen in den Griff zu bekommen, bedient man sich einer geeigneten Entwurfstechnik, dem Top-Down-Entwurf.

Bei dem Top-Down-Entwurf handelt es sich um eine aus dem Software-Entwurf bekannten Vorgehensweise ([Fai85]), bei der ein komplexes Problem schrittweise in mehrere einfachere Teilprobleme unterteilt wird. Diese Unterteilung wird solange fortgeführt, bis die Teilprobleme überschaubar und damit lösbar werden. Für ein integriertes Produktmodell bedeutet das konkret, das komplexe Gesamtmodell in sogenannte Partialmodelle aufzuteilen, von denen jedes einen logisch abgeschlossenen Teil des Modells darstellt. [Eul90] zeigt eine Unterteilung in die Partialmodelle Funktionsmodell, Geometriemodell, technisches Gestaltsmodell, Technologiemodell, Parametriemodell, Darstellungsmodell, Fertigungsmodell und Spezifikationsmodell sowie mehr betriebswirtschaftlich orientierte Partialmodelle wie Berechnungsmodell und Kostenmodell, wie in Abbildung 4.3 dargestellt ist.

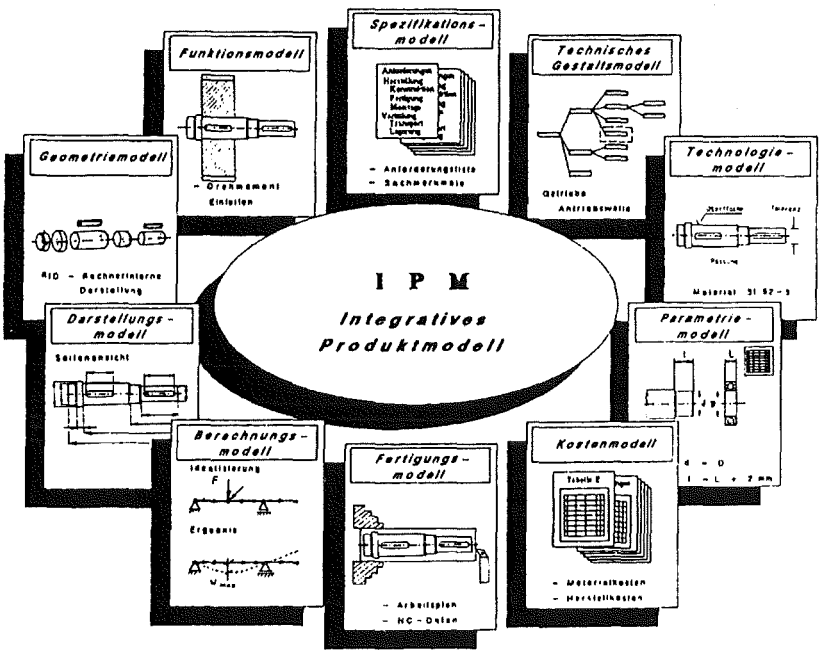


Abbildung 4.3: Integratives Produktmodell mit Partialmodellen (aus [Eul90])

4.4.1 Beispiel eines phasenbezogenen Produktmodells

Am Institut für Mikrostrukturtechnik (IMT) wurde eine Datenbasis erstellt, die eine Speicherung von unterschiedlichen Produktdaten vornimmt, die während des Produktlebenszyklus entstehen. Bei diesem Datenmodell handelt es sich um ein phasenbezogenen Produktmodell. Die dabei betrachteten Produktdaten sind Werte von Parametern, die für jedes Produkt bei der Durchführung von Prozeßschritten relevant waren.

Die Datenbasis ist auf der Grundlage des relationalen Datenmodells ([Cod70]) entworfen (Vorgehensweise siehe Kapitel 4.3) und wird durch das relationale Datenbanksystem Oracle[5] verwaltet.

Der vorhandenen Entwurf der Datenbank, der in [WBS92a] ausführlich untersucht wird, bedingt in dem dynamischen Umfeld, in dem sich der LIGA-Prozeß befindet, einige Probleme. Das gewichtigste soll etwas näher betrachtet werden:

Als Ergebnis des konzeptuellen Datenbankentwurfs, den man in der Regel in einem Gegenstands-Beziehungsdiagramm darstellt, kann ein Produkt wie in Abbildung 4.4 dargestellt modelliert werden: ein Produkt (z.B. Substrat) wird durch eine Sequenz von Prozeßschritten (S_1, S_2, ..., S_n) bearbeitet (z.B. S_1 = Galvanik, S_2 = Ebeam, S_3 = Entschichtung). Ein solches konzeptionelles Schema liegt der vorhandenen Datenbank am IMT zugrunde.

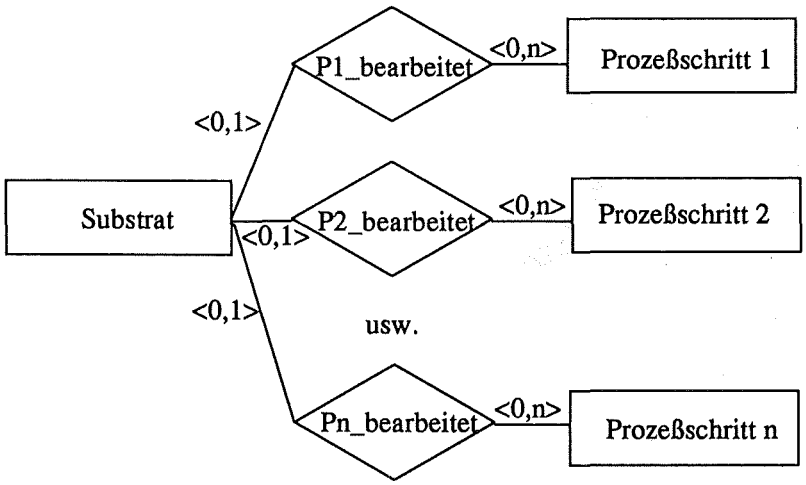


Abbildung 4.4: Herkömmliches Produktmodell

Dieses Produktmodell enthält jedoch Semantik über den Prozeß, mit dem ein Produkt bearbeitet wird, da für jeden Prozeßschritt ein Gegenstand im Schema enthalten ist. Bei der Umsetzung in eine relationale Datenbank wird aus jedem Gegenstand i.d.R. eine Relation, d.h. es wird für jeden Prozeßschritt[6] eine Relation in der Datenbank gehalten. Die Information (z.B. Parameter), die

[5] Oracle wird von der Oracle Corporation entwickelt ([Ora]).

[6] Oder für die Zusammenfassung mehrerer Prozeßschritte, was aber nichts an der Struktur ändert.

zu einzelnen Prozeßschritten gespeichert wird, wird als Attribut in der Relation repräsentiert. Ein Beispiel dazu enthält Abbildung 4.5, die eine Relation *Chrommaske* mit den 5 Attributen *Maskennummer, Auftraggeber, Herstellungsdatum, Fläche* und *Dicke* darstellt.

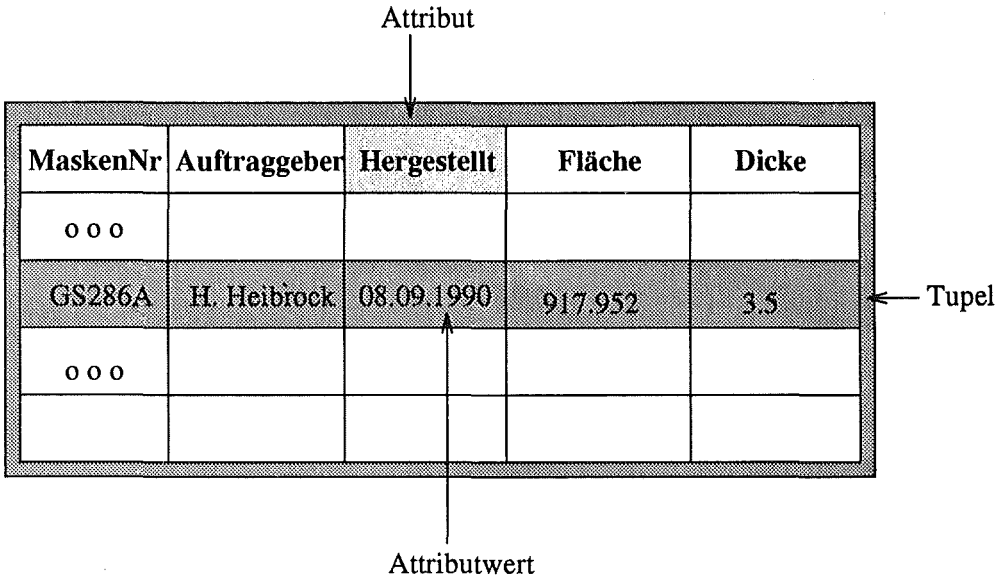


Abbildung 4.5: Eine Beispielrelation des herkömmlichen Produktmodells

Ändert sich nun aber, und damit ist bei der Dynamik des LIGA-Fertigungsprozesses zu rechnen, das Prozeßmodell, so muß das Produktmodell nachgeführt werden:

- Die Einführung eines neuen Prozeßschritts würde eine neue Relation in der Datenbank zur Folge haben.
- Das Löschen eines Prozeßschritts zieht zwei recht unbefriedigende Alternativen nach sich:
  1. Man löscht auch die äquivalente Relation, mit dem Nachteil, daß die dort enthaltenen Daten ebenfalls gelöscht werden und damit verloren sind.
  2. Man beläßt die Relation in der Datenbank, und nimmt damit in Kauf, daß das Schema nicht mehr aktuell ist.
- Das Ändern von Information zu einem Prozeßschritt, z.B. das Hinzufügen eines neuen Parameters, der fortan von Interesse ist, erzwingt die Änderung der Relation, d.h. Hinzufügen oder Löschen von Attributen.

Nur durch diese Änderungen kann die Konsistenz der Prozeßsemantik des Produktmodells mit der Prozeßsemantik des Prozeßmodell gewährleistet werden.

Anschließend müßte zusätzlich fast der gesamte Entwicklungszyklus (Abbildung 4.2) erneut durchlaufen werden.

Da eine solche Vorgehensweise für einen sich in der Veränderung befindlichen Prozeß permanente Änderungen des Produktmodells und damit des Datenbankschemas mit sich bringen würde, muß ein Produktmodell entwickelt werden, das flexibler bezüglich Änderungen des Prozesses ist.

Es ist sinnvoll, das Prozeßwissen soweit irgend möglich aus dem Datenbankschema herauszuhalten, so daß auch bei Änderung des Prozeßwissens das Datenbankschema unverändert bleibt. Einen geeigneten Ansatz dazu zeigen [WBS92b] und [Bra94], wobei beim Datenbankentwurf eine höhere Abstraktionsebene der gespeicherten Information erreicht wird. Man führt z.B. neben einem Gegenstand *Substrat* noch die Gegenstände *Prozeßschritt*, *Parameter* und *Wert* ein. Alle Gegenstände werden durch die Beziehung *bearbeitet* verbunden, wie in Abbildung 4.6 dargestellt.

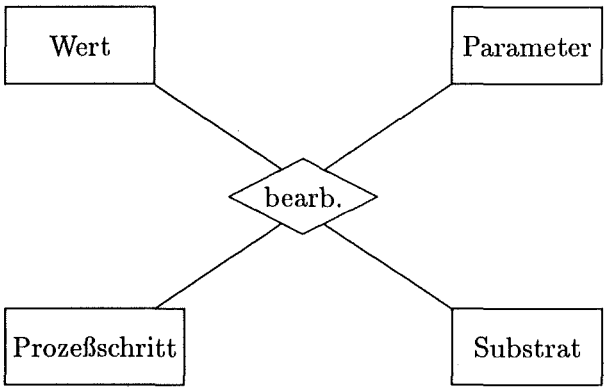


Abbildung 4.6: Prototypisches, semantikarmes Produktmodell

Die entsprechenden Relationen (im Falle einer Übertragung auf das relationale Datenmodell) enthalten die für ein Substrat existierenden Prozeßschritte, deren Parameter und Werte bei der Produktion fortan als Daten. Dies ist in Abbildung 4.7 dargestellt.

Substrat	Prozeßschritt	Parameter	Wert
ooo			
GS286A	Chrommaske	Fläche	917.952
ooo			

Abbildung 4.7: Eine Beispielrelation des semantikarmen Produktmodell

Ändert ein Prozeßschritt seine Parameterliste, so ändert sich der Inhalt der in Abbildung 4.7 dargestellten Relationen, was als Änderung der Daten einfach durchzuführen ist. Das Datenbankschema ist von dieser Änderung nicht betroffen. Gleiches gilt für das Einfügen komplett neuer Prozeßschritte bzw. das Löschen nicht mehr benötigter, alter Schritte. Auch dort ändert man ausschließlich den Inhalt der Relation, nicht aber das Datenbankschema.

Bei der Umsetzung dieser höheren Abstraktionsebene in ein relationales Datenmodell stellt sich die Frage, ob ein Datenbanksystem von den Antwortzeiten her den Leistungsanforderungen genügen kann. Will man Information über einen Prozeßschritt selektieren, so müssen sehr aufwendige Verbundoperationen durchgeführt werden, die zuvor – durch die Semantik im Datenbankschema – überflüssig waren. Es ist also zu erwarten, daß die Realisierung dieses Konzeptes in einem relationalen System sehr viel schlechtere Leistungsdaten erreichen wird, als das bisher eingesetzte System.

Diesen Nachteil kann man durch den Übergang zum objektorientierten Datenmodell umgehen, das eine effizientere Modellierung erlaubt. Dadurch werden die teuren Verbundoperationen hinfällig. Eine Möglichkeit, Prozeßwissen objektorientiert zu modellieren, ist in Kapitel 5.2 gezeigt.

4.4.2 Beispiel eines kohärenten Produktmodells

Für den Austausch von Daten in einer heterogenen Rechnerumgebung ist es notwendig, die Schnittstellen zwischen den unterschiedlichen Systemen festzulegen. In Teilbereichen des Produktlebenszyklus wird daher eine solche Definition bzw. Normung der Schnittstellen durchgeführt. Dies geschieht bisher meist auf nationaler Ebene. Für folgende Bereiche liegt bereits eine Schnittstellendefinition vor ([GAS89a]):

- Die Festlegung der IGES-Schnittstelle (Initial Graphics Exchange Specification) zur Übertragung von Produktinformation, die in Form von Zeichnungen oder strukturierten geometrischen Modellen wie Kanten-, Flächen-, und Volumenmodellen (z.B. in Form von Verknüpfungsmodellen, CGS oder Finite-Elemente-Netze, FEM) vorliegen.
- Die Entwicklung der Schnittstelle CAD*I (CAD-Interface) im Rahmen eines ESPRIT-Projekts, die zur Übertragung strukturierter Geometriedaten für Linien-, Flächen-, und Volumenmodellen sowie für Finite-Elemente-Modelle. Diese Entwicklung stellt eine Verbesserung des IGES-Ansatzes dar.
- Die Entwicklung der Schnittstelle SET (Standard d'Exchange et de Transfer) für die Übertragung sämtlicher im CAD/CAM-Bereich anfallenden Daten.
- Die Festlegung der Schnittstelle VDAFS für die Übertragung von Freiformflächen bei der deutschen Automobilindustrie.

- Die Entwicklung der Schnittstelle PDES (Prodct Data Exchange Specification) in den USA für die Übertragung aller Daten des Produktlebenszyklus.

Diese Ansätze haben zwei Nachteile:

- Zum einen betreffen sie – mit Ausnahme von PDES – nur Teilbereiche der Daten, die im Laufe des Produktlebenszyklus anfallen.
- Zum anderen sind sie meist national begrenzt oder beachten zumindest die international nicht allgemeingültige Normen zu wenig und lassen sich daher international nicht durchsetzen.

Vor diesem Hintergrund wurde eine *internationale* Norm entwickelt und spezifiziert werden, die eine Schnittstelle zum Austausch *aller* produktdefinierenden Daten bietet, die im Produktlebenszyklus entstehen: STEP (**St**andard for the **E**xchange of **P**roduct Model Data), auch unter dem Namen der Norm ISO 10303 ([N.N93a, N.N93b, N.N93c]) bekannt.

Die Entwicklung von STEP wurde von wichtigen Zielsetzungen wie Vollständigkeit, Erweiterbarkeit, Effizienz, Kompatibilität mit bestehenden Normen, Unabhängigkeit von der Rechnerumgebung u.ä. geleitet ([GAS89a]).

Da insbesondere die Forderung der Vollständigkeit, und zwar unabhängig vom späteren Anwendungsgebiet, ein sehr komplexes Gesamtmodell zur Folge haben muß, wurde auch hier die Komplexität der Modellierung dadurch verringert, daß das Gesamtmodell in Partialmodelle (Basismodelle) zerlegt wurde. Die Basismodelle (Partialmodelle) dienen dazu die allen Produkten gemeinsamen Aspekte festzulegen.

In STEP wurden folgende Partialmodelle definiert [GAS89a, GAS89b]:

- Das Partialmodell *Presentation* enthält Information für die Erzeugung der Produktdarstellung, d.h. Information zur Visualisierung des Produkts.
- Das *Geometriemodell* dient der Abbildung von Kurven- und Flächengeometrien. Es enthält Komponenten zur Definition von Koordinatensystemen, Punkt- und Vektorbeschreibungen, sowie Kurven- und Flächenbeschreibungen (analytisch, interpolierend und approximierend).
- Das *Topologiemodell* beschreibt die Nachbarschaftsbeziehungen. Elemente wie Oberflächen, Kanten und Eckpunkten werden mit den entsprechenden Geometrieelementen verknüpft.
- Aufbauend auf das Geometriemodell und das Topologiemodell ermöglicht das *Shape Representation Modell* die eigentliche Definition der Geometrie. Dazu werden verschiedene Geometrien unterstützt, was den Austausch von Daten zwischen Systemen ermöglicht, die auf verschiedenen Geometriebeschreibungen aufbauen:

  - Das *Kantenmodell*, das ein Objekt in eindimensionale Elemente zerlegt.
  - Das *Flächenmodell*, bei dem die Begrenzungsflächen beschrieben werden.
  - Unterschiedliche *Volumenmodelle*, bei denen durch generative Verfahren (z.B. CSG-Modelle, d.h. die Verknüpfung von primitive Volumenelemente wie Quader oder Kugel durch boolesche Operatoren) bzw. akkumulative Verfahren (z.B. Boundary-Representation-Modelle, wobei die Produktgestalt durch die Metrik beschrieben wird) die Geometrien beschrieben werden.

- Das *Form Features Modell* ermöglicht die Bereitstellung allgemeiner Funktionalität, die es anderen Modellen erlaubt, gewisse Geometriebereiche mit Zusatzinformation, beispielsweise über die Fertigung gewisser Bohrungen o.ä. zu versehen.

- Das *Toleranzenmodell*, das sich auf die Gestalt des Produkts beziehende Toleranzangaben (Maß-, Form- und Lagetoleranzen) beschreibt.

- Zur Beschreibung der Materialeigenschaften dient das *Materialmodell*. Es definiert neben einer Materialelastizitätsmatrix auch materialspezifische Koeffizienten wie Dichte, Dämpfungskoeffizient, Wärmekapazität, Wärmeübergangszahl, thermischer Ausdehnungskoeffizient usw. Neben der Beschreibung von homogenen Materialien ist eine Spezifikation von Mischgefügen u.ä. möglich.

- Das *Surface Conditions Modell*, das Oberflächenmerkmale getrennt nach chemischen, physikalischen, geometrischen und fertigungstechnischen Eigenschaften beschreibt.

Neben diesen sogenannten Basismodellen gibt es weitere Partialmodelle, die anwendungsspezifische Information beschreiben, sogenannte Anwendungsmodelle. Bisher berücksichtigte Anwendungsmodelle sind:

- das *Mechanikmodell*, das die Besonderheiten der Mechanik berücksichtigt.

- das *Electrical Modell* für benötigte Daten zur Schaltplanerstellung, Simulation, Layouterstellung sowie Test. Entsprechend beinhaltet das Electrical Modell drei Partialmodelle, das Electrical Schematic Modell, das Electrical Schematic Modell und das LEP-Modell (Layered Electrical Product).

- das Partialmodell *Architecture, Engineering and Construction* berücksichtigt Besonderheiten im Bereich Schiffsbau und Bauwesen, einem der aktuellen Spezifikationsbereiche von STEP.

Neben Basismodellen und Anwendungsmodellen gibt es weitere Partialmodelle, die zur Beschreibung bestimmter Merkmale des Produktlebenszyklus dienen. Es sind dies:

- Daten zur Arbeitsplanung
- Qualitätssicherungsdaten
- Modelle zur Berücksichtigung von Produktlogistik, Produktvertrieb, Produktentsorgung u.a.

Die Auflistung dieser Partialmodelle zeigt, daß es nicht sinnvoll sein kann, die Gesamtspezifikation von STEP für die Speicherung von LIGA-Produktdaten zu verwenden. So sind beispielsweise Partialmodelle, die Besonderheiten des Schiffsbaus berücksichtigen, irrelevant für die LIGA-Technik.

Ein möglicher Weg kann aber eine Beschränkung auf einzelne Partialmodelle oder eine Teilmenge daraus sein, die die Anforderungen der LIGA-Mikrostrukturen abdecken. Man hat bei der Verwendung eines objektorientierten Datenbanksystems auch die Möglichkeit, die für LIGA-Produkte relevanten Teile selbst zu entwerfen und zu implementieren bzw. diese mit Teilen von STEP zu kombinieren.

Kapitel 5

Partialmodelle für Entwurf und Fertigung von Mikrostrukturen

Im vorausgegangenen Kapitel ist gezeigt, daß die Zerlegung in Partialmodelle ein adäquater Weg ist, die Komplexität der Produktmodellierung zu verringern. Das folgende Kapitel beschreibt nun zwei solche Partialmodelle, die für den Entwurf und die Fertigung von Mikrostrukturen erarbeitet wurden. Dabei handelt es sich um ein Partialmodell für die Speicherung von geometrischen Strukturen und um ein Partialmodell für die Speicherung von Prozeßinformation.

5.1 Ein Partialmodell der Entwurfssicht für Mikrostrukturen

In Abbildung 5.1[1] wird eine Aktivitätenkette dargestellt, die bei der Herstellung von Mikrostrukturen typischerweise durchlaufen wird. Das Beispiel soll zur Konkretisierung der bisher vorgestellten Konzepte für ein Systems zur Erfassung und Weiterverarbeitung von Produktinformation dienen und das Partialmodell für Geometriedaten dabei einführen. Dabei wird insbesondere auf die Aktivität *Vervielfältigung* und die von ihr benutzten Ein- und Ausgabeobjekte (IGES-Datei, GDSII-Datei und Vermessungsdatei) eingegangen.

Der Zusammenhang der Aktivitäten, der zur Ausführung dieser Aktivitäten benötigten Werkzeuge und der von diesen Werkzeugen verarbeiteten Dateiformate ist ebenfalls in Abbildung 5.1 dargestellt. CAD-System, Elektronenstrahl-Schreiber und Vermessungssystem COSMOS-2D [BGH91] sind durch eine White-Box-Integration über Zwischendatei (siehe Kapitel 3.3.2) eingebunden. Alle drei

[1] Die Verbindung zwischen *CAD-Entwurf* und *Vervielfältigung* ist gestrichelt, weil auf die Darstellung der Aktivität *IGES-Transfers auf SUN* (vergleiche Abbildung 2.3) verzichtet wurde.

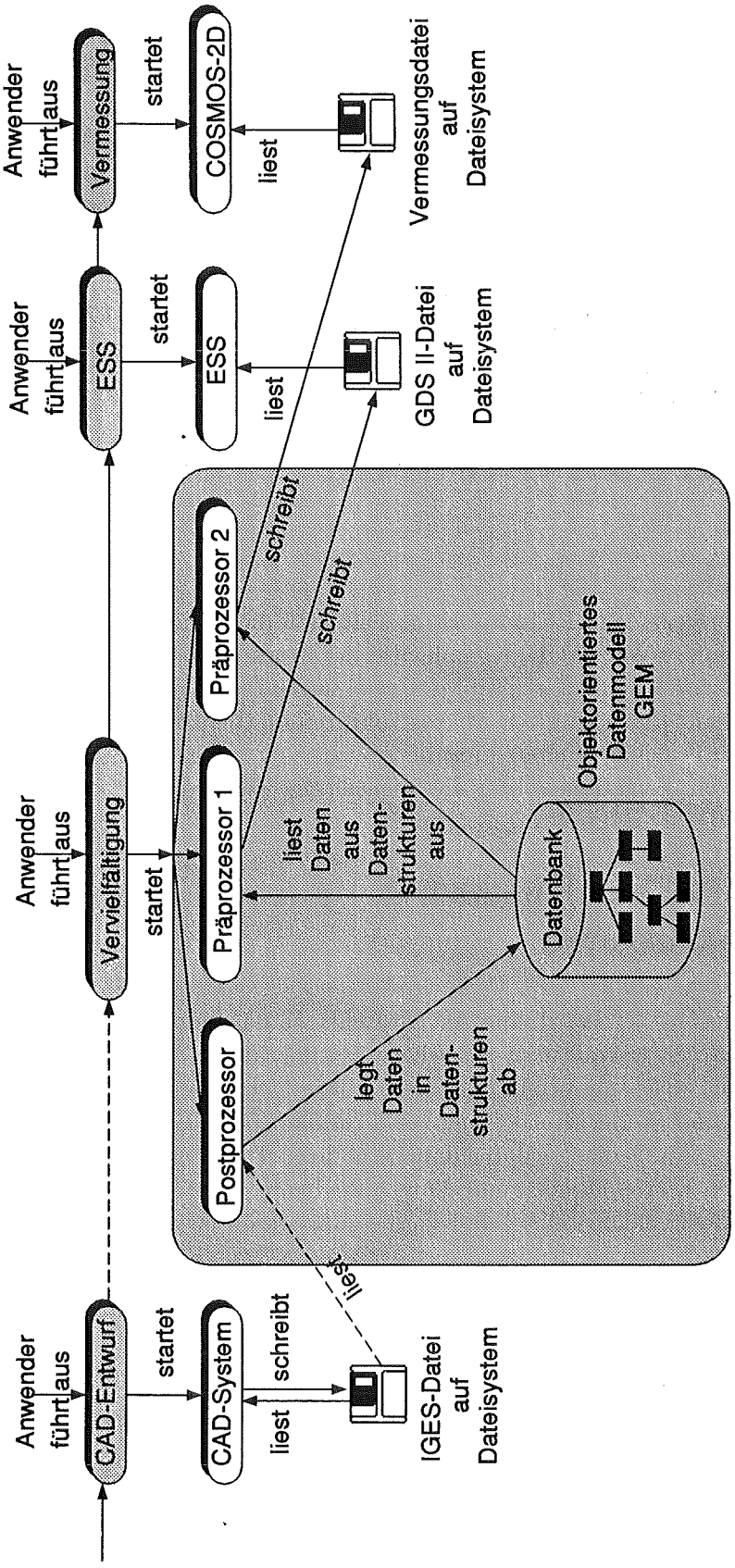


Abbildung 5.1: Aktivitätenkette mit Werkzeugen und Daten

Werkzeuge arbeiten dateiorientiert, doch wird die Information der Dateien unter Einbeziehung eines objektorientierten Geometriemodells (GEM) verarbeitet.

Die Vorteile, die ein solches Geometriemodell bietet, insbesondere wenn – wie im vorliegenden Fall – Daten zwischen Werkzeugen mit unterschiedlichen Formaten ausgetauscht werden sollen, wurden in Kapitel 3.3.2 bereits erörtert. Für die Konvertierung der Daten in GEM und die Erzeugung von Dateien aus GEM sind Prä- und Postprozessoren notwendig.

5.1.1 Die Aktivität *CAD-Entwurf*

Die Konstruktion der Mikrostrukturen erfolgt mit Hilfe eines CAD-Systems für den mechanischen Entwurf (MCAD). Die Bemaßung von Zeichnungen beim CAD-Entwurf, die für das spätere Erzeugen von Vermessungsaufträgen notwendig ist, wird in Anhang A.1 beschrieben.

Das CAD-System stellt für den Datenaustausch mit externen Systemen die Schnittstellendefinition IGES (Initial Graphics Exchange Specification) zur Verfügung. Entsprechend wird nach Beendigung der Aktivität *CAD-Entwurf* eine IGES-Datei auf das Dateisystem geschrieben.

5.1.2 Die Aktivität *Vervielfältigung*

Für die Vervielfältigung bestehen zwei mögliche Optionen. Entweder man erzeugt aus den Daten einer Figur ein neue Figur (rotiert, skaliert und/oder verschoben) oder ein zweidimensionales Figurenfeld identischer Figuren. Die neue Figur entsteht durch Vervielfältigung der Orginalfigur unter Angabe einer Rotationsmatrix (einschließlich möglicher Skalierung der Figur) und einem Translationsvektor. Das zweidimensionale Figurenfeld entsteht durch Vervielfältigung der Orginalfigur unter Angabe der Dimension des Feldes und des Versatzes der Figuren, jeweils in X- und Y-Richtung.

Die Aktivität *Vervielfältigung* stößt drei Verarbeitungsschritte an:

1. **Postprozessor:**

   Für die Kommunikation mit nachfolgenden Werkzeugen wurde ein Postprozessor entwickelt, der die IGES-Daten vom Dateisystem liest und in das objektorientierte Geometriemodell überträgt.

   Aus diesem Geometriemodell sollen Konstruktionsdaten an einen Elektronenstrahl-Schreiber (ESS) zur Erzeugung von Ätzmasken für die Herstellung der konstruierten Mikrostrukturen, sowie Vermessungsdaten an das Bildverarbeitungssystem COSMOS-2D zur Vermessung dieser Ätzmasken abgeleitet werden.

2. **Präprozessor 1:**

   Die für den Elektronenstrahl-Schreiber verfügbare Schnittstelle für den Datenaustausch ist GDSII. Somit ist es erforderlich, mit Hilfe eines Präprozessors Daten aus dem objektorientierten Geometriemodell auszulesen, in das GDSII-Format zu konvertieren und eine GDSII-Datei auf das Dateisystem zu schreiben. Bei diesem Vorgang ist es möglich, aus den beim MCAD-Entwurf konstruierten Einzelstrukturen (sogenannten Zellen) Vervielfachungen zu erzeugen. Die GDSII-Datei wird vom *Elektronenstrahl-Schreiber* bei Ausführung der Aktivität *ESS* gelesen.

3. **Präprozessor 2:**

   Für das Bildverarbeitungssystem COSMOS-2D wird durch einen weiteren Präprozessor aus dem objektorientierten Geometriemodell Information ausgelesen und eine Vermessungsdatei erzeugt, die Vermessungsaufträge enthält. Diese Vermessungsdatei wird ebenfalls auf das Dateisystem geschrieben, von wo aus sie vom Vermessungssystem bei Ausführung der Aktivität *Vermessung* gelesen wird.

Im Anhang werden die Dateiformate, sowie die Funktionsweisen der gesamten Post- und Präprozessoren näher beschrieben. Kapitel A.2 im Anhang beschreibt das IGES-Format und den dafür entwickelten Postprozessor für die Umsetzung von IGES-Daten in das objektorientierte Geometriemodell. Kapitel A.3 beschreibt das GDSII-Format und die Erzeugung von GDSII-Dateien aus dem objektorientierten Geometriemodell durch einen Präprozessor. In Kapitel A.4 wird schließlich die Erzeugung von Vermessungsdateien aus dem objektorientierten Geometriemodell durch einen Präprozessor erläutert.

5.1.3 Das objektorientierte Geometriemodell (GEM)

Bei dem objektorientierten Geometriemodell [SES93a] handelt es sich um ein Partialmodell, welches sich zur Zeit auf die Speicherung zusammenhängender 2D-Strukturen (ohne Splines) beschränkt.

Die Hierarchie der definierten Klassen ist in Abbildung 5.2 gemäß der in Coad/Yourdon [CY91] verwendeten Notation dargestellt[2].

Die Klassenhierarchie zielt auf einen strukturierten Aufbau der Geometrieelemente ausgehend von einer Basisklasse *Topologische Repräsentation* unter Berücksichtigung und Ausnutzung von Vererbung.

- Eine Instanz der Klasse *Topologische Repräsentation* enthält als Attribute jeweils einen Vorgänger und einen Nachfolger. Alle abgeleiteten Klassen erben diese Attribute. Somit hat beispielsweise eine Ecke eine Vorgänger-

[2] Um die Übersichtlichkeit zu erhalten werden nur die Klassen, nicht aber die Attribute und Methoden dargestellt.

und Nachfolgerkante, eine Kante ihre Anfangs- und ihre Endecke als Vorgänger bzw. Nachfolger.

Zur einfachen Handhabung der Vervielfachungsinformation erhält jedes Objekt der Klasse *Topologische Repräsentation* das Attribut Verschiebung.

- In der Klasse *Punkt* wird die Position als Attribut in Form von x/y-Koordinaten bereitgestellt.
- *Ecken* erhalten gegenüber Punkten keine weiteren geometrischen Attribute. Es können aber topologische Eigenschaften angegliedert werden.

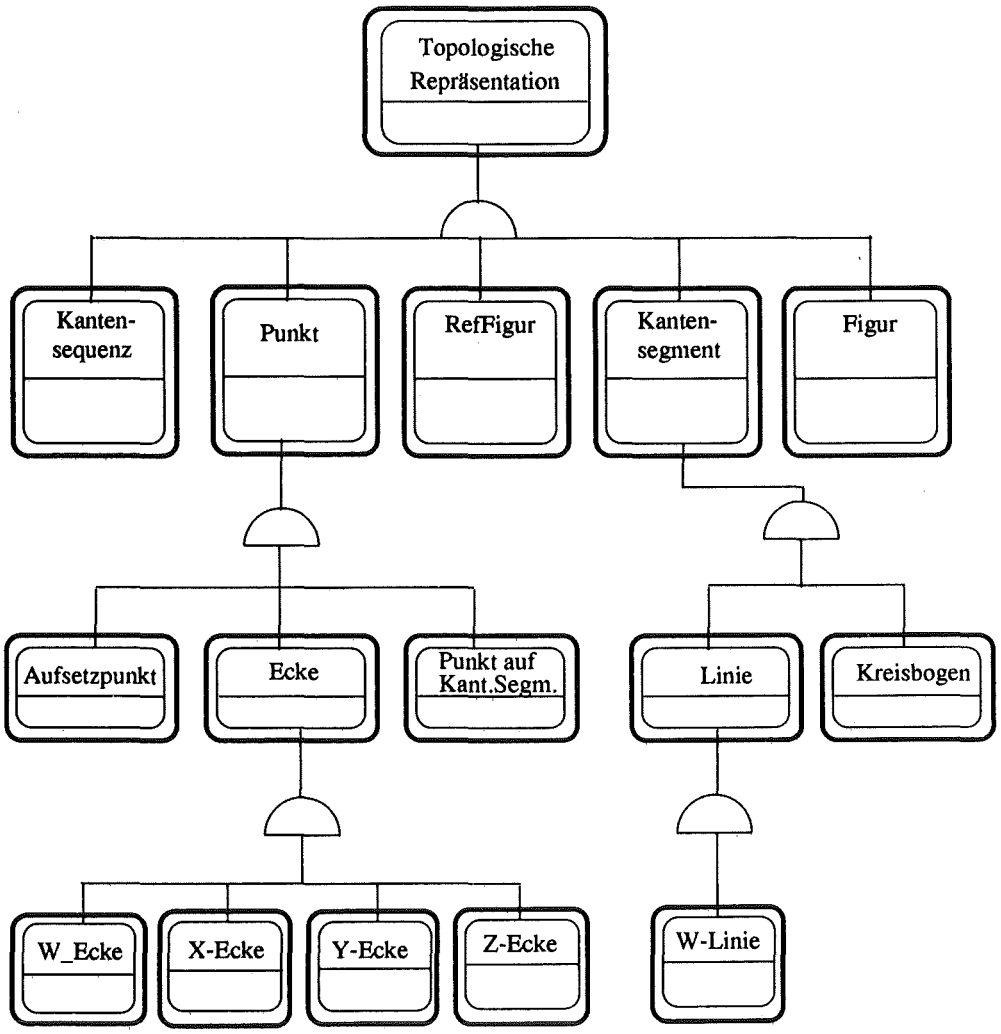


Abbildung 5.2: Klassenhierarchie von GEM

- Ein *Aufsetzpunkt* repräsentiert einen gesamten Bildausschnitt. Zu diesem gehört die Information Objektivvergrößerung und Bildfeldgröße, sowie die Liste der in diesem Bildfeld abzuarbeitenden Vermessungsbefehle.

- Ein *Kantensegment* hat als Attribute die Position des Anfangspunktes sowie des Endpunktes, die Länge und die Bounding Box des Kantensegmentes, d.h. das kleinste achsenparallele Rechteck, in das das Kantensegment einbeschrieben werden kann.

- Diese Attribute sind zur Beschreibung einer *Linie* ausreichend, beim *Kreisbogen* interessieren zusätzlich noch der Mittelpunkt, der Radius und die Drehrichtung. *W-Linien* sind Linien, die den Rand des Bildfeldes schneiden, neben Anfangspunkt und Endpunkt wird ein Zeiger auf die zugehörige Linie notwendig.

- *Punkte auf Kantensegmenten* enthalten neben der Position eine Referenz auf das Kantensegment, auf dem sie liegen.

- Bei *Kantensequenzen* (geschlossene Folgen von Kantensegmenten) interessiert die Orientierung sowie die Bounding Box, d.h. das kleinste achsenparallele Rechteck, in dem alle zur Kantensequenz gehörenden Kantensegmente liegen.

- Eine *Figur* enthält neben Verwaltungsinformation (Datum der letzten Modifikation, Dimension der Zahlenwerte u.ä.) die Listen für Ecken, Kantensegmente, Kantensequenzen, Punkte auf Kantensegmenten, Aufsetzpunkte und Subfiguren. Bei der Figur wird auch die Vervielfachungsinformation gespeichert, zum einen die Wiederholungsfaktoren in horizontaler und vertikaler Richtung, zum andern die jeweiligen Verschiebungen.

- Mit *RefFiguren* wird die Klasse von Figuren bezeichnet, die sich aus anderen Figuren durch lineare Abbildungen herleiten lassen. Im MCAD-System wird dabei nicht die komplette Figureninformation gespeichert, sondern lediglich die Abbildungsvorschrift zu deren Erzeugung. Analog enthält die Klasse der referenzierten Figuren neben Namen und eindeutiger Identitätsnummer der RefFigur noch die Transformationsmatrix und den Verschiebungsvektor sowie einen Zeiger auf die Figur, aus der die RefFigur durch Anwendung der linearen Abbildung erzeugt werden kann.

Für die Vermessung existiert eine gesonderte Klassenhierarchie.

5.2 Ein Partialmodell der Fertigungssicht für Mikrostrukturen

Das im folgenden vorgestellte Partialmodell der Fertigungssicht hatte als Ziel, das in [Bra94] eingeführte Prozeßmodell auf Grundlage eines objektorientierten Datenbanksystems zu implementieren. Dieses Prozeßmodell verfolgt dem Ansatz, die Prozeßsemantik vom Prozeßmodell zu trennen, wie in Kapitel 4.4.1 vorgeschlagen. Es ist dadurch möglich, folgende Änderungen in der Prozeßsemantik durchzuführen, *ohne* daß Änderungen am Datenbankschema bzw. den darauf aufsetzenden Anwendungsprogrammen durchgeführt werden müssen:

- Einfügen neuer Prozeßschritte[3].
- Einfügen von neuen Prozeßvarianten durch die Angabe von Alternativen zu beliebigen Prozeßschritten bzw. Prozeßschrittnetzen.
- Einfügen von neuen Prozeßvarianten durch die Definition von neuen Vorgänger-/Nachfolgerschritten.
- Einfügen von Information, die an Prozeßschritte geknüpft ist, durch Anhängen von beliebigen Attributen an beliebige Prozeßschritte.
- Ändern und Löschen beliebiger Prozeßschritte.
- Ändern und Löschen von Prozeßvarianten durch das Ändern und Löschen von Alternativen.
- Ändern und Löschen von Prozeßvarianten durch das Ändern und Löschen von Vorgänger-/Nachfolgerschritten.
- Ändern und Löschen von Information in Form von Attributen und ihren Werten.

5.2.1 Das Prozeßmodell

Die Klassenhierarchie des Prozeßmodells ist in Abbildung 5.3 gemäß der in Coad/Yourdon [CY91] verwendeten Notation dargestellt.

Die wichtigsten Klassen des Prozeßmodells sind:

- Die Klasse *Prozeßmodell* enthält alle Prozeßschritte eines Prozeßmodells. Sie enthält außerdem Methoden, die notwendig sind, um Prozeßschritte konsistent anzulegen, zu erweitern oder zu löschen. Zusätzlich enthält die Klasse *Prozeßmodell* einen *Graphen* zur Darstellung des Prozeßschrittnetzes.

[3] Die Erweiterung des Modells von Prozeßschritten auf Aktivitäten erfolgte erst nach der Implementierung des hier vorgestellten Partialmodells. Daher wird im folgenden von Prozeßschritten die Rede sein.

- In der Klasse *Prozeßschritt* werden die Prozeßschritte repräsentiert. Prozeßschritte werden durch beliebig viele unterschiedliche Attribute beschrieben. Außerdem enthält jeder Prozeßschritt Informationen über seine Einordnung in den Prozeßablauf, d.h. insbesondere Konkretisierungen, zeitliche Vorgänger und Nachfolger sowie alternative Prozeßschritte.

- Die Klasse *Attribute* ist eine abstrakte Klasse, d.h. sie besitzt selbst keine Instanzen. Sie enthält daher nur die Definition generischer Methoden, die für alle Attributklassen aufgerufen werden können (darstellen, ändern, löschen). Von der abstrakten Klasse *Attribute* werden spezielle Attributklassen wie *Dokument, Gleitkommazahl, Zeichenkette* und *Bild* abgeleitet. Eine Attributklasse (z.B. *Dokument*) enthält die Instanzen aller Attribute dieses Typs (z.B. alle Dokumente).

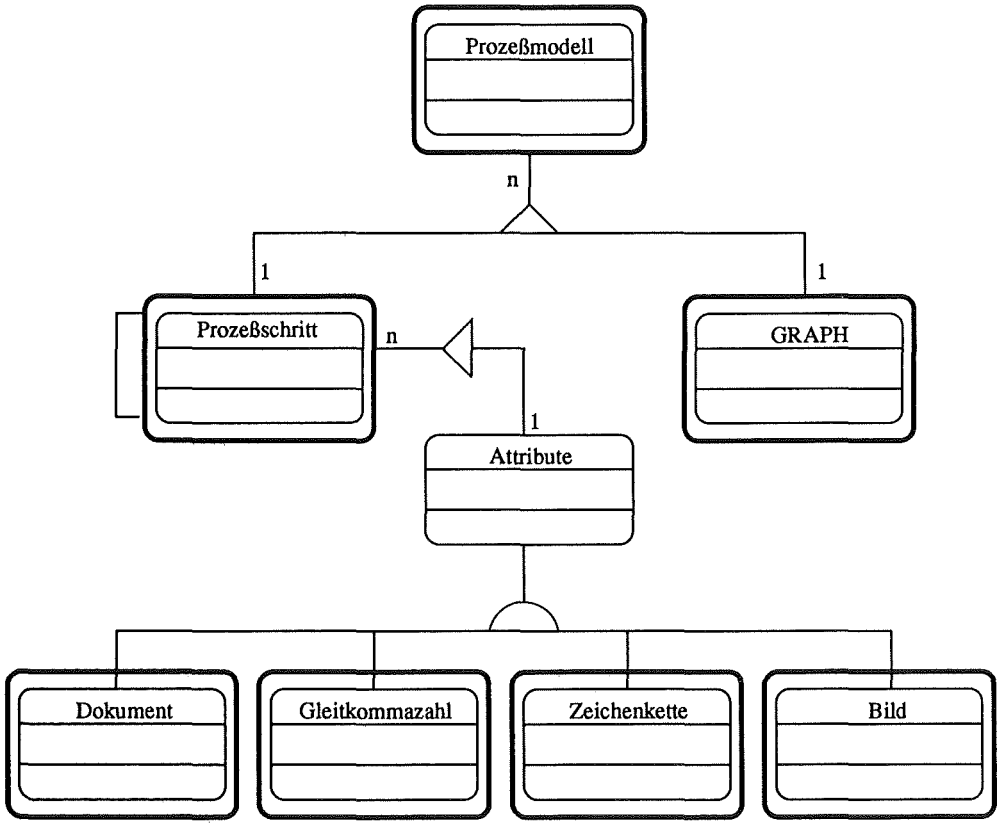


Abbildung 5.3: Die Klassenhierarchie des prototypisch implementierten Partialmodells

Für das oben beschrieben Partialmodell der Fertigungssicht wurde ein System prototypisch implementiert, mit dessen Hilfe der Inhalt des Partialmodells geändert und dargestellt werden kann.

Die Implementierung und Handhabung dieses prototypischen Systems sind in Anhang B beschrieben.

Der Entwurf des Schemas mit dem Ansatz, die Prozeßsemantik vom Prozeßmodell zu trennen, hat sich bewährt. Die oben genannten Anforderung werden vom Partialmodell erfüllt. Von der Prozeßsicht aus kann der Anwender Information eingeben, verändern, abrufen und löschen, die als Dokumente, Zeichenketten, numerische Werte oder Bilder an Prozeßschritte geknüpft ist. Dafür wurden die Werkzeuge *lemacs* (als Dokumenteneditor) und *xv* (als Bildbetrachtungs- und Bildverarbeitungswerkzeug) eingebunden. Diese Werkzeuge wurden beim Anwählen der entsprechenden Funktion automatisch, d.h. für den Anwender transparent, gestartet. Auch das entsprechende Versorgen der Werkzeuge mit den Daten aus der Datenbank, bzw. das Speichern der Ergebnisse in der Datenbasis wurde wie gewünscht realisiert.

5.2.2 Bemerkungen zu den Partialmodellen

Anhand der für das Partialmodells der Entwurfssicht aufgeführten Beispielkette läßt sich erkennen, wie komplex die Zusammenhänge der Aktivitäten, Werkzeuge und Daten sind, selbst wenn man sich nur auf die Analyse einer Sequenz von vier Aktivitäten und den dabei benötigten vier unterschiedlichen Datenformaten (IGES-, GDSII-, Vermessungsdatei sowie dem Geometriemodell GEM) beschränkt. Will man die Zusammenhänge für den gesamten Ablauf des LIGA-Prozesses erfassen, so ist dazu ein rechnergestütztes System erforderlich, das die in Kapitel 2.1 bis 6 vorgestellten Konzepte realisiert.

Als Prototyp für die Visualisierung des Partialmodells der Fertigungssicht wurden ausschließlich Teile des Prozeßsicht implementiert. Die Implementierung ist entsprechend den Anforderungen aus Kapitel 2 so zu erweitern, daß die Prozeßsicht vollständig und darüber hinaus die Produktsichten und die Produktionssicht vom System angeboten werden können. Ansätze der Modellierung dieser Erweiterungen enthält Kapitel 7.

Kapitel 6

Entwurf einer Systemarchitektur von *PRAXIS*

Die Komponenten

- Werkzeug zur Visualisierung der Systemsichten,
- erweiterbare Sammlung von Werkzeugen und
- objektorientiertes Datenbanksystem,

aus denen *PRAXIS* besteht, sowie die daraus resultierende Gesamtarchitektur von *PRAXIS* sind in Abbildung 6.1 dargestellt.

6.1 Werkzeug zur Visualisierung der Systemsichten

Ein Werkzeug zur Visualisierung der gewünschten Systemsicht (Produkt- Prozeß- oder Produktionssicht) bildet die Komponente, durch die der Benutzer mit *PRAXIS* interagiert. Das Werkzeug stellt die vom Benutzer gewünschte Sicht bereit und ermöglicht diesem anschließend die Interaktion (z.B. Navigation und Abrufen von Information) mit dieser Sicht. Je nach Benutzergruppe sind unterschiedliche Sichten von Bedeutung und verschiedene Bearbeitungsvorgänge möglich.

Es werden folgende Personengruppen unterschieden, die mit *PRAXIS* arbeiten:

- **Prozeßexperten**

  Die Prozeßexperten werden mit der Prozeßsicht arbeiten. Sie speichern und aktualisieren das Prozeßwissen mit Hilfe der Prozeßsicht.

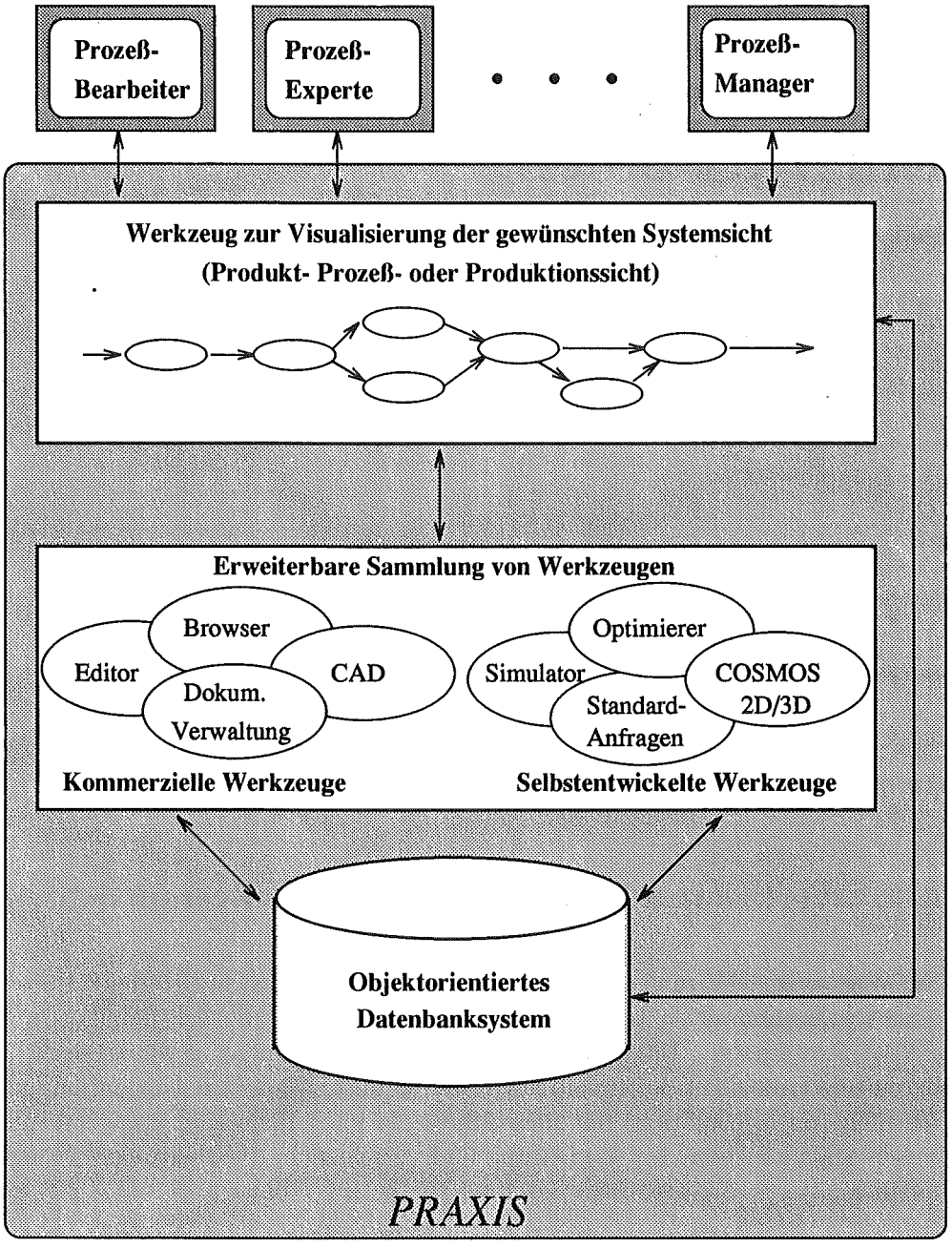


Abbildung 6.1: Die Gesamtarchitektur von *PRAXIS*

Dies geschieht z.B. durch Aktualisieren der Prozeßvarianten (Hierarchie, Vorgänger-/Nachfolgerbeziehung, Alternativaktivitäten, Nebenläufigkeiten oder Ein-/Ausgabeobjekttypen).

Die Aktualisierung der Prozeßvarianten kann z.B. mit Hilfe eines graphischen Editors durchgeführt werden, durch den die Prozeßsicht bearbeitet werden kann. Sind die Aktualisierungen der Prozeßsicht konsistent abgeschlossen, so wird eine neue Version der Prozeßsicht festgeschrieben. Neu zu fertigende Produkte werden dann auf Grundlage dieser neuen Version hergestellt.

Die Prozeßexperten sind der einzige Personenkreis, der die Prozeßsicht ändert. Alle anderen Personengruppen können über die Prozeßsicht nur Informationen abfragen.

- **Projektmanager**

  Ein Projektmanager erstellt auf Grundlage der neusten Version der Prozeßsicht eine neue Produktsicht für die Fertigung eines Produkts. Dies geschieht durch die Auswahl der Aktivitäten aus der Prozeßsicht, die für die Herstellung dieses konkreten Produkts notwendig sind. Er erstellt somit eine Art Vorgehensmodell dafür, wie die Bearbeiter der Aktivitäten während der Fertigung des Produkts vorzugehen haben. Dieses Vorgehensmodell stellt er den Bearbeitern in Form der Produktsicht zur Verfügung. Durch die Produktsicht kann er den Status eines einzelnen Produkts abrufen und damit das Projekt überwachen.

- **Projektkoordinator**

  Der Projektkoordinator steuert und überwacht die parallele Produktion verschiedener Produkte. Er nutzt dazu die Möglichkeiten der Produktionssicht, in die alle Produkte einbezogen werden, die sich in der Produktion befinden. Durch die Produktsicht kann der Projektkoordinator den Status eines einzelnen Produkts abrufen.

- **Bearbeiter**

  Die Bearbeiter der einzelnen Aktivitäten werden mit der Produktsicht des von ihnen gerade bearbeiteten Produkts arbeiten.

  Die Produktsicht stellt für die Bearbeiter ein Vorgehensmodell bei der Herstellung einer LIGA-Mikrostruktur dar. Aus der Produktsicht ist ersichtlich, welches die nächsten Aktivitäten bei der Bearbeitung eines Produktes sind.

  Außerdem bekommt der Bearbeiter Information darüber, welche Eingabeobjekte, Ressourcen und Materialien für die Durchführung dieser Aktivitäten benötigt werden.

  Für die Durchführung der Aktivität benötigte Werkzeuge werden während der Durchführung der Aktivität vom Bearbeiter gestartet und dabei automatisch mit den richtigen Daten versorgt.

Werkzeuge, die die Information archivieren, die bei der Durchführung der Aktivität entstehen, werden während oder nach der Durchführung der Aktivität ebenfalls gestartet.

6.2 Erweiterbare Sammlung von Werkzeugen

In die erweiterbare Sammlung von Werkzeugen können beliebige Werkzeuge eingebunden und anschließend den entsprechenden Aktivitäten zugeordnet werden. Diese Zuordnung geschieht beispielsweise dadurch, daß für jede Aktivität ein Skript erstellt wird, durch das das Werkzeug bei der Ausführen der Aktivität gestartet wird.

Die Daten, die zur Verarbeitung von den Werkzeugen benötigt werden und die Information, die bei der Bearbeitung entsteht, wird in der gewählten Integrationsweise abgelegt, und vom objektorientierten Datenbanksystem der untersten Komponente verwaltet.

6.3 Objektorientiertes Datenbanksystem

Das objektorientierte Datenbanksystem, von dem heraus die Werkzeuge mit den entsprechenden Daten versorgt werden, und das das Speichern der erzeugten Daten übernimmt, bildet die unterste Komponente von *PRAXIS*. Zusätzlich zu den Daten der Werkzeuge verwaltet das objektorientierte Datenbanksystem die Information zu den Sichten, die vom Werkzeug zur Visualisierung der Sichten benötigt wird.

Die Anforderungen an das objektorientierte Datenbanksystem liegen in der Erfüllung des Standards ODMG-93 der Object Database Management Group, der in Kapitel 4.2 eingeführt und in [Cat93] ausführlich beschrieben wird.

Kapitel 7

Objektorientierte Modellierung von *PRAXIS*

Das im Anschluß präsentierte objektorientierte Modell von *PRAXIS* berücksichtigt die Anforderungen aus Kapitel 2 bis 4.

Es beinhaltet eine ausführliche Beschreibung der einzelnen Klassen als Hauptkomponenten des Entwurfs. Dabei werden die

- Komponenten (Attribute), aus denen die Objekte der einzelnen Klassen bestehen,

sowie die

- Verarbeitungsfunktionen (Methoden) für die Objekte der Klassen

erläutert.

Parallel dazu erfolgt die Darstellung der Zusammenhänge der Klassen. Dazu wird – wie in den vorausgegangen Kapiteln – die graphische Notation von Coad/Yourdon [CY91] verwendet.

Die Sprachdefinition der einzelnen Klassen erfolgt im Anhang C.

7.1 Die Klasse *praxis_diagram*

Ein *praxis_diagram* beschreibt eine komplette Sicht, d.h. die Prozeßsicht, eine Produktsicht oder die Produktionssicht. Da eine Sicht sehr komplex sein kann (d.h. sehr viele Aktivitäten enthalten kann), wird sie gemäß den Empfehlungen von [Har84] in weniger komplexe Teile, sogenannte *praxis_pages* zerlegt. Ein *praxis_diagram* besteht demnach aus einzelnen *praxis_pages*, wie in Abbildung 7.1 angedeutet.

praxis_pages auf unterschiedlichen Hierarchieebenen haben unterschiedlichen Detaillierungsgrad. Die *praxis_page* der obersten Ebene gibt einen groben Überblick über die gesamte Sicht, während eine *praxis_page* auf unterer Ebene einen recht detaillierten Überblick über einen sehr kleinen Ausschnitt der Sicht liefert.

Das *praxis_diagram*, das in Abbildung 7.1 dargestellt wird, besteht aus drei *praxis_pages*: Der in der Hierarchie obersten *praxis_page* mit der Aktivität A, der daraus verfeinerten *praxis_page* mit den Aktivitäten A_1 bis A_4 und der aus Aktivität A_2 verfeinerten *praxis_page* mit den Aktivitäten A_{21}, A_{22} und A_{23}.

Die logischen Zusammenhänge der *praxis_pages* bei der Darstellung einer Sicht wird ausschließlich über die Verfeinerungsmöglichkeit der Aktivitäten bestimmt.

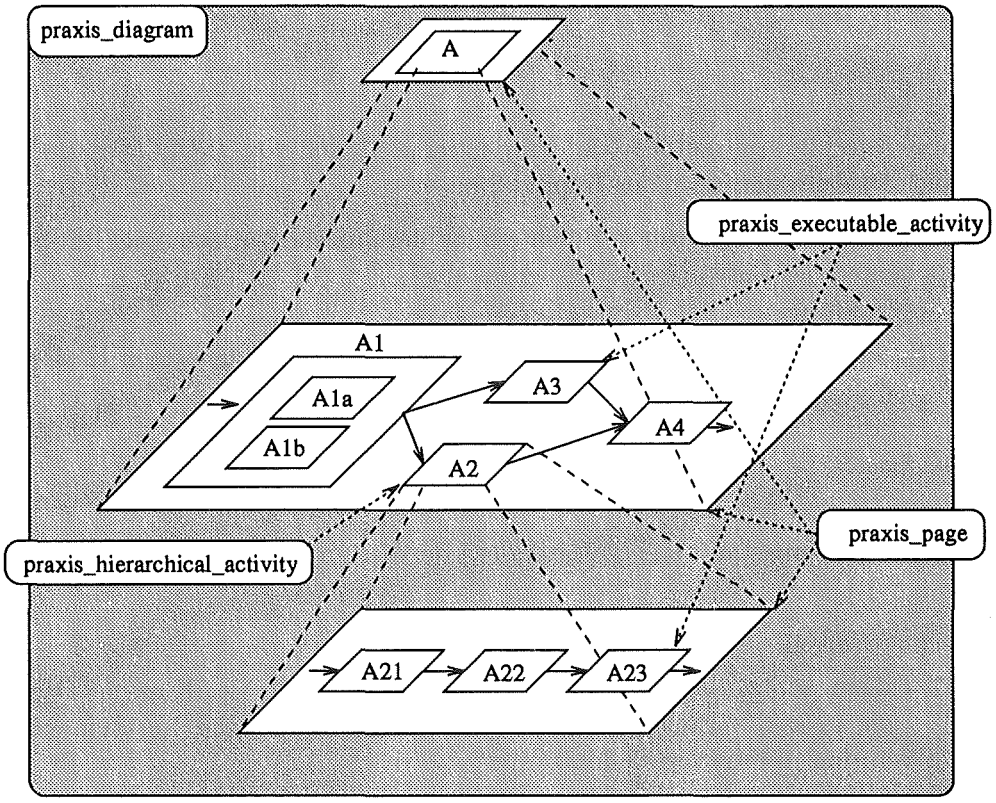


Abbildung 7.1: Aufbau eines *praxis_diagram*

- ❒ Ein *praxis_diagram* besteht aus der Liste der Seiten (*praxis_pages*) einer Sicht (Prozeß-, Produkt- oder Produktionssicht) (siehe Abbildung 7.3).

- ○ Da der Anwender bestimmt, mit welcher Sicht er arbeiten möchte, muß das Laden und Speichern von kompletten *praxis_diagrams* (Sichten) unterstützt werden (siehe Abbildung 7.3).

- ○ Es muß möglich sein, in *praxis_diagrams* neue Seiten und damit neue Abstraktionsebenen einzufügen, und alte zu löschen (siehe Abbildung 7.3).

7.2 Die Klasse *praxis_page*

Eine *praxis_page* ist eine komplette Seite als Komponente von *praxis_diagram*. Sie enthält einen überschaubaren Ausschnitt der entsprechenden Sicht mit den zugehörigen Aktivitäten und den zugehörigen Daten, d.h. Eingabe-, Ausgabeobjekte sowie Attribute.

Abbildung 7.2 enthält beispielhaft den Aufbau einer *praxis_page*. Die *praxis_activities* sind in der Reihenfolge ihrer Abarbeitung miteinander verknüpft. Dadurch wird die kausale Vorgänger-/Nachfolgerbeziehung realisiert. Parallelität ist visualisierbar, indem eine *praxis_activity* mehr als eine Nachfolgeraktivität hat. Abbildung 7.2 zeigt neben den Aktivitäten noch Container für Ein-/Ausgabeobjekte (*praxis_things*, links) und Container für die Attributwerte (rechts).

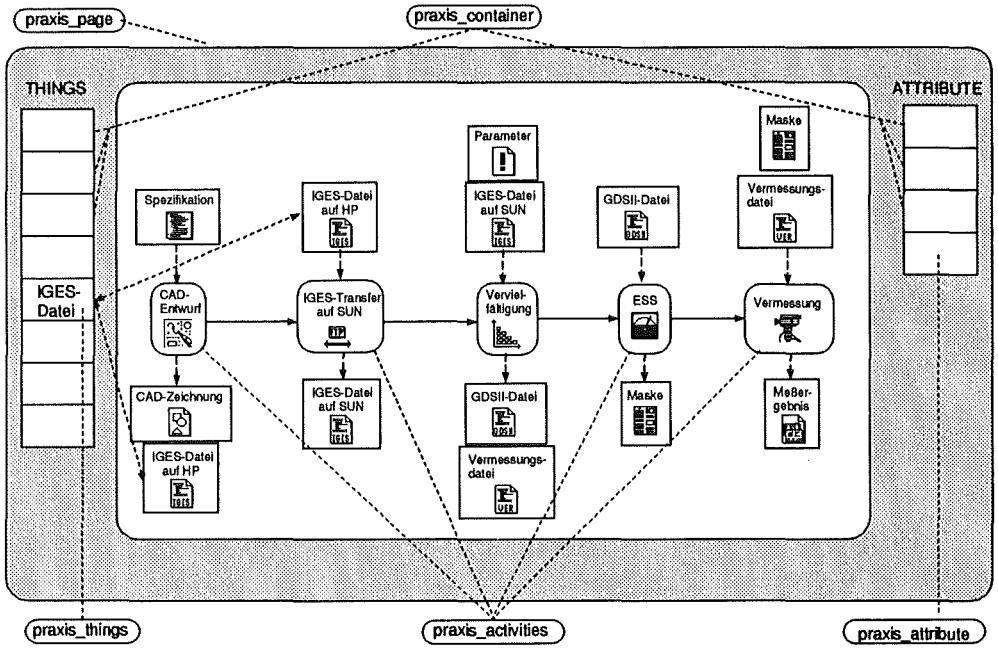


Abbildung 7.2: Aufbau einer *praxis_page*

Die Verteilung von *praxis_activities* auf *praxis_pages* übernimmt der Benutzer (Prozeßexperte). Er hat damit Einfluß auf Strukturierung der Aktivitäten, auf die Zusammenfassung von Aktivitäten auf Seiten und die Verfeinerung von Aktivitäten auf separaten Seiten, falls die Darstellung auf einer Seite zu komplex wird.

- ❒ Eine *praxis_pages* enthält eine Liste der Aktivitäten, die auf der Seite dargestellt werden (siehe Abbildung 7.3).

- ❍ Als Operationen muß der Benutzer *praxis_activities* auf einer *praxis_page* hinzufügen und löschen können (siehe Abbildung 7.3).

Für die Aktivitäten besteht bei der Visualisierung auf Seiten eine eindeutige Abbildung, d.h. eine Aktivität kann nie auf mehreren Seiten innerhalb des gleichen Diagramms auftreten.

❒ Neben den *praxis_activities* enthält jede *praxis_page* noch (siehe Abbildung 7.3):

- Container (siehe Kapitel 7.3) für alle Ein-/Ausgabeobjekte (*praxis_things*) der Aktivitäten dieser *praxis_page*
- Container für die Attributwerte (*praxis_attributes*) für alle Attributwerte der Aktivitäten dieser *praxis_page*
- Container für die Parameterwerte (*praxis_parameters*) der Aktivitäten dieser *praxis_page*

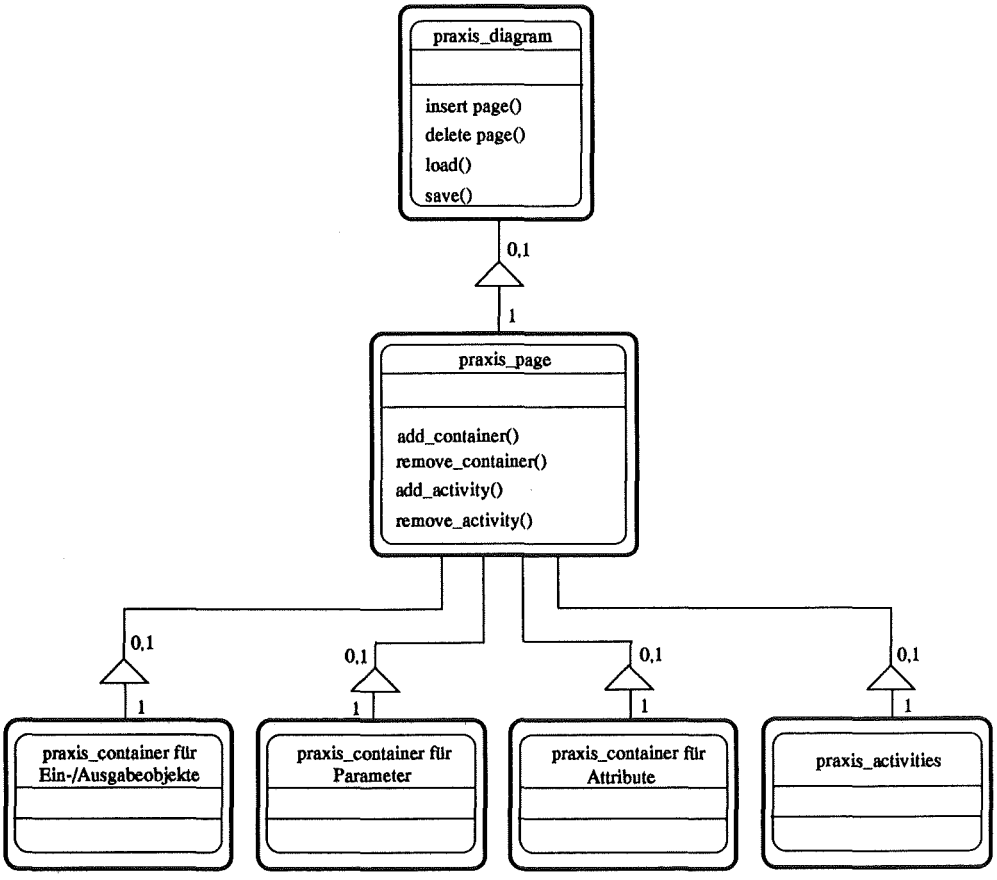


Abbildung 7.3: Objektorientierter Entwurf: Teil 1

❍ Es ist möglich, auf *praxis_pages* beliebige Container zu definieren und zu löschen (siehe Abbildung 7.3). Die Definition dieser Container übernimmt ebenfalls der Benutzer (Prozeßexperte).

7.3 Die Klasse *praxis_container*

Container (*praxis_container*) sind Sammelbehälter für Objekte gleichen Typs. Sie werden notwendig, weil es die Möglichkeit gibt, *praxis_activities* wiederholt auszuführen, und da a priori nur festlegbar ist, was für Typen die Ein- bzw. Ausgabeobjekte haben, aber nicht welche tatsächlichen Ausprägungen.

So wird z.B. bei der Durchführung einer Aktivität *Vermessung* eine Datei mit *Meßergebnissen* erzeugt, und zwar bei jeder Vermessung eine neue, unabhängig davon, ob die Vermessung auf identischen Eingabedaten und Attributwerten abläuft oder nicht. Demnach muß beim Erstellen einer Seite ein Container für die Aufnahme von Meßergebnisdateien definiert werden (siehe Abbildung 7.4).

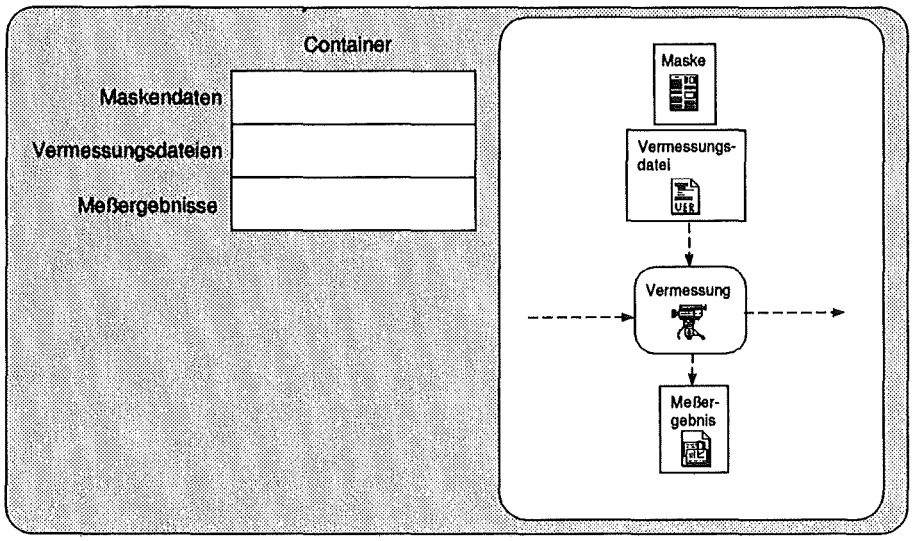


Abbildung 7.4: Definition der *praxis_container*

Später wird sichergestellt, daß bei jeder Durchführung der Aktivität ein neues Objekt *Datei mit Meßergebnissen* in diesen Container geschrieben wird (siehe Abbildung 7.7).

- ❒ Container haben einen Namen zur Identifikation und einen Basistyp, in dem der Datentyp der in ihm enthaltenen Objekte (z.B. IGES-Datei) festgelegt wird (siehe Abbildung 7.5).

Es ist auch denkbar, daß ein Container selbst überhaupt keine Objekte enthält, sondern auf einen Container verweist, der in der Seitenhierarchie übergeordnet ist.

Das bedeutet, daß dieser Container seine Objekte aus solchen sogenannten globalen Container liest bzw. in diesen globalen Container schreibt.

- ❒ Daher hat jeder *praxis_container* auch eine lokale oder globale Gültigkeit.

Für die Verarbeitung muß ein Container alle Untercontainer kennen, die auf ihn verweisen, falls er selbst global ist. Ebenso kennt ein Container den Container, auf den er gegebenenfalls verweist (siehe Abbildung 7.5).

- Es muß die Möglichkeit geben, Objekte in den Container einzufügen und zu löschen, sowie Information über Objekte bereitstellen, die sich im Container befinden (siehe Abbildung 7.5).

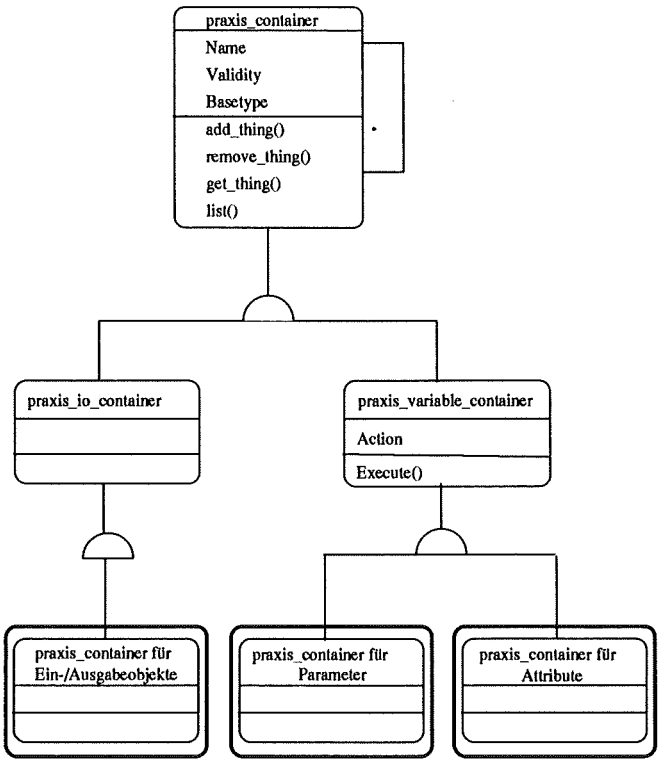


Abbildung 7.5: Objektorientierter Entwurf: Teil 2

Wie Abbildung 7.5 zeigt, werden *praxis_container* weiter in die beiden Klassen *praxis_variable_container* und *praxis_io_container* unterteilt, wobei sich diese beiden Klassen folgendermaßen unterscheiden:

- *praxis_io_container* sind Container für *praxis_things*, d.h. sie enthalten *Verweise* auf eigenständige Objekte in der Datenbasis.
- *praxis_variable_container* sind Container für *praxis_variables*, d.h. sie enthalten *reale Objekte* (z.B. *praxis_attributes* oder *praxis_parameters*). Zusätzlich zu den oben dargestellten Informationen von *praxis_container* besitzen *praxis_variable_container* noch ein Action-Field, in das z.B. der Aufruf eines externen Tools zur Errechnung eines *praxis_parameter*-Wertes eingetragen wird, bzw. der Aufruf einer Maske zur Eingabe des

praxis_parameter- Wertes (wie das auch bei *praxis_attributes* der Fall ist).

7.4 Die Klasse *praxis_activity*

Eine Aktivität ist eine Einheit des Produktionsablaufs. Sie kann entweder weiter verfeinert werden (*praxis_hierarchical_activity*) oder ausgeführt werden (*praxis_executable_activity*).

Die Verfeinerung einer verfeinerbaren *praxis_activity* (z.B. Aktivität A_2 aus Abbildung 7.1) wird auf einer separaten *praxis_page* dargestellt (unterste *praxis_page* in Abbildung 7.1).

- ❒ Eine *praxis_activity* enthält neben der Aktion, die bei der Ausführung gestartet wird (z.B. das Starten eines Werkzeugs oder das Aufblenden der Verfeinerungsseite) auch Information darüber, welcher Personenkreis für die Ausführung der *praxis_activity* vorgesehen ist, wann der geplante Zeitpunkt der Ausführung ist, sowie welche Werkzeuge für die Ausführung einer *praxis_activity* vorgesehen sind (siehe Abbildung 7.6).

- ❒ Für die Visualisierung der kausalen Vorgänger-/Nachfolgerbeziehung muß eine *praxis_activity* die *praxis_activity* kennen, die vor bzw. nach ihr ausgeführt werden. Jede *praxis_activity* hält daher eine Liste der Vorgänger- und Nachfolgeraktivitäten (siehe Abbildung 7.6).

- ○ Über die Listen der Vorgänger- und Nachfolgeraktivitäten sind die kausalen Vorgänger-/Nachfolgerbeziehung veränderbar (siehe Abbildung 7.6).

Bisher wurde bei den *praxis_pages* nur *praxis_containers* eingeführt, ohne diese mit den *praxis_activities* auf den *praxis_pages* in irgendeiner Weise zu verknüpfen. Die Zuordnung, welche *praxis_containers* zu einer *praxis_activity* gehören, und ob als Eingabe- oder Ausgabeobjekte, wird vom Benutzer (Prozeßexperte) vorgenommen.

Die gestrichelten Pfeile in Abbildung 7.2 deuten diese Zuordnung der Container zu den *praxis_activities* an. So hat beispielsweise die Aktivität *CAD-Entwurf* als Ausgabeobjekt eine *IGES-Datei*, die von der Aktivität *IGES-Transfer* als Eingabeobjekt verwendet wird.

- ❒ Eine *praxis_activity* enthält Referenzen (siehe Abbildung 7.6) auf
  - die *praxis_containers* ihrer Ein- und Ausgabeobjekte,
  - die *praxis_containers* ihrer Attribut- und Parameterwerte.

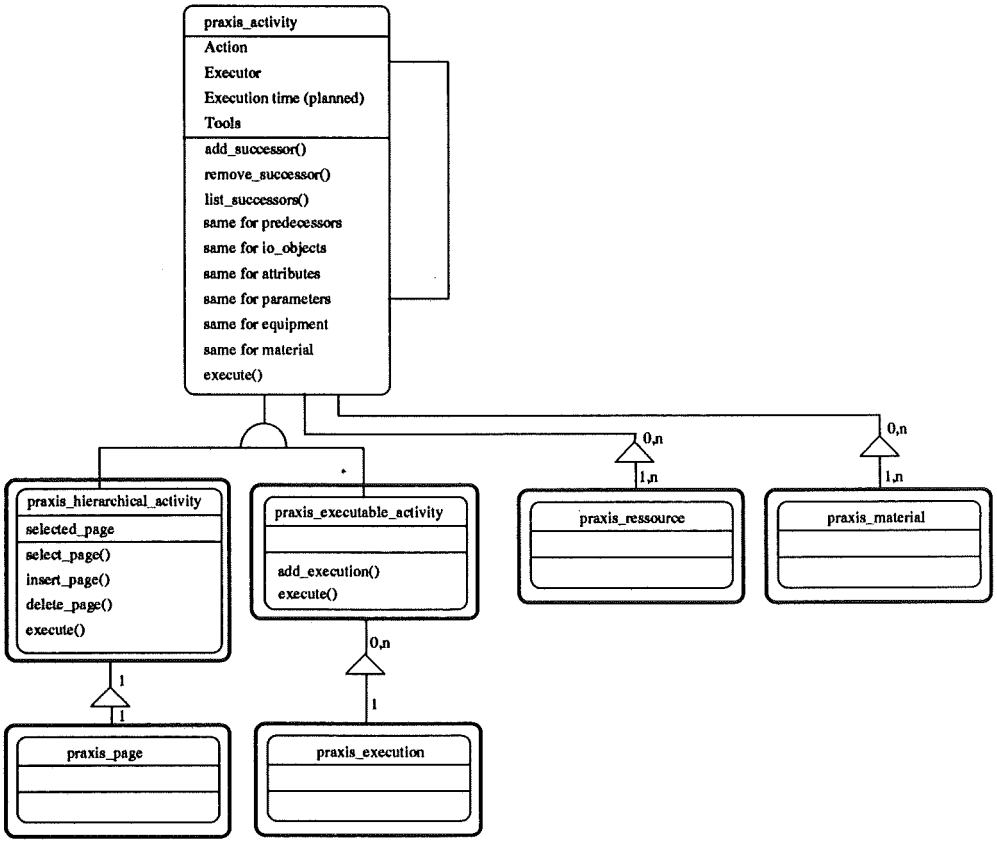


Abbildung 7.6: Objektorientierter Entwurf: Teil 3

Die Container werden nur referenziert, da ein *praxis_ container* Ausgabeobjekt einer *praxis_ activity* und gleichzeitig Eingabeobjekt einer anderen *praxis_ activity* enthalten kann.

❒ Zusätzlich kann man für jede *praxis_ activity* noch die Ressourcen (Ausrüstungsgegenständen) und Materialien festlegen, die für eine Ausführung potentiell möglich sind (siehe Abbildung 7.6). Welche Ressourcen und Materialien dann bei der Ausführung tatsächlich verwendet werden, wird erst bei der *praxis_ execution* gespeichert.

❍ Als Operationen, die eine *praxis_ activity* ausführen kann, sei

- das Ändern des Prozeßablaufs durch Änderung (Hinzufügen oder Löschen) der Vorgänger- bzw. Nachfolgerbeziehungen,
- das Ändern der Zuordnung von Eingabe, Ausgabe-, Parameter-, Attributwert-, Material- und Ressourcencontainer, sowie
- das Ausführen einer *praxis_ activity*

genannt (siehe Abbildung 7.6).

7.5 Die Klasse *praxis_executable_activity*

Die ausführbare *praxis_activity* kann mehrmals ausgeführt werden. Dabei können Eingabe-, Ausgabe-, Attribut- und Parameterobjekte jeweils unterschiedliche Ausprägungen besitzen (siehe Abbildung 7.7).

Mehrmaliges Ausführen einer *praxis_activity* führt zu mehreren *praxis_executions* (siehe Execution 1 und 2 in Abbildung 7.7). Dabei referenziert jede *praxis_execution* auf ein Objekt des entsprechenden *praxis_container*.

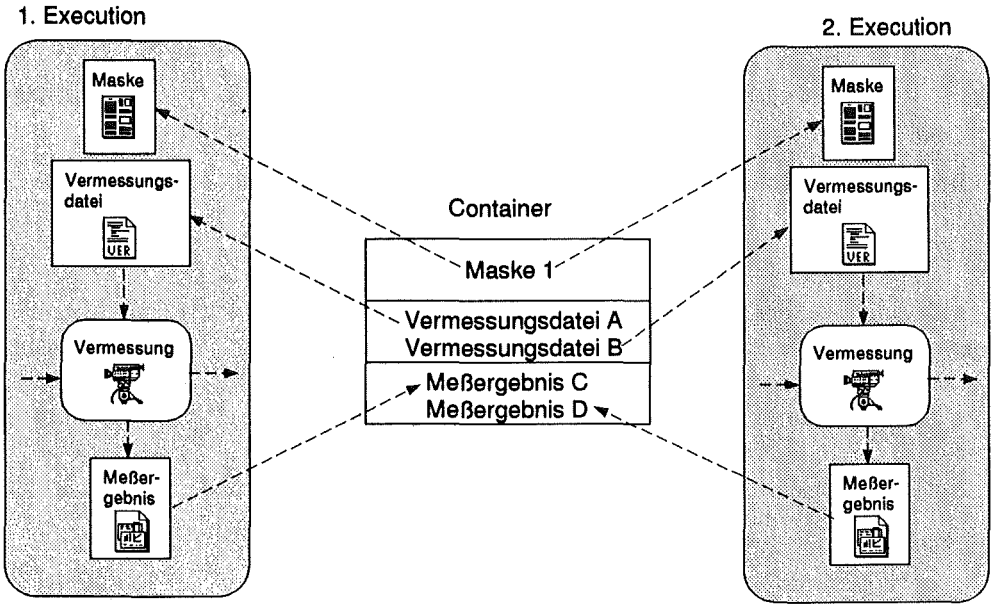


Abbildung 7.7: Mehrmaliges Ausführen einer *praxis_activity*

❒ Die *praxis_activity* führt eine Liste ihrer Ausführungen (*praxis_executions*) (siehe Abbildung 7.6).

○ Diese Liste muß erweiterbar sein. Eine Erweiterung dieser Liste durch Hinzufügen einer *praxis_execution* wird automatisch beim Ausführen einer *praxis_executable_activity* erzeugt (siehe Abbildung 7.6).

7.6 Die Klasse *praxis_hierarchical_activity*

Eine *praxis_hierarchical_activity* ist eine Aktivität, die weiter verfeinert werden kann..

❒ Daher enthält die *praxis_hierarchical_activity* alle möglichen Alternativ-(kind)seiten. Zu einer Zeit ist nur die Darstellung genau einer dieser Alter-

nativ(kind)seiten möglich. Somit enthält eine *praxis_ hierarchical_ activity* auch Information über die zur Darstellung aktuell ausgewählten Seite (siehe Abbildung 7.6).

- Es muß möglich sein, neue Alternativ(kind)seiten hinzuzufügen und alte zu löschen. Ebenso muß man die zur Darstellung gewünschte Seite anwählen und darstellen können. Die Ausführung einer *praxis_ hierarchical_ activity* ist identisch mit der Visualisierung der gewünschten Alternativ(kind)seite (siehe Abbildung 7.6).

7.7 Die Klasse *praxis_ execution*

Eine *praxis_ execution* entspricht der Ausführung einer *praxis_ executable_ activity.*

- Eine *praxis_ execution* ist gekennzeichnet durch einen Ausführenden (evtl. eine Personengruppe), der zu einem bestimmten Zeitpunkt, unter Zuhilfenahme eines bestimmten Werkzeugs die *praxis_ executable_ activity* ausführt (siehe Abbildung 7.8).

- Die Ausführung einer *praxis_ activity* benötigt benötigt für jeden Objekttyp (Eingabe-, Ausgabe-, Attribut- und Parameterobjekte) die für die Ausführung relevanten realen Objekte. Für jeden Objekttyp wird genau ein reales Objekt gehalten (siehe Abbildung 7.8).

Wenn beispielsweise ein Eingabeobjekt einer *praxis_ activity* vom Typ *Maske* ist, so wird an dieser Stelle die Instanz verwaltet, die bei einer speziellen *praxis_ execution* dieser *praxis_ activity* tatsächlich benutzt wurde (z.B. *Maske 1* in Abbildung 7.7).

- Ebenso werden hier die tatsächlich verwendeten Ressourcen und Materialien benötigt. All diese Information wird bei der entsprechenden *praxis_ execution* gehalten (siehe Abbildung 7.8).

7.8 Die Klasse *praxis_ things*

Die Klasse *praxis_ things* ist eine Oberklasse aller in der Datenbank gehaltenen Objekte im Zusammenhang mit Aktivitäten. Die Objekte der Klasse *praxis_ things* sind weiter unterteilbar in Ein- und Ausgabeobjekte (*praxis_ things*) und Variablen (*praxis_ variables*).

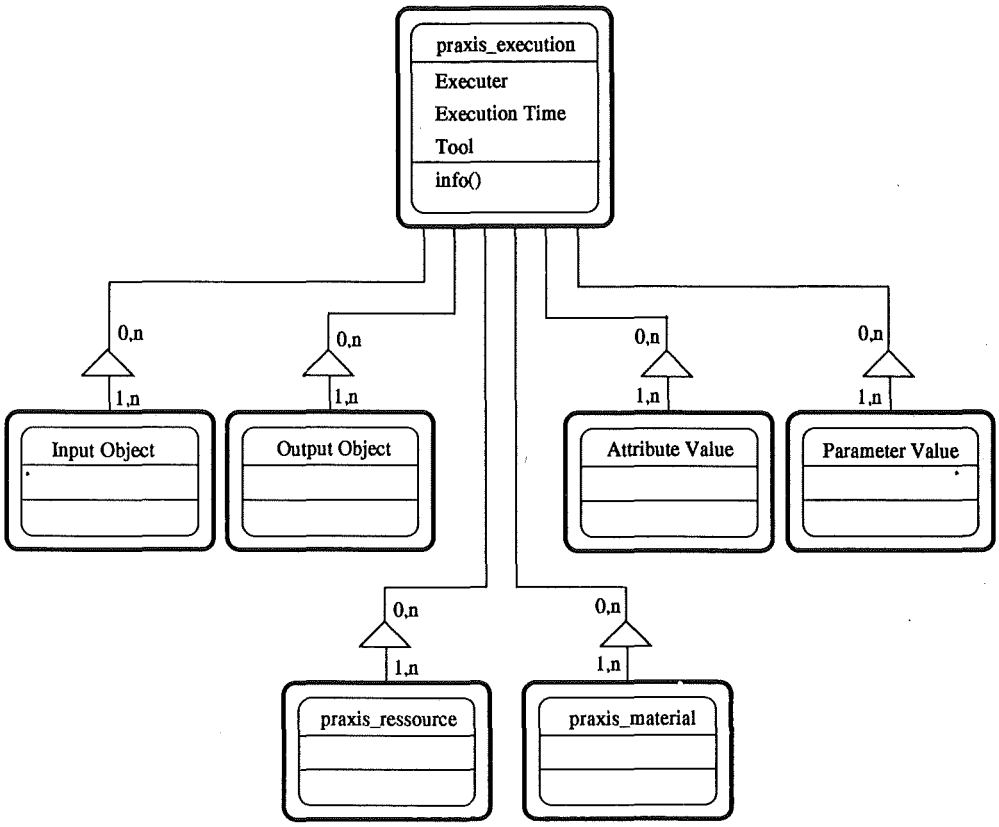


Abbildung 7.8: Objektorientierter Entwurf: Teil 4

Die Ein-/Ausgabeobjekte sind spezielle Objekte, die sehr komplexer Struktur sein können. Denkbar ist beispielsweise eine Geometrie, die durch die Beschreibung im Geometriemodell GEM aus Kapitel 5.1.3 repräsentiert wird. Die Art dieser Modelle richtet sich nach der Vielzahl der zu speichernden Objekte.

Durch die Flexibilität, die eine objektorientierte Implementierung ermöglicht, ist es leicht möglich, jeweils neue Strukturen für neue Objekte auch im Laufe der Systembenutzung hinzuzufügen

- ❒ *praxis_things* haben neben einem Namen als Bezeichner (siehe Abbildung 7.9).

- ○ Daneben haben *praxis_things* eine Methode zur Darstellung des Objekts (z.B. die Darstellung einer Geometriebeschreibung mit Hilfe eines entsprechenden Werkzeugs) und zum Ändern des Objekts (siehe Abbildung 7.9).

praxis_variables kann man weiter in Attribute (*praxis_attributes*) und Parameter (*praxis_parameters*) unterteilen. Variablen haben, wie bereits angesprochen, einen einfach strukturierten Wert. Sie werden meist in Gruppen zusammengefaßt in einer speziellen Maske zur Verfügung gestellt.

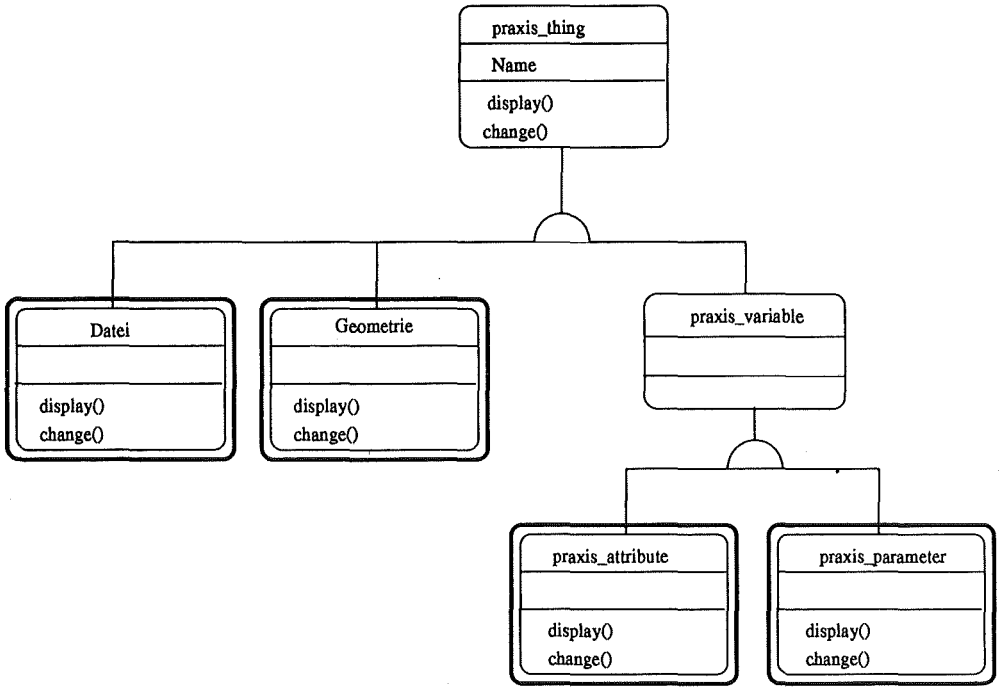


Abbildung 7.9: Objektorientierter Entwurf: Teil 5

7.9 Die Klassen *praxis_material* und *praxis_ressource*

Die beiden Klassen haben rein informellen Charakter bezüglich der vorhandenen Materialien und der verfügbaren Ressourcen. Bereits früher wurde die Notwendigkeit diskutiert, an diese beiden Klassen Parameter zu binden. Beispielsweise ist die Materialdicke eindeutig ein Materialparameter, die Ofentemperatur ein Parameter der Ressource Ofen. Allerdings sind die Ausprägungen dieser Parameter stark mit den Aktivitäten korreliert, welche entweder Materialien oder Ressourcen benötigen. So interessiert es nicht, welche Temperaturen ein bestimmter Ofen bisher seit Inbetriebnahme irgendwann einmal angenommen hat. Welche Ofentemperatur er als Ressource einer bestimmten Aktivität hatte, ist sehr wohl relevant. Demnach wird bei Materialien und Ressourcen zwar die Liste der jeweils zugehörigen Parameter gehalten, bei der Handhabung unterscheiden sich materialabhängige Parameter und ressourcenabhängige Parameter jedoch nicht mehr von aktivitätsabhängigen.

7.10 Die Klasse *praxis_object*

Schließlich wird mit *praxis_object* noch eine Klasse eingeführt, von der alle eigenständigen Klassen mit persistenten Daten in irgendeiner Weise, d.h. eventuell über mehrere Zwischenstufen, abgeleitet werden, wie in Abbildung 7.10 dargestellt.

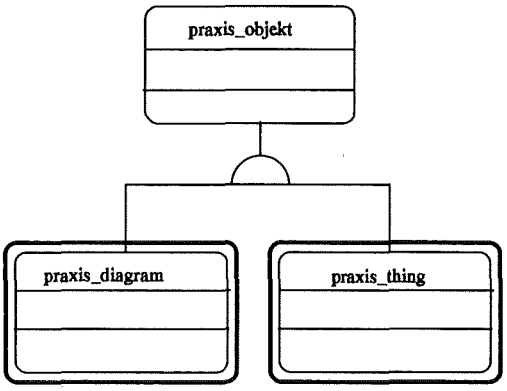


Abbildung 7.10: Objektorientierter Entwurf: Teil 6

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

Das LIGA-Verfahren ist ein am Forschungszentrum Karlsruhe entwickeltes Verfahren zur Herstellung von Mikrostrukturen. Es wird heute am Institut für Mikrostrukturtechnik eingesetzt, um Mikrostrukturen herzustellen und dabei ständig erweitert und verbessert.

Um das LIGA-Verfahren mit seinen Einzelprozessen besser überschauen zu können, ist es entscheidend, ein System zur rechnergestützten Dokumentation des LIGA-Verfahrens zu realisieren. Dazu gehört die *Erfassung aller Prozeßschritte* des LIGA-Verfahrens mit *allen Parametern, allen Materialdaten* sowie einer *Beschreibung der Ressourcen* (Ausrüstungsgegenstände). Für diese rechnergestützte Dokumentation ist ein Modell zu entwickeln, um das Prozeßwissen zu handhaben.

Auch die Erfahrungen, die aus der Produktion von Mikrostrukturen gewonnen werden, stellen Fertigungswissen dar. Daher ist es notwendig, das Modell so zu konzipieren, daß auch die Sammlung von Information über *sämtliche* in der Vergangenheit *gefertigten Produkte* möglich ist.

In der vorliegenden Arbeit werden Möglichkeiten aufgezeigt, ein System zur Erfassung und Weiterverarbeitung von Produktinformation bei der Herstellung von Mikrostrukturen (PRoduct And EXperimentation Information System, kurz *PRAXIS*) zu modellieren. Der Schwerpunkt der Arbeit liegt dabei auf der Integration unterschiedlichster Komponenten und Systeme aus verschiedensten Bereichen.

Für die Sammlung und Weiterverarbeitung von Produktinformation steht eine breite Palette an informationsverarbeitenden Werkzeugen zur Verfügung. Diese Palette wird ständig erweitert. Das modellierte System muß demnach die Möglichkeit bieten, *beliebige Werkzeuge* einzubinden und sie auf einem *integrierten*

Datenbestand arbeiten zu lassen.

Die vorliegende Arbeit untersucht zunächst die Anforderungen an ein solches System aus folgenden drei Sichten:

- aus der Prozeßsicht, d.h. dem LIGA-Fertigungsprozeß aus ganzheitlicher Sicht mit allen für den Prozeß möglichen Fertigungsalternativen,
- aus der Produktsicht, d.h. mit aller Information, die bei der Produktion eines ganz bestimmten Produktes anfällt,
- sowie aus der Produktionssicht, die die vorhandenen Produktsichten mit dem Ziel des Projektmanagements koordiniert.

Je nach Systemsicht gibt es Möglichkeiten zur

- Hierarchiebildung der an der Produktion beteiligten Aktivitäten.
- Darstellung von kausalen Vorgänger- /Nachfolgerbeziehung der an der Produktion beteiligten Aktivitäten.
- Darstellung von Alternativaktivitäten in der Prozeßsicht.
- Darstellung von Nebenläufigkeiten der an der Produktion beteiligten Aktivitäten.
- Darstellung von Zuständen und Wiederholungen von Aktivitäten.
- Zuordnung von Ein- und Ausgabeobjekten der an der Produktion beteiligten Aktivitäten.
- Handhabung von Versionen der Produktdaten in der Produktsicht.
- Darstellung von Zuständen der Produkte.
- Zusammengefaßten Darstellung der sich in der Produktion befindlichen Produkte.

Im Anschluß an die Definition der Anforderungen bezüglich der Systemsichten, werden Alternativen zur Anbindung von Werkzeugen und zur Integration der Daten sowie zur Verwaltung der Daten durch ein objektorientiertes Datenbanksystem vorgestellt.

Diese Anforderungen und Konzepte führten zu einer Architektur von *PRAXIS*, die aus drei Komponenten besteht:

- Einem Werkzeug zur Visualisierung der Systemsichten.
- Einer erweiterbaren Sammlung von Werkzeugen.
- Einem objektorientierten Datenbanksystem zur Verwaltung des integrierten Datenbestandes.

Anhand der beiden Bereiche *Entwurfsschritte* und *Fertigungsschritte* von Mikrosystemen wurden Lösungen für Partialmodelle erarbeitet, vorgestellt und teilweise implementiert. Diese Partialmodelle mit den dazugehörenden Prozessoren sind Ergebnis von Arbeiten, die am Institut für Angewandte Informatik (IAI) in Zusammenarbeit mit dem Institut für Mikrostrukturtechnik (IMT) durchgeführt wurden [SES93b, ES93a, ES93b]. Die dort dargelegte Komplexität zeigt, daß die Sammlung und Weiterverarbeitung von Produktinformation ohne die entsprechenden Systemsichten, ohne geeignete rechnergestützte Werkzeuge und ohne einen integrierten Datenbestand nicht möglich ist. Daher wird zum Abschluß der Arbeit mit *PRAXIS* ein System zur Sammlung und Weiterverarbeitung von Produktinformation objektorientiert ansatzweise modelliert, das den gestellten Anforderungen gerecht wird.

8.2 Ausblick

Nachdem die Anforderungen an ein System zur Erfassung und Weiterverarbeitung von Produktinformation bei der Herstellung von LIGA-Mikrostrukturen formuliert sind, und das System *PRAXIS* objektorientiert modelliert wurde, müssen die nächsten Schritte erfolgen:

Man hat einerseits die Möglichkeit, für die einzelnen Komponenten der Architektur ein kommerzielles System zu finden, das die Anforderungen erfüllt. Untersuchungen in dieser Richtung wurden bereits mit der Evaluierung eines Frameworks [WBCS93] und objektorientierter Datenbanksysteme unternommen.

Wenn vorhanden Werkzeuge die hier definierten Anforderungen des System nicht oder nur teilweise erfüllen, kann man andererseits die entsprechenden Komponenten auch selbst implementieren. Dazu gehört die

- Implementierung eines Werkzeuges zur Visualisierung (und gegebenenfalls der Bearbeitung) der gewünschten Systemsicht,
- Implementierung einer Komponente, die die Einbettung beliebiger Werkzeuge ermöglicht,
- sowie die Implementierung der benötigten Partialmodelle für die integrierte Datenhaltung.

Aspekte dieser Komponenten sind in der Beschreibung der Partialmodelle in Kapitel 5 sowie in der objektorientierten Modellierung des Systems *PRAXIS* in Kapitel 7 enthalten.

8.2.1 Evaluierung verfügbarer Werkzeuge

Das von der Firma Siemens/Nixdorf zur Verfügung gestellte Framework *Siframe* (Version 2.0/2.1) wurde verwendet, um Aussagen über die Eignung des

Frameworks im Hinblick auf zuvor herausgearbeitete Randbedingungen für die Verwendung innerhalb von *PRAXIS* machen zu können.

Dafür wurde ein kleiner Ausschnitt der Dokumentenbearbeitung exemplarisch modelliert.

Die Umgebung *Siframe* bietet eine Palette an globalen Ressourcen an, die für die Definition eines Frameworks mit konkreten Objekten instantiiert werden können. In *Siframe* stehen die in Abbildung 8.1 gezeigten Ressourcen zur Verfügung:

- Werkzeuge.
- Anwender.
- Aktivitäten, d.h. einzelne Funktionalitäten von Werkzeugen.
- Teams, d.h. Anwendergruppen.
- Viewtypes, d.h. Austauschformate zwischen Aktivitäten.
- Flüße, d.h. Aktivitätssequenzen.
- Rollen, die der Anwender bei der Benutzung des Frameworks einnimmt.

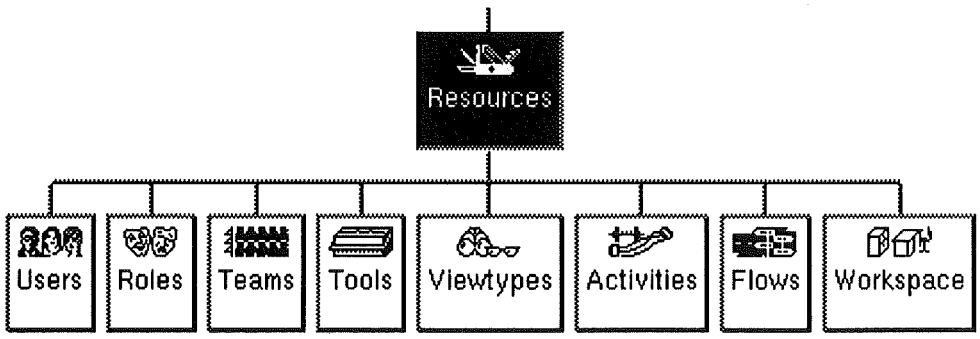


Abbildung 8.1: Die möglichen Ressourcen unter *Siframe*

Die Definition von konkreten Objekten erfolgt interaktiv. Nachdem Anwender erzeugt, ihnen bestimmte Rollen zugewiesen, sowie Anwender zu Gruppen zusammengefaßt worden sind, werden Werkzeuge in das Framework integriert.

Die Funktionalität von Werkzeugen wird anschließend in Aktivitäten als Untereinheiten unterteilt. Sie rufen ein Werkzeug mit zu spezifizierenden Parametern auf und definieren Ein- und Ausgabe des Werkzeuges über Viewtypes.

Diese Aktivitäten werden abschließend zu Flüssen verknüpft, die die Ablaufsequenz der Aktivitäten festlegen.

Bei der Realisierung der Fallstudie wurden folgende Eigenschaften von *Siframe* festgestellt:

- **Vorhandene Ressourcen ausreichend**

  Die von *Siframe* zur Verfügung gestellten Ressourcen sind für die Modellierung der Abläufe auch größerer Projekte ausreichend. Wenn allerdings die Anzahl der definierten Objekte als Instanzen der Ressourcen zunimmt, kann die Darstellung recht unübersichtlich werden. Angenehm ist dabei, daß sich der Anwender auf die Darstellung der für ihn interessanten Teilbäume beschränken kann.

- **Zerlegung komplexer Projekte in Teilprojekte möglich**

  Die Schritte eines Projekts lassen sich in Unterschritte zerlegen. Projektschritte lassen sich in unterschiedlichen Hierarchieebenen darstellen, was die Möglichkeit schafft, sehr komplexe Projekte mit einer Vielzahl von Arbeitsschritten übersichtlich zu gliedern.

- **Aktueller Stand des Projekts leicht ablesbar**

  Die einzelnen Aktivitäten, die zu einem Aktivitätsfluß verknüpft werden, können bestimmte Zutände wie *nicht ausgeführt, in Benutzung, ausgeführt, beendet* oder *ungültig* annehmen. Dadurch kann man den aktuellen Stand des Projekts anhand des Status seiner Aktivitäten sofort ablesen.

- **Überwachung von Parallelzugriff realisiert**

  Bei einem Projekt größerer Dimension arbeiten oft viele Benutzer gleichzeitig auf einem gemeinsamen Datenpool. Daher ist es wichtig, ein System zu haben, das den parallelen Zugriff auf Daten in irgendeiner Weise sequentialisiert (concurrency control). *Siframe* ermöglicht dem Anwender durch das Reservieren von Objekten im persönlichen Arbeitsbereich, diese vor dem Zugriff durch andere Anwender zu sperren. Erst wenn dieser Anwender nach durchgeführten Änderungen das Objekt wieder freigibt, kann der nächste Anwender es zur weiteren Manipulation für sich reservieren. Er setzt dabei automatisch auf dem neuesten Stand auf.

- **Ausschließlich Black-Box-Integration möglich**

  Zur Zeit ist nur eine Black-Box-Integration von Werkzeugen in das Framework möglich, d.h. Daten, die von den Werkzeugen gelesen, modifiziert und geschrieben werden, sind nur als Dateien zwischen den Werkzeugen austauschbar. Die Nachteile dieser Integrationsalternative sind in Kapitel 3.3.2 hinreichend aufgeführt.

- **Datenbankschema nicht änderbar**

  Das Framework, das unter Zuhilfenahme von *Siframe* modelliert wird, wird im sogenannten Object Management System (OMS) abgelegt. Dort werden auch die Dateien gespeichert, die von Werkzeugen innerhalb des Frameworks ausgetauscht werden. Das Datenschema ist dabei festgelegt und kann vom Anwender nicht geändert werden. Er hat also keine Möglichkeit, benutzerdefinierte Datenstrukturen einzubringen, auf denen er Manipulationen von Daten ausführen kann. Somit handelt es sich bei dem

Object Management System nicht um eine objektorientierte Datenbank im klassischen Sinne, nach den Anforderungen in [ABD 89].

- **Objektorientierte Datenbank als Zusatzwerkzeug erforderlich**

  Wenn – wie bei einer White-Box-Integration üblich – Datenstrukturen definiert werden sollen und die Daten dort auch manipuliert werden, dann muß zusätzlich zu dem Framework eine objektorientierte Datenbank angebunden werden. Dazu muß ein Datenbanksystem installiert werden. Darauf aufsetzend werden Anwendungsprogramme implementiert, die auf die objektorientierte Datenbank zugreifen. Diese Anwendungsprogramme können dann vom Framework aus als Werkzeug aufgerufen werden, das Daten innerhalb des Datenbankschemas manipuliert.

  Die parallele Nutzung der Datenbank von *Siframe* für die Speicherung von Framework-Objekt-Daten (Details über Werkzeuge, Anwender, Aktivitäten, usw.) sowie vom Anwender (für seine eigenen Daten innerhalb selbst zu definierenden Datenstrukturen) ist *nicht* möglich.

- **Aufbau des Framework ausschließlich interaktiv**

  Es gibt keine Möglichkeit, das Framework automatisch z.B. durch eine Sprache aufzubauen. Die Definition von konkreten Objekten erfolgt interaktiv.

  Wünschenswert wäre eine Sprachschnittstelle zum Benutzer mit der Möglichkeit, das Framework im Batchbetrieb aufzubauen.

- **Schwierigkeiten, nachträglich Änderungen durchzuführen**

  Sind bei einer Umgebung Datenflüsse definiert, in denen Aktivitäten sequenzialisiert werden, so ist es nicht möglich, die Aktivitäten zu modifizieren. Stellt man also bei der Definition des Datenflusses fest, das eine Aktivität mit den falschen Parametern aufgerufen wird, so muß man *alle* Datenflüsse löschen, die diese Aktivität ausführen, die Aktivität dann modifizieren, um den Datenfluß anschließend erneut zu definieren.

- **Keine parallele Arbeit *eines* Anwenders möglich**

  Aufgefallen ist, daß jeder Anwender sich nur einmal im System anmelden kann, ein zweites Anmelden wird verwehrt. Insbesondere beim parallelen Arbeiten an mehreren Projekten von verschiedenen Rechnern aus kann dies aber von Bedeutung sein.

Von den Komponenten von *PRAXIS* (siehe Abbildung 6.1), erfüllt *Siframe* die Anforderungen wie folgt:

- *Siframe* übernimmt vorwiegend die Aufgaben der obersten Ebene, der Visualisierung der Sichten. Diese können unter Zuhilfenahme von *Siframe* modelliert werden. Die dabei unterstützen Konzepte in Form von unterschiedlichen Ressourcen sind dabei sehr hilfreich, auch im Hinblick auf

die Benutzergruppen der Umgebung mit ihren unterschiedlichen Kenntnisständen und Zugriffsrechten. Allerdings lassen sich keine Sichten automatisch generieren, die das z.B. bei der Produktionssicht notwendig ist.

- Die Komponente der **erweiterbaren Sammlung von Werkzeugen** wird ebenfalls von *Siframe* realisiert, allerdings ausschließlich für eine Black-Box-Integration auf der Ebene von Dateien, die zwischen den integrierten Werkzeugen ausgetauscht werden, nicht jedoch für White-Box-Integration auf der Ebene von Datenstrukturen.

- Es werden zwar die in *Siframe* modellierten Objekte **objektorientiert** abgelegt, doch sind diese Daten für den Anwender nicht direkt zugreifbar, d.h. er kann insbesondere die Datenstrukturen der Datenbank nicht definieren und manipulieren. Für die Daten, die von *Siframe* zur Modellierung des Frameworks verwendet werden, macht dies Sinn, für eine White-Box-Integration jedoch ist dies eine unumgängliche Anforderung an eine verwendete Datenbank.

  Das Datenmodell zerfällt in 2 Teile: auf der einen Seite die Sichten (gehalten im *Siframe* -eigenen, Object Management System) auf der anderen Seite das Modell der von den Werkzeugen verwendeten und manipulierten Daten. Dieses muß – da die *Siframe* -Datenbank nicht von außen zugänglich ist – in einer eigenen, objektorientierten Datenbank untergebracht werden. Anwendungen, die auf dieser Datenbank aufsetzen, können anschließend als Werkzeuge (mit Dummy-Viewtypes) in *Siframe* integriert werden. Auf diese Weise lassen sich objektorientierte Datenstrukturen in einer objektorientierten Datenbank speichern und mit Hilfe von *Siframe* manipulieren, d.h. auch White-Box-Integration von Werkzeugen durchführen.

Das Framework *Siframe* konnte demnach den Anforderungen, die in dieser Arbeit herausgearbeitet wurden, nur *teilweise* gerecht werden. Insbesondere für die Anbindung an eine objektorientierte Datenbank muß noch einige konzeptionelle Arbeit geleistet werden. Ob die Versionenverwaltung, die in zukünftigen Versionen von *Siframe* enthalten sein soll, sich mit der Versionenverwaltung eines objektorientierten Datenbanksystems koppeln läßt (was speziell für die Entwicklung von Mikrosystemen bedeutsam ist), kann erst geprüft werden, wenn dies tatsächlich in *Siframe* realisiert ist.

Ein objektorientiertes Datenbanksystem als unterste Systemkomponenten wurde bereits für die prototypische Implementierung zum Partialmodell der Fertigungssicht verwendet. Entsprechende Anmerkungen dazu finden sich in Anhang B.

Literaturverzeichnis

[AASW94] ABRAMOWICZ, K. ; ALT, J. ; SCHREIBER, H. ; WALLRATH, M.: *Evaluierung objektorientierter Datenbanksysteme.* Forschungszentrum Informatik an der Universität Karlsruhe, 1994

[ABD89] ATKINSON, M. ; BANCILHON, F. ; DEWITT, D. ; DITTRICH, K. ; MAIER, D.: The object-oriented database system Manifesto. **In:** KIM, W. (Hrsg.) ; NICOLAS, J.-M. (Hrsg.) ; NISHIO, S. (Hrsg.): *Proc. 1st International Conference on Deductive and Object-Oriented Databases* North-Holland, 1989, S. 40–57

[And93a] ANDERL, R.: *CAD-Schnittstellen. Methoden und Werkzeuge zur CA-Integration.* Carl Hanser Verlag, 1993

[And93b] ANDERL, R.: STEP – Grundlagen der Produktmodelltechnik. **In:** STUCKY, W. (Hrsg.) ; OBERWEIS, A. (Hrsg.): *Datenbanksysteme in Büro, Technik und Wissenschaft* BTW93, 1993

[ASU86] AHO, A. ; SETHI, R. ; ULLMAN, J.D.: *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986

[AWSL92] AHMED, S. ; WONG, A. ; SRIRAM, D. ; LOGCHER, R.: Object-Oriented Database Management Systems for Engineering: A Comparison. **In:** *Journal of Object-Oriented Programming* 5 (1992), June, Nr. 3, S. 27–44

[Bar93] Barry and Associates. Burnsville, Minnesota, USA: *DBMS Needs Assessment for Objects.* 1993

[BDK91] BANCILHON, F. (Hrsg.) ; DELOBEL, C. (Hrsg.) ; KANELLAKIS, P. (Hrsg.): *Building an Object-Oriented Database System: The Story of O2.* Morgan Kaufmann, 1991

[BESS92] BRAUCH, I. ; EGGERT, H. ; SCHERER, K.P. ; STILLER, P.: Einsatz wissensbasierter Methoden für Konstruktion, Fertigung, und Test von LIGA-Mikrostrukturen. **In:** GÖRKE, W. (Hrsg.) ; RININSLAND, H. (Hrsg.) ; SYRBE, M. (Hrsg.): *Information als Produktionsfaktor* Springer Verlag, 1992, S. 714–721

[BGH91] BÜRG, B. ; GUTH, H. ; HELLMANN, A.: COSMOS-2D: Ein System zur Verifikation und Vermessung von zweidimensionalen geometrischen Formen. **In:** *KfK Nachrichten Jahrgang 23 2-3/91*, 1991. – Kernforschungszentrum Karlsruhe, S. 100–109

[BM91a] BLEY, P. ; MENZ, W.: Stand und Entwicklungsziele des LIGA-Verfahrens zur Herstellung von Mikrostrukturen. **In:** *KfK Nachrichten Jahrgang 23 2-3/91*, 1991. – Kernforschungszentrum Karlsruhe, S. 69–75

[BM91b] BURBAUM, C. ; MOHR, J.: *Herstellung von mikromechanischen Beschleunigungssensoren in LIGA-Technik..* KfK 4859, Kernforschungszentrum Karlsruhe, 1991

[Boo91] BOOCH, G. (Hrsg.): *Object Oriented Design with Applications.* Benjamin/Cummings Publishing Company, 1991

[Bra94] BRAUCH, I. Wissensbasierte Modellierung des LIGA–Fertigungsprozesses. Dissertation, Universität Karlsruhe, Institut für Maschinenbau. 1994

[BS93] BUDDE, R. ; SYLLA, K.H. Objektorientierte System-Entwicklung: Entwurfstechniken. Unterlagen zum Seminar der Deutschen Informatik Akademie. 12 1993

[BSW93] BRAUCH, I. ; SCHERER, K.P. ; WIELAND, P.: Unveröffentlichter Bericht. 1993. – Kernforschungszentrum Karlsruhe

[Cal84] Calma Company: *Stream Format GDS II, Release 5.1.* 1984

[Cat91] CATTELL, R.G.G. (Hrsg.): *Object Data Management: Object-Oriented and Extended Relational Database Systems.* Addison-Wesley, 1991

[Cat93] CATTELL, R.G.G. (Hrsg.): *The ODMG-93 Standard for Object Databases.* Morgan-Kaufmann Publishers, San Mateo, CA, 1993

[Che76] CHEN, P.P.: The entity-relationship model: toward a unified view of data. **In:** *ACM Transactions on Database Systems* 1 (1976), Nr. 1, S. 9–36

[CLF92] CHAMPEAUX, D. de ; LEA, D. ; FAURE, P.: The Process of Object-Oriented Design. **In:** PAEPCKE, Andreas (Hrsg.): *Proc. Conference on Object-Oriented Programming: Systems, Languages, and Applications, ECOOP '92* acm Press, 1992, S. 45–62

[Cod70] CODD, E.F.: A relational model of data for large shared data banks. **In:** *Communications of the ACM* 13 (1970), Nr. 6, S. 377–387

[COD71] CODASYL: CODASYL Data Base Task Group Report. **In:** *Proceedings ACM Conference on Data Systems Languages, New York* (1971)

[CY91] COAD, P. ; YOURDON, E.: *Object-Oriented Analysis.* Prentice-Hall, 1991

[Dat81] DATE, C.J.: *An introduction to database systems..* Addison-Wesley, 1981

[DeW90] DEWITT, D.: A study of three alternative Workstation-Server Architectures for Object Oriented Database Systems. **In:** *Proceedings of the 16th VLDB Conference, Brisbane, Australia*, 1990, S. 107–121

[Dit91] DITTRICH, K.R.: Object-Oriented Database Systems: The Notation and the Issues. **In:** DITTRICH, K.R. (Hrsg.) ; DAYAL, U. (Hrsg.) ; A.P.BUCHMANN (Hrsg.): *On object-oriented Database Systems.* Springer-Verlag, 1991, Kapitel 1, S. 3–10

[EB87] EHRFELD, W. ; BECKER, E.W.: Das LIGA-Verfahren zur Herstellung von Mikrostrukturkörpern mit großem Aspektverhältnis und großer Strukturhöhe. **In:** *KfK-Nachrichten* (1987), 4, S. 167–179

[ES93a] EGGERT, H. ; STILLER, P.: Unveröffentlichter Bericht. 1993. – Kernforschungszentrum Karlsruhe

[ES93b] EGGERT, H. ; STILLER, P.: Unveröffentlichter Bericht. 1993. – Kernforschungszentrum Karlsruhe

[Eul90] EULENBACH, D. Konstruktionssystem WISKON – Ein Beitrag zur Weiterentwicklung von CAD-Systemen. VDI Fortschrittsberichte R.20/Nr 32, VDI-Verlag, Düsseldorf. 1990

[Fai85] FAIRLEY, R.: *Software Engineering Concepts.* McGraw-Hill International, 1985

[GA90] GOSSAIN, S. ; ANDERSON, B.: An Iterative-Design Model for Reusable Object-Oriented Software. **In:** MEYROWITZ, N. (Hrsg.): *Proc. Conference on Object-Oriented Programming: Systems, Languages, and Applications, OOPSLA '90* acm Press, 1990, S. 12–27

[GAP93] GRABOWSKI, H. ; ANDERL, R. ; POLLY, A.: *Integriertes Produktmodell.* Beuth Verlag, 1993

[GAS89a] GRABOWSKI, H. ; ANDERL, H. ; SCHMITT, M.: Das Produktmodellkonzept von STEP. **In:** *VDI-Zeitschrift (Entwicklung Konstruktion Produktion), Düsseldorf 131* (1989), 12, S. 84–96

[GAS89b] GRABOWSKI, H. ; ANDERL, H. ; SCHMITT, M.: STEP – Entwicklung einer Schnittstelle zum Produktdatenaustausch. **In:** *VDI-Zeitschrift (Entwicklung Konstruktion Produktion), Düsseldorf 131* (1989), 9, S. 68–76

[GH91] GUPTA, R. (Hrsg.) ; HOROWITZ, E. (Hrsg.): *Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD.* Prentice Hall, 1991

[GZ92] GRYCZAN, G. ; ZÜLLIGHOVEN, H. Objektorientierte Systementwicklung. Informatik-Spektrum. 12 1992

[Har84] HARRINGTON, J.: *Understanding the Manufacturing Process*. Marcel Dekker, 1984

[Hel] HELLER, D.: *XView Programming Manual. An OPEN LOOK Toolkit for X11.* : O'Reilly and Associates, Inc.

[Heu89] HEUBERGER, A. (Hrsg.): *Mikromechanik*. Springer-Verlag, 1989

[Heu92] HEUER, A.: *Objektorientierte Datenbanken: Konzepte, Modelle, Systeme..* Addison-Wesley, 1992

[HSE90] HENDERSON-SELLERS, B. ; EDWARDS, J.M.: The Object-Oriented Systems Life Cycle. **In:** *Communications of the ACM* 33 (1990), S. 143–159

[IGE91] IGES/PDES Organization: *The Initial Graphics Exchange Specification (IGES), Version 5.1*. 1991

[Ken83] KENT, W.: A Simple Guide to Fife Normal Forms in Relational Database Theory. **In:** *Communications of the ACM* 26 (1983), Nr. 2, S. 120 –125

[Kim90] KIM, W.: Object-Oriented Databases: Definitions and Research Directions. **In:** *IEEE Transactions on Knowledge and Data Engineering* 2 (1990), Nr. 3, S. 327–341

[KM93] KEMPER, A. ; MOERKOTTE, G.: Basiskonzepte objektorientierter Datenbanksysteme. **In:** *Informatik-Spektrum* 16 (1993), S. 69–80

[LHKS91] LEWIS, J.A. ; HENRY, S.M. ; KAFURA, D.G. ; SCHULMAN, R.S.: An Empirical Study of the Object-Oriented Paradigm and Software Reuse. **In:** PAEPCKE, Andreas (Hrsg.): *Proc. Conference on Object-Oriented Programming: Systems, Languages, and Applications, ECOOP '91* acm Press, 1991, S. 184–196

[Loo91] LOOMIS, M.E.S. Objects and SQL: Accessing relational Databases. Versant Object Technology. 1991

[Loo92] LOOMIS, M.E.S. Integrating Objects with Relational Databases. Versant Object Technology. 1992

[LS87] LOCKEMANN, P.C. (Hrsg.) ; SCHMIDT, J.W. (Hrsg.): *Datenbank-Handbuch*. Springer-Verlag, 1987

[LSTK83] LOCKEMANN, P.C. (Hrsg.) ; SCHREINER, A. (Hrsg.) ; TRAUBOTH, H. (Hrsg.) ; KLOPPROGGE, M. (Hrsg.): *Systemanalyse*. Springer-Verlag, 1983

[MA] MELTON, J. (Hrsg.) ; A.R.SIMON (Hrsg.): *Understanding the New SQL: A Complete Guide.* Morgan Kaufmann Pub. ISBN:1-155860-245-3

[MB93] MENZ, W. ; BLEY, P.: *Mikrosystemtechnik für Ingenieure.* VCH Verlagsgesellschaft, 1993

[McH93] MCHENRY, S. RDBMSs vs. ODBMSs for Product Information Management Systems. 1993

[MD92] MEIER, A. ; DIPPOLD, R. Migration und Koexistenz heterogener Datenbanken. Informatik-Spektrum. 15 1992

[Men93] MENZ, W.: Die LIGA-Technik und ihr Potential für die industrielle Anwendung. **In:** *1. Statuskolloquium des Projektes Mikrosystemtechnik* KfK5238, Kernforschungszentrum Karlsruhe, 1993, S. 19–28

[Mey88] MEYER, B. (Hrsg.): *Objektorientierte Softwareentwicklung.* Carl Hanser Verlag, 1988

[Näh92] NÄHER, S.: *LEDA User Manual, Version 3.0.* Saarbrücken: Max-Planck-Institut für Informatik, 1992

[N.N93a] N.N. Industrial automation systems – Product data representation and exchange. ISO 10303-1 Overview and fundamental principles – Part 1. ISO TC184/SC4 N197 1993

[N.N93b] N.N. Industrial automation systems – Product data representation and exchange. ISO 10303-11 The EXPRESS language reference manual – Part 11. ISO TC184/SC4 N197 1993

[N.N93c] N.N. Industrial automation systems – Product data representation and exchange. ISO 10303-41 Fundamentals of product description and support – Part 41. ISO TC184/SC4 N197 1993

[Ora] Oracle Corporation: *Oracle for Sun-4, User's Guide, Programmer's Guide, Reference Manual, Reference Guide*

[RS77] ROSS, D. ; SCHOMAN, K.: Structured Analysis for Requirements Definition. **In:** *IEEE Transactions on Software Engineering* (1977)

[SBE 94a] SCHERER, K.P. ; BUCHBERGER, P. ; EGGERT, H. ; STILLER, P. ; WIELAND, P.: Knowledge Based Design and Manufacturing for LIGA-Microstructures. **In:** REICHL, H. (Hrsg.) ; HEUBERGER, A. (Hrsg.): *Micro Systems Technologies '94*, VDE-Verlag, Berlin, 1994, S. 1172–1174

[SBE 94b] SCHERER, K.P. ; BUCHBERGER, P. ; EGGERT, H. ; STILLER, P. ; WIELAND, P.: Wissensbasierte Entwurfsunterstützung zur Herstellung von LIGA-Mikrostrukturen. **In:** JOHN, W. (Hrsg.) ; EGGERT, H. (Hrsg.): *Methoden- und Werkzeugentwicklung für den Mikrosystementwurf*, 1994, S. 156–163

[SBHH90] SCHOMBURG, W. ; BLEY, P. ; HEIN, H. ; HOHR, J. Masken für die Röntgentiefenlithographie. VDI Berichte Nr. 870. 1990

[Sch90] SCHEER, A.W.: *CIM – Der computergesteuerte Industriebetrieb.* Springer-Verlag, 1990

[SES93a] SCHERER, K.P. ; EGGERT, H. ; STILLER, P.: Unveröffentlichter Bericht. 1993. – Kernforschungszentrum Karlsruhe

[SES93b] SCHERER, K.P. ; EGGERT, H. ; STILLER, P.: Unveröffentlichter Bericht. 1993. – Kernforschungszentrum Karlsruhe

[Sie92a] Siemens/Nixdorf: *Design Management, User's Guide.* SIFRAME V2.0. 1992

[Sie92b] Siemens/Nixdorf: *Desktop, Administrator's Guide.* SIFRAME V2.0. 1992

[Sie92c] Siemens/Nixdorf: *Desktop, User's Guide.* SIFRAME V2.0. 1992

[Sie92d] Siemens/Nixdorf: *Object Management System, Administrator's Guide.* SIFRAME V2.0. 1992

[Sie92e] Siemens/Nixdorf: *System Installation and Release Document, Administrator's Guide.* SIFRAME V2.0. 1992

[Sie92f] Siemens/Nixdorf: *System Overview.* SIFRAME V2.0. 1992

[Sie93a] Siemens/Nixdorf: *Design Management, User's Guide.* SIFRAME V2.1. 1993

[Sie93b] Siemens/Nixdorf: *Desktop, Administrator's Guide.* SIFRAME V2.1. 1993

[Sie93c] Siemens/Nixdorf: *Desktop, User's Guide.* SIFRAME V2.1. 1993

[Sie93d] Siemens/Nixdorf: *Object Management System, Administrator's Guide.* SIFRAME V2.1. 1993

[Sie93e] Siemens/Nixdorf: *System Installation and Release Document, Administrator's Guide.* SIFRAME V2.1. 1993

[Sie93f] Siemens/Nixdorf: *System Overview.* SIFRAME V2.1. 1993

[Str92] STROUSTRUP, B. (Hrsg.): *Die C++ Programmiersprache.* Addison-Wesley, 1992

[SZ87] SMITH, K.E. ; ZDONIK, S.B.: Intermedia: A case study of the diffeences between Relational and Object-Oriented Database Systems. **In:** *Proc. Conference on Object-Oriented Programming: Systems, Languages, and Applications, OOPSLA '87*, 1987, S. 452–465

[Ver] Versant Object Technology Corporation: *VERSANT ODBMS: System Manual, C++/VERSANT Usage Guide, C++/VERSANT Reference*

[Ver93] Versant Object Technology: *The VERSANT Star Architecture. Stepwise Integration of Relational and Object Technology.* 1993

[WBCS93] WIELAND, P. ; BRAUCH, I. ; C. DÜPMEIER ; SCHERER, K.P.: Unveröffentlichter Bericht. 1993. – Kernforschungszentrum Karlsruhe

[WBS92a] WIELAND, P. ; BRAUCH, I. ; SCHERER, K.P.: Unveröffentlichter Bericht. 1992. – Kernforschungszentrum Karlsruhe

[WBS92b] WIELAND, P. ; BRAUCH, I. ; SCHERER, K.P.: Unveröffentlichter Bericht. 1992. – Kernforschungszentrum Karlsruhe

[Wed92] WEDEKIND, H.: *Objektorientierte Schemaentwicklung: ein kategoraler Ansatz für Datenbanken und Programmierung.* BI Wissenschaftsverlag, 1992

[WFGL91] WEINREB, D. ; FEINBERG, N. ; GERSON, D. ; LAMB, C.: An Object-Oriented Database System to Support an Integrated Programming Environment. **In:** GUPTA, R. (Hrsg.) ; HOROWITZ, E. (Hrsg.): *ObjectOriented Databases with Applications to CASE, Networks, and VLSI CAD.* Prentice Hall, 1991, S. 116–129

[ZM91] ZDONIK, S.B. ; MAIER, D.: Fundamentals of Object-Oriented Databases. **In:** *Readings in Object-Oriented Database Systems.* Morgan Kaufmann Publishers, 1991, S. 1–32

Anhang A

Anhang zum Partialmodell der Entwurfssicht für Mikrostrukturen

In Kapitel 4.4 wurden das Partialmodell der Entwurfssicht für Mikrostrukturen vorgestellt. Dieser Anhang enthält eine Beschreibung der dabei relevanten Datenformate IGES [ES93a] und GDSII, sowie eine Funktionsbeschreibung der Konverter (Prä- und Postprozessoren [ES93b]).

A.1 Vorbedingungen aus dem CAD-Entwurf

Die Konstruktion der Mikrostrukturen erfolgt mit Hilfe eines CAD-Systems für den mechanischen Entwurf (MCAD). Beim Entwurf ist darauf zu achten, daß

- alle Kantenzüge aus Geraden oder Kreisbögen bestehen,
- alle Kantenzüge geschlossen sind,
- zwei Kantenzüge sich nicht schneiden, und insbesondere
- Kanten nicht doppelt gezeichnet werden,
- keine T-Stücke auftreten (eine Kante endet auf einer anderen Kante).

Die MCAD-Systeme bieten gegenüber den CAD-Systemen für den Elektronik-Entwurf (ECAD) die Möglichkeit der Bemaßung von Zeichnungen oder Zeichnungsteilen.

Unter Verwendung der Bemaßungsfunktionalität (durch Anhängen einer oder mehrerer Kennungen) erfolgt die Spezifikation derjenigen Strukturteile, die bei der Qualitätskontrolle zu untersuchen sind. Dabei bedeuten:

L	verifiziere Linie	bestimme Standard- und Maximalabweichung der geraden Verbindung zwischen den Endpunkten
S	verifiziere Kreisbogen	bestimme Mittelpunkt, Radius und Standard- und Maximalabweichung des Kreisbogens
T	verifiziere Vollkreis	bestimme Mittelpunkt, Radius und Standard- und Maximalabweichung des Vollkreises
D	bestimme Distanz	bestimme Abstand zwischen den Endpunkten
B	verifiziere Linienbreite	bestimme mittlere Linienbreite mit Standard- und Maximalabweichung
A	überprüfe Vollständigkeit	verifiziere alle vorhandenen Kanten

Die Kennungen können beliebig in Groß- oder Kleinbuchstaben angegeben werden. Innerhalb des CAD-Systems erfolgt keine Konsistenzprüfung der angehängten Kennungen an die Figurenteile, eine Kennung 'L' an einem Kreisbogen führt also nicht zu einer Fehlermeldung[1]. Der CAD-Konstrukteur muß also selbst dafür sorgen, daß

- L an Linien
- S an Kreisbögen
- T an Vollkreisen
- D an zwei Punkten, also Endpunkte von Linien oder Kreisbögen oder Punkte auf Linien oder Kreisbögen
- B an zwei Punkte auf zwei Linien

angehängt werden. Die Kennung 'A' an einer beliebigen Kante der Figur bewirkt die Vollständigkeitüberprüfung der gesamten Figur.

Toleranzen können in der Form

Maß ± Toleranz

vorgegeben werden, die Kennung ist in diesem Fall an die Toleranz anzuhängen (z. B. .100±.005 D für eine Distanzmessung mit dem Sollwert 100 mm und einer Toleranz von ±5 mm).

[1] Die Konsistenzprüfung erfolgt später beim Aufbau der Objekthierarchie, dort werden inkonstistente Kennungen entweder still ignoriert oder es wird eine Fehlermeldung ausgedruckt.

A.2 Das IGES-Format und dessen Transformation in das Geometriemodell

A.2.1 Überblick über IGES

Die Abkürzung IGES steht für *Initial Graphics Exchange Specification.* Unter IGES versteht man eine Spezifikation der Beschreibung von geometrischen Produktdaten mit dem Ziel, diese Produktdaten zwischen CAD/CAM-Systemen unterschiedlicher Hersteller auf unterschiedlicher Hardware austauschen zu können. Um eine weitgehende Rechnerunabhängigkeit zu erreichen, wird die Beschreibung in eine sequentielle Datei mit festen 80 Zeichen langen Sätzen gepackt und als Zeichensatz wird ASCII (7-Bit) vorgeschrieben.

Zur Beschreibung der geometrischen Produktdaten stehen in IGES eine Vielzahl von Elementen zur Verfügung. Diese Elemente lassen sich grob in 4 Klassen einteilen:

- Beschreibung von Kurven und Flächen
- Beschreibung von dreidimensionalen Körpern (CSG, Constructive Solid Geometry Model)
- Beschreibung von Bemaßungen
- Beschreibung der Struktur von geometrischen Daten

Zur Übersicht sind in Tabelle A.1 alle in IGES (Version 5.0) definierten Elemente entsprechend ihren Objektklassen aufgeführt.

Für die Konstruktion von LIGA-Strukturen wurde eine Untermenge aller IGES-Elemente implementiert. Die Auswahl der Elemente ergab sich zum einen aus der Aufgabenstellung (Kurven, Flächen und Bemaßung) und zum anderen aus den vom CAD-System automatisch generierten IGES-Strukturen. Das bedeutet natürlich eine Abhängigkeit vom gewählten CAD-System in bezug auf die implementierten Strukturen. Beim Einsatz anderer CAD-Systeme ist zu prüfen, inwiefern die generierten IGES-Dateien weitere Elemente (insbesondere Strukturelemente) enthalten. Nachfolgend sind alle implementierten Elemente aufgelistet:

Kurven und Flächen	Bemaßung	Strukturen
Circular Arc	Copious Data	Subfigure Definition
Line	General Note	Associativity Instance
Point	Leader (Arrow)	Singular Subfigure Instance
	Linear Dimension	
	Ordinate Dimension	
	Point Dimension	
	Radius Dimension	

Die Beschreibung der Elemente ist [IGE91] zu entnehmen.

Kurven und Flächen	3-dimensionale Elemente	Bemaßung	Strukturen
Circular Arc	Block	Copious Data	Null
Composite Curve	Right Angular Wedge	Angular Dimension	Connect Point
Conic Arc	Right Circular Cylinder	Curve Dimension	Node
Copious Data	Right Circular Cone Frustum	Diameter Dimension	Finite Element
Plane	Sphere	Flag Note	Nodal Displacement
Line	Torus	General Label	Nodal Results
Parametric Spline Curve	Solid of Revolution	General Note	Associativity Definition
Parametric Spline Surface	Solid of Linear Extrusion	New General Note	LineFont Definition
Point	Boolean Tree	Leader (Arrow)	MACRO Definiton
Ruled Surface	Solid Assembly	Linear Dimension	Subfigure Definition
Surface of Revolution	Solid Instance	Ordinate Dimension	Text Font Definition
Tabulated Cylinder		Point Dimension	Text Display Template
Transformation Matrix		Radius Dimension	Color Definiton
Flash		General Symbol	Units Data
B-Spline Curve		Section Area	Network Subfigure Definition
B-Spline Surface			Associativity Instance
Offset Curve			Drawing
Offset Surface			Property
Boundary			Singular Subfigure Instance
Curve on a Parametric Surface			View
Bounded Surface			Rectangle Arry Subfigure Instance
Trimmed Parametric Surface			Circular Arry Subfigure Instance
			External Reference
			Nodal Load/Constraint
			Network Subfigure Instance
			Attribute Table Instance

für die Konstruktion von LIGA-Strukturen implementierte Elemente (heutiger Stand)

Tabelle A.1: IGES-Elemente (Version 5.0), entsprechend ihren Objektklassen

Aufbau einer IGES-Datei

Wie schon anfangs erwähnt besteht eine IGES-Datei aus einer Folge von Sätzen, die jeweils 80 (ASCII) Zeichen lang sind. Ein Satz wird in 3 Felder geteilt: Spalte 1 bis 72 enthält das Informationsfeld, Spalte 73 enthält die Kennung des aktuellen Abschnitts und in Spalte 74-80 steht rechtsbündig angeordnet die Zeilennummer im jeweiligen Abschnitt. Die Syntax des Informationsfeldes wird im jeweiligen Abschnitt festgelegt. Eine Datei ist in fünf unterschiedliche Abschnitte (Sections) gegliedert:

1. Start Section

   Im Startabschnitt steht ein lesbarer Kommentar. In IGES selbst ist keine Syntax für das Informationsfeld festgelegt. Bei VDAIS (IGES-Subset des Verbands der Autohersteller) wird hier allerdings ein Kopf mit Informationen über Leistungsstufe, Modellbeschreibung, Sender und Empfänger des Modells in einem fest vorgegebenen Format eingefügt. Die Abschnittskennung besteht aus dem Buchstaben **S**.

2. Global Section

   Die in diesem Abschnitt aufgeführten Angaben dienen zur Interpretation der Daten in den beiden nachfolgenden Abschnitten. So legen die beiden ersten Angaben die Trennzeichen für Parameter und "Records"(Parameterfolgen über mehrere Sätze) fest. Weitere wichtige Angaben sind die Skalierung der gesamten Zeichnung, die Maßeinheit der Zeichnung, maximale Strichstärke und Anzahl der unterschiedlichen Strichstärken. Die vollständige Liste der Parameter ist in [IGE91] Kapitel 2.2.4.2 *Global Section* aufgeführt. Die Abschnittskennung der *Global Section* besteht aus dem Buchstaben **G**.

3. Directory Entry Section

   Dieser Abschnitt enthält ein Verzeichnis mit allen Elementen, aus denen ein Modell aufgebaut wird. Die Einträge in diesem Verzeichnis sind gleich strukturiert. Jedem Element sind zwei Zeilen zugeordnet und die Information dieser beiden Zeilen ist in jeweils 9 Felder je 8 Zeichen lang unterteilt. Die Parameter werden ohne Trennzeichen rechtsbündig in diese Felder geschrieben. Ein Verzeichniseintrag enthält eine Nummer, die den Elementtyp beschreibt, einige allgemeine Attribute zu diesem Element und einen Verweis auf eine Stelle in der *Parameter Data Section*, an der weitere elementspezifische Parameter stehen. Allgemeine Elementattribute sind z.B. Farbe, Zeichnungsebene, Linientyp usw. Die vollständige Liste der Parameter ist in [IGE91] Kapitel 2.2.4.3 *Directory Section* aufgeführt. Die Abschnittskennung der Directory Section besteht aus dem Buchstaben **D**.

4. Parameter Data Section

   In diesem Abschnitt werden alle Parameter (Koordinaten, Texte, Größenangaben, usw.) zu den Elementen aufgelistet, die in der *Directory*

Entry Section eingetragen sind. Die Anzahl und der Typ der Parameter ist vom jeweiligen Element abhängig. Parameter werden im freien Format, mit dem in der *Global Section* spezifizierten Zeichen getrennt, im Informationsfeld eines Datensatzes eingetragen. Das Informationsfeld wird allerdings um 8 Zeichen auf Spalte 64 gekürzt. Spalte 65 muß ein Leerzeichen enthalten und in Spalte 66 bis 72 steht die Zeilennummer des Eintrags in der Directory Section, der das Element mit den hier aufgeführten Parametern enthält. Diese Rückverzeigerung dient als zusätzliche Kontrolle für die Konsistenz der Daten. Der erste Parameter im Informationsfeld ist immer die Nummer des Elementtyps. Das Ende einer Folge von Parametern wird durch ein "Record"-Trennzeichen (definiert in der *Global Section*) markiert. Die vollständige Beschreibung der *Parameter Data Section* ist in [IGE91] Kapitel 2.2.4.4 *Parameter Data Section* aufgeführt. Die Abschnittskennung der *Parameter Data Section* besteht aus dem Buchstaben **P**.

5. Terminate Section

   Die Terminate Section besteht nur aus einem Satz und enthält zur Kontrolle die Anzahl der Datensätze der vorherigen Abschnitte.

A.2.2 Erzeugen einer Objekthierarchie

Einlesen

Beim Einlesen einer Zeile aus der gefilterten IGES-Datei wird jeweils ein Objekt der entsprechenden Klasse (Figur, Punkt, Linie, Kreisbogen, Bemaßung) erzeugt, und mit den gefundenen Parametern initialisiert. Für eine Kante (Linie oder Kreisbogen) werden die Anfangsecke und die Endecke dieser Kante erzeugt (beide noch vom generischen Eckentype 'E') und in die Vorgänger- und Nachfolgerattribute der Kante eingetragen. Analog wird die Kante als Nachfolger der Anfangsecke und als Vorgänger der Endecke initialisiert, d.h. alle Ecken haben hier entweder einen Eintrag im Vorgängerattribut oder einen Eintrag im Nachfolgerattribut. Anschließend werden die Kanten in die Kantenliste, die Ecken in die Eckenliste eingetragen. Punkte werden in die Punkteliste, Bemaßungen in die Bemaßungsliste und Figuren in die Subfigurenliste der aktuellen Figur eingetragen.

Referenzfiguren werden in die Referenzfigurenliste der gerade gelesenen Figur eingetragen, unabhängig davon, ob sie sich auf diese oder auf eine andere Figur beziehen. Deswegen muß nach dem Einlesen der komplette Figurenbaum nochmals durchlaufen werden, um die Verbindungen der Referenzfiguren zu den richtigen Figuren herzustellen. Dazu dient als Zuordnungskriterium die Figuren/RefFiguren-id, die (als eindeutige Kennung) die Zeilennummer der Definition der Figur in der IGES-Datei enthält. Nachdem diese Zuordnung erfolgt ist, werden alle Punkte, Kanten und Bemaßungseinträge der Figur der linearen Abbildung unterworfen, d.h. für jede RefFigur wird eine konkrete Figur erzeugt.

Sortieren der Kanten

Nach dem Einlesen liegen in der Kantenliste alle Kanten, in der Eckenliste alle Ecken einer Figur in unsortierter Form vor. Für die Kanten sind jeweils Anfangs- und Endecken bekannt, jede Ecke kennt allerdings nur entweder ihre Vorgänger- oder ihre Nachfolgerkante.

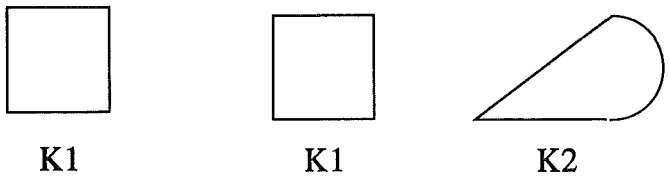
Ausgehend von der ersten Kante K in der Kantenliste (mit bekannter Endecke E) wird die Eckenliste nach Ecken durchsucht, die die gleiche X/Y-Position haben wie die Endecke der Kante, allerdings von dieser Endecke verschieden sind (d.h. eine andere Eckeninstanz besitzen). Es ergeben sich die folgenden Möglichkeiten:

- Es existiert keine andere Eckeninstanz E' mit der X/Y-Position von E, d.h. nur eine Ecke mit dieser Position wurde erzeugt. Der Kantenzug endet hier mit einer Ecke, die keine Nachfolgerkante hat (Sackgasse). Dieser Kantenzug wird bei der nachfolgenden Bearbeitung nicht mehr berücksichtigt.

- Es existiert genau eine andere Eckeninstanz E' mit der X/Y-Position von E, die von E verschieden ist. E' wurde bein Einlesen einer Kante K' als deren Anfangsecke oder Endecke erzeugt. Ist E' die Anfangsecke von K', dann wird K' als Nachfolgekante von E eingetragen. Ist E' die Endecke von K', dann werden in K' Anfangspunkt und Endpunkt sowie Anfangsecke und Endecke vertauscht und danach K' als Nachfolgekante von E eingetragen - allerdings nur unter der Bedingung, daß K' nicht die gleiche Anfangsecke und Endecke wie K hat; dies würde bedeuten, daß die Kante K im CAD-Entwurf zweimal vorkommt, die Kante doppelt gezeichnet wäre; dieser Kantenzug würde bei der weiteren Bearbeitung nicht berücksichtigt.

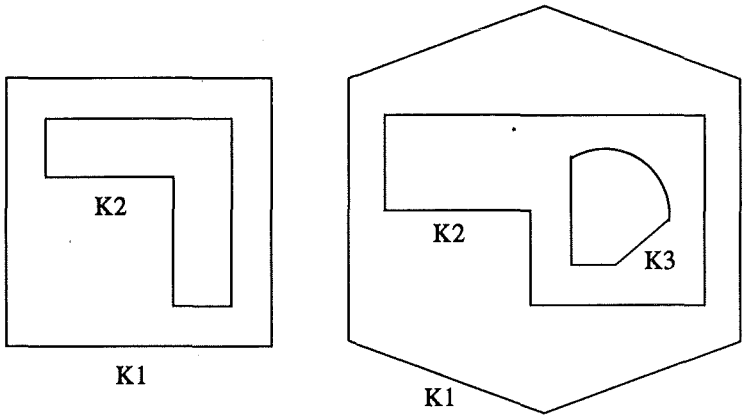
- Es existieren mehrere Eckeninstanzen E', E", ... mit der X/Y-Position von E, die alle untereinander verschieden sind. Diese wurden beim Einlesen der Kanten K', K", ... erzeugt, also laufen alle diese Kanten an der X/Y-Position der Ecke E zusammen. Das verstößt gegen die (von COSMOS vorgegebene) Randbedingung, daß in jede Ecke nur genau zwei Kanten zusammenlaufen dürfen. Diese Situation kann auch auftreten, wenn beim CAD-Entwurf mehrere Kanten übereinander gezeichnet werden. Dann sieht die CAD-Zeichnung zwar korrekt aus, der entsprechende Kantenzug wird aber bei der nachfolgenden Bearbeitung nicht mehr berücksichtigt.

Falls zur Kante K eine Nachfolgerkante gefunden werden konnte, wird analog weiterverfahren, bis entweder alle Kanten in der Kanteliste abgearbeitet wurden, oder die Startkante K wieder erreicht ist. Alle in dieser Form gefundenen Kanten werden in einer *Kantensequenz* zusammengefaßt. Falls die Startkante K wieder erreicht wurde, allerdings noch nicht alle Kanten in der Kantenliste abgearbeitet wurden, besteht die Figur aus mehreren Kantensequenzen (siehe Abbildung A.1).



(a) Figur 1 bestehend aus einer Kantensequenz

(b) Figur 2 bestehend aus zwei Kantensequenzen K1 und K2



(c) Figur 3 bestehend aus zwei Kantensequenzen

(d) Figur 4 bestehend aus drei Kantensequenzen

Abbildung A.1: Kantensequenzen

Klassifikation der Ecken

Die Klassifikation der Ecken erfolgt im Hinblick auf die Bildverarbeitungsoperationen, die später zur Erkennung dieser Ecken angewandt werden. Da für jede Kante einer Kantensequenz die Anfangs- und Endecke und damit für jede Ecke die Vorgänger- und Nachfolgerkante bekannt sind, können die Ecken klassifiziert werden:

- X-Ecken haben zwei Linien als Vorgänger- und Nachfolgerkante, bei
- Y-Ecken ist genau eine Nachbarkante eine Linie und die andere ein Kreisbogen,
- Z-Ecken haben jeweils einen Kreisbogen als Vorgänger- und Nachfolgerkante.

Punkte auf Kanten

Wird im CAD-System ein Punkt auf einer Kante konstruiert (etwa weil der Abstand von der Kante zu einer Ecke bestimmt werden soll), dann teilt die-

ser Punkt die Kante in zwei Kanten des gleichen Typs. Demnach müssen nun alle X- und Z-Ecken (also nicht die Y-Ecken) daraufhin untersucht werden, ob die Vorgänger- und die Nachfolgerkante die gleiche Richtung haben (Linien), bzw. ob sie auf dem gleichen Kreisbogen liegen (gleicher Mittelpunkt und gleicher Radius reichen für die Bestimmung des gleichen Kreisbogens nicht aus). Gegebenenfalls wird die X- oder Z-Ecke aus der Eckenliste entfernt und ein Objekt des Types *Punkt_auf_Kantensegment* an dieser Stelle erzeugt und in die Punkteliste eingetragen, sowie die Vorgänger- und die Nachfolgerkante zusammengefaßt. Punkte, die nicht auf Kanten liegen, also nicht vom Typ *Punkt_auf_Kantensegment* sind, werden aus der Punkteliste entfernt, da sie bildverarbeitungsmäßig nicht erfaßbar sind.

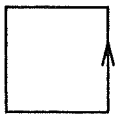
Orientierung der Kantensequenzen

Da die Orientierung jeder Kantensequenz für das Bildverarbeitungssystem eine wesentliche Rolle spielt, wird sie für jede Kantensequenz bestimmt, und gegebenenfalls gemäß den Anforderungen gesetzt. Kantensequenzen werden entweder im Uhrzeigersinn oder im Gegenuhrzeigersinn durchlaufen, wobei die Forderung erfüllt sein muß, daß wenn man jeder Kante eine Richtung unterstellt, die vom Anfangspunkt der Kante zum Endpunkt der Kante zeigt und man sich auf der Kante in dieser Richtung bewegt, daß dann immer auf derselben Seite der Kante das *Innere* der Figur liegt. Demzufolge muß eine Kantensequenz, die vollständig innerhalb einer anderen Kantensequenz liegt, immer mit der entgegengesetzten Orientierung durchlaufen werden (Abbildung A.2).

Das Ordnen der Kantensequenzen bezüglich der 'liegt-innerhalb'-Relation muß über alle Kantensequenzen aller Figuren durchgeführt werden (Abbildung A.3).

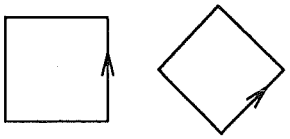
Zunächst werden innerhalb einer Figur für alle Kantensequenzen die Punkte mit der größten X-Koordinate (RX) bestimmt, und der Größe nach geordnet. Von diesen aus wird jeweils ein zur X-Achse paralleler Strahl mit allen Kanten der Kantensequenzen mit größeren RX-Werten geschnitten. Je nach Anzahl der Schnittpunkte kann so bestimmt werden, welche Kantensequenzen innerhalb welcher Kantensequenzen liegen (Kantensequenzen dürfen sich nicht schneiden). Nachdem dies erfolgt ist, müssen lediglich noch die äußeren Kantensequenzen der Figuren untereinander verglichen werden.

Damit sind die CAD-Daten in eine Objekthierarchie überführt. Wurzel dieser Hierarchie ist immer eine Figur, die ihrerseits wieder aus Figuren sowie Ecken, Kanten, Punkten und Kantensequenzen bestehen kann. Aus den Elementen dieser Hierarchie werden zum einen die GDSII Daten für den Elektronenstrahlschreiber, zum anderen unter Verwendung der Bemaßungen die Vermessungsaufträge für das Bildverarbeitungssystem erzeugt.



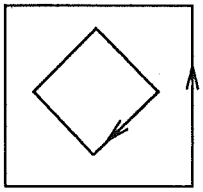
(a)

Figur 1 bestehend aus einer Kantensequenz, die im Gegenuhrzeigersinn durchlaufen wird.



(b)

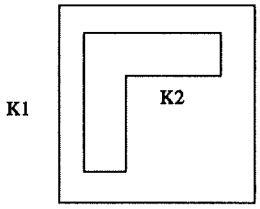
Figur 2 bestehend aus zwei Kantensequenzen, die im Gegenuhrzeigersinn durchlaufen werden.



(c)

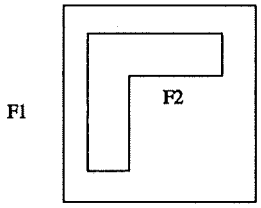
Figur 3 bestehend aus zwei Kantensequenzen K1 und K2, die im Gegenuhrzeigersinn (K1) und im Uhrzeigersinn (K2) durchlaufen werden. K2 beschreibt die Umrandung eines Loches in der Figur 3, K1 beschreibt den Außenrand der Figur 3.

Abbildung A.2: Orientierung Kantensequenzen



(a)

Figur 1 bestehend aus zwei Kantensequenzen, K1 und K2.



(b)

Figur 2 bestehend aus zwei Figuren F1 und F2, die jeweils aus einer Kantensequenz bestehen.

Abbildung A.3: Orientierung Figuren

A.3 Das GDSII-Format und dessen Erzeugung aus dem Geometriemodell

Zum Verständnis des Konverters sind Informationen über Struktur und Eigenschaften der Ein- und Ausgabeformate des Konverters notwendig. Das Eingabeformat beruht auf dem objektorientierten Datenformat (GEM), wie es in Kapitel 5.1.3 beschrieben ist. Das Ausgabeformat GDSII ist in [Cal84] definiert. Nicht alle in GDSII verfügbaren Elemente sind implementiert. Es wurden nur die Elemente ausgewählt, die für die Konvertierung der im objektorientierten Datenformat definierten Figuren in eine GDSII-Struktur notwendig sind.

Im objektorientierten Datenformat wurden Referenzen auf definierte Figuren implementiert, d.h. zu einer Figur existiert eine oder mehrere Referenzen, die eine Abbildung dieser Figur definieren. Die Abbildungsfunktion wird durch eine Transformationsmatrix und einen Translationsvektor festgelegt. Diese Funktionalität soll auch bei der Erzeugung der GDSII-Ausgabe zur Verfügung stehen. Bei der Implementierung wird dazu das GDSII-Element SREF benutzt. Allerdings gilt die Einschränkung, daß als Abbildungsfunktionen nur Rotation, Skalierung und Translation erlaubt sind, da eine Transformation einer Struktur in GDSII nur über die Angabe eines Rotationswinkels, eines Skalierungsfaktors und eines Translationsvektors möglich ist.

A.3.1 Überblick über GDSII

Das Stream-Format der Graphic Data Station II (GDSII) ist ein internes Datenformat der Firma Calma Company zur Übertragung und Sicherung von Konstruktionsdaten ihrer eigenen Grafikrechner (Graphic Data Station II). Da die Firma Calma nicht mehr existiert, wird das Datenformat nicht mehr gepflegt. Auch Dokumentation ist nur eingeschränkt verfügbar [Cal84]. Obwohl das Format signifikante Nachteile aufweist, hat es sich als ein defacto-Standard für den Datenaustausch zwischen Konstruktionssystemen für den elektronischen Entwurf durchgesetzt. Mit GDSII ist ausschließlich eine Beschreibung von Flächen möglich. Ein Fläche wird durch das sie umfassende Polygon beschrieben, d.h. die Kante n eines Polygons sind immer Geradenstücke. Eine kreisrunde Fläche muß durch ein Polygon mit einer Vielzahl von Eckpunkte approximiert werden. Ein schwerwiegender Nachteil von GDSII ist hier die Beschränkung auf Polygone mit weniger als 200 Ecken. Ein Polygon, das mehr als 200 Punkte aufweist, kann nicht verarbeitet werden. Muß zum Beispiel eine kreisrunde Fläche sehr genau approximiert werden, so müssen zum Beschreiben mehrere Polygone erzeugt werden, die zusammengesetzt die gewünschte Kreisfläche ergeben.

A.3.2 Syntax einer GDSII-Datei

Der Aufbau einer GDSII-Datei läßt sich durch eine Syntaxdefinition in Backus-Naur-Repräsentation [ASU86] beschreiben. Eckige Klammern ([ ]) beschreiben

einen Inhalt, der optional ist. Geschweifte Klammern ({ }) bedeuten, der Inhalt ist optional, kann aber beliebig oft wiederholt auftreten. Geschweifte Klammern mit einem Pluszeichen ({ }+) bedeuten, daß der Inhalt mindestens einmal auftreten muß, aber beliebig oft wiederholt werden kann. Ein senkrechter Strich (|) steht für *oder*, d.h. aus einer Liste von Elementen getrennt, durch senkrechte Striche, muß genau ein Element gewählt werden.

<GDSII-Datei> ::= HEADER BGNLIB LIBNAME [REFLIBS] [FONTS] [ATTRTABLE] [GENERATIONS] [<FormatType>] UNITS {<Structures>} ENDLIB

<FormatType> ::= FORMAT | FORMAT {MASK}+ ENDMASKS

<Structures> ::= BGNSTR STRNAME [STRCLASS] {<Element>} ENDSTR

< Element> ::= <boundar> | <path> | <SREF> | <AREF> | <text> | <node> | <box> {<property>} ENDEL

<boundary> ::= BOUNDARY [ELFLAGS] [PLEX] LAYER DATATYPE XY

<path> ::= PATH [ELFLAGS] [PLEX] LAYER DATATYPE [PATHTYPE] [WIDTH] [BGNEXTN] [ENDEXTN] XY

<SREF> ::= SREF [ELFLAGS] [PLEX] SNAME [<strans>] XY

<AREF> ::= AREF [ELFLAGS] [PLEX] SNAME [<strans>] COLROW XY

<text> ::= TEXT [ELFLAGS] [PLEX] LAYER <textbody>

<node> ::= NODE [ELFLAGS] [PLEX] LAYER NODETYPE XY

<box> ::= BOX [ELFLAGS] [PLEX] LAYER BOXTYPE XY

<textbody> ::= TEXTTYPE [PRESENTATION] [PATHTYPE] [WIDTH] [<strans>] XY STRING

<strans> ::= STRANS [MAG] [ANGLE]

<property> ::= PROPATTR PROPVALUE

A.3.3 Klassenhierarchie der GDSII-Daten

Die definierten Klassen und ihre Hierarchie sind in Abbildung A.4 dargestellt.

Die Klasse *GDSfile* repräsentiert die GDSII-Dateistruktur, sie enthält unter anderem eine Liste mit allen definierten GDS-Strukturen. Die Klasse *GDSstructure* repräsentiert eine GDSII-Struktur, sie enthält unter anderem eine Liste mit allen Elementen dieser Struktur.

Die Klasse *GDSelement* ist die Basisklasse für die unterschiedlichen GDSII-Elemente. Die Klasse *GDSboundary* für das GDSII-Element BOUNDARY, das Element repräsentiert ein geschlossenes Polygon. Es enthält die Liste aller Eckpunkte des Polygons und ein Verweis auf die Orientierung der Punktefolge (mit oder gegen den Uhrzeigersinn).

Die Klasse *GDSsref* für das GDSII-Element SREF, das Element repräsentiert eine Referenz auf eine andere GDSII-Struktur. Die Klasse *GDSaref* für das GDSII-Element AREF, das Element repräsentiert eine Tabelle von gleichen GDSII-Strukturen. Sie enthält den Verweis auf die GDSII-Struktur, die Anzahl der Spalten und Reihen der Tabelle und die Ausdehnung der Tabelle in X- und Y-Richtung.

Die Klasse *GDSpoint* repräsentiert einen Punkt mit GDSII Daten-Koordinaten als Besonderheit enthält der Punkt einen Verweis auf die Position in einer Liste, sofern der Punkt einer Liste zugeordnet ist. Die Klasse *GDSline* repräsentiert eine Linie mit GDSII Daten-Koordinaten. Sie ist eine Hilfsklasse und in Abbildung A.4 nicht dargestellt.

A.3.4 Satztypen

In GDSII werden 56 Satztypen definiert, die in Tabelle A.2 aufgelistet sind. Die später im Programm benötigten Typen sind grau hinterlegt.

HEADER	BGNLIB	LIBNAME	UNITS
ENDLIB	BGNSTR	STRNAME	ENDSTR
BOUNDARY	PATH	SREF	AREF
TEXT	LAYER	DATATYPE	WIDTH
XY	ENDEL	SNAME	COLROW
TEXTNDOE	NODE	TEXTTYPE	PRESENT.
SPACING	STRING	STRANS	MAG
ANGLE	UINTEGER	USTRING	REFLIBS
FONTS	PATHTYPE	GENERATION	ATTRTABLE
STYPTABLE	STRTYPE	ELFLAGS	ELKEY
LINKTYPE	LINKKEYS	NODETYPE	PROPATTR
PROPVALUE	BOX	BOXTYPE	PLEX
BGNEXTN	ENDEXTN	TAPENUM	TAPECODE
STRCLASS	RESERVED	FORMAT	MASK

Tabelle A.2: Tabelle der definierten Satztypen

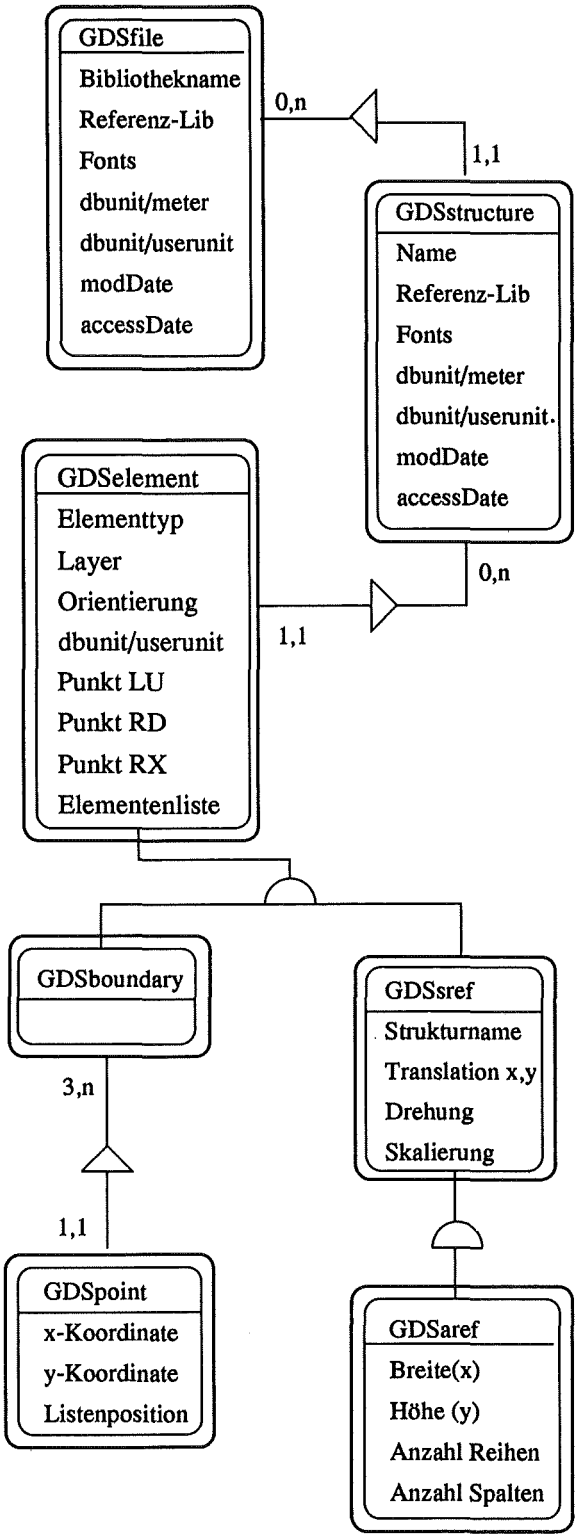


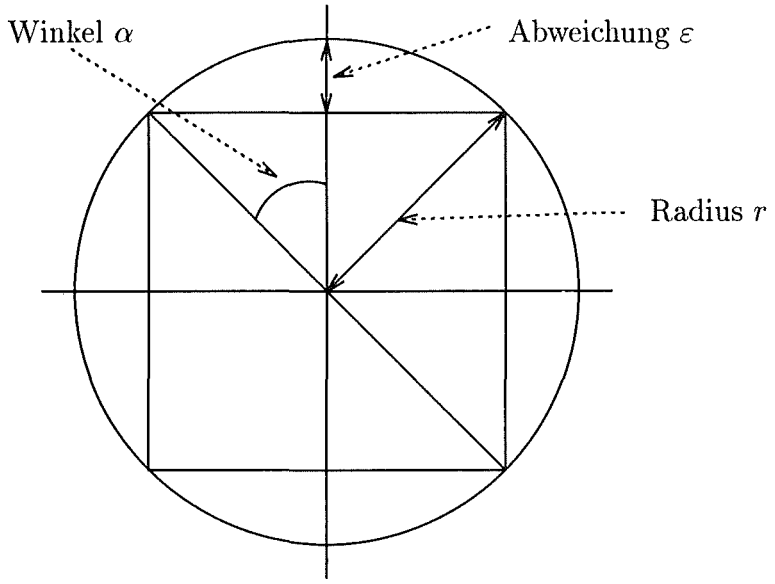
Abbildung A.4: Abhängigkeiten der Klassen bei GDSII

A.3.5 Umwandlung von Kantensequenzen in Polygone

Für die Konvertierung in GDSII-Daten müssen Kantensequenzen in GDS-Boundary-Elemente (Polygone) umgewandelt werden, d.h. jede Kantensequenz wird in eine Folge von Punkten konvertiert. Die Anzahl der Punkte, die bei der Umwandlung eines Kreisbogens erzeugt werden, wird über einen Genauigkeitsparameter gesteuert. Dazu gibt es zwei Möglichkeiten:

- Zum einen wird die Genauigkeit durch die maximale Anzahl der Punkte für einen Vollkreis festgelegt. Die Anzahl der Punkte ist damit vom Radius unabhängig. Kleine und große Kreise bzw. Kreissegmente werden immer in die gleiche Anzahl von Punkten zerlegt.
- Zum anderen wird eine maximale Abweichung der Polygonkante vom Kreisbogen vorgegeben. Für jedes Kreissegment wird dann die Anzahl der Punkte wie folgt berechnet:

$$\text{Anzahl der Punkte} = (\pi / \arccos(1 - (\varepsilon / r)))$$



A.3.6 Funktionsbeschreibung des Konverters

Der Konverter besteht aus den beiden Komponenten *GEM2gef* und *gef2gds*. Zwischen diesen Konvertern wird zur Zeit noch eine Zwischendatei zu Testzwecken angelegt.

Nach Sortieren und Orden der Daten werden alle Listen mit Kantensequenzen der Figuren des objektorientierten Datenformats durchlaufen und für jede Kantensequenz ihre Punkte in eine Zwischendatei geschrieben (*GEM2gef*). Diese Zwischendatei wird dann von dem zweiten Programm *gef2gds* gelesen, das die eigentliche Konvertierung in GDSII vornimmt. Die Zwischendatei ist eine

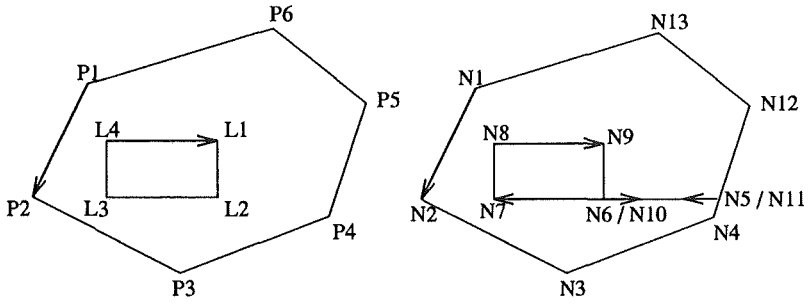
sequentielle ASCII-Datei und enthält außer den Punkten der Kantensequenzen noch Informationen über die IGES-Ursprungsdatei und über die Figuren.

Das Programm gef2gds liest nun die Daten aus der Zwischendatei und baut eine Objekthierarchie entsprechend der GDS-Klassenhierarchie auf. Beim Aufruf des Programms wird über Kommandozeilenparameter festgelegt, ob die einzulesende Figur in eine einfache GDS-Struktur oder vielfach in eine Strukturentabelle konvertiert werden soll. Anhand der Info-Zeile und den Programmoptionen wird ein Objekt der Klasse GDSfile angelegt und initialisiert.

Wird eine Strukturentabelle gefordert, so wird zusätzlich zur Struktur für die zu konvertierende Figur eine zweite Struktur mit einem GDS-Element AREF erzeugt. In AREF werden die Angaben über Spalten, Reihe und Abstand der Strukturentabelle und der Name der zu platzierenden Struktur abgelegt. Alle nachfolgenden Figurdefininitionen werden als GDS-Strukturen in die Strukturenliste des GDSfile-Objekts eingetragen. Für jede Referenzen auf eine Figur wird das GDS-Element SREF erzeugt. Ist der Einlesevorgang beendet müssen die Daten für die Konvertierung vorbereitet werden. Dazu sind zwei Schritte notwendig:

1. Schritt:

Da nur Flächenpolygone existieren dürfen, muß untersucht werden, ob Flächen mit Löcher existieren. Gegebenenfalls müssen diese Löcher durch ein neues Flächenpolygon ersetzt werden. Per Definition gilt, daß Polygone, deren Punktefolge einen mathematisch positiven Drehsinn ergeben, immer Flächen beschreiben. Ist der Drehsinn mathematisch negativ (im Uhrzeigersinn), beschreibt das Polygon den Rand eines Loches. Durch zwei (aufeinanderliegende) zusätzliche Linien ist ein Flächenpolygon in der Lage, ein *Loch*-Polygon zu einem neuen Flächenpolygon zu erweitern. Diese Verschmelzung wird in Abbildung A.5 an einem Beispiel erläutert.



(a) Ein Polygon P (Fläche) umfaßt ein Polygon L (Loch)

(b) Das neue Polygon N beschreibt eine Fläche mit einem Loch

Abbildung A.5: Verschmelzung von Polygonen

Ein Polygon P (Fläche) umfaßt ein Polygon L (Loch), wie in Abbildung A.5(a)

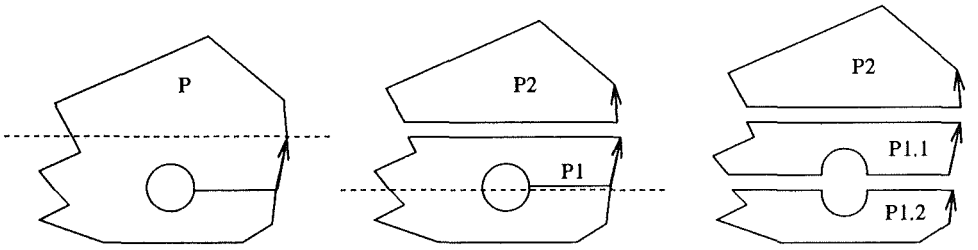
dargestellt ist. Durch Verschmelzung von P mit L wird ein neues Polygon N erzeugt. Die Reihenfolge der Punkte verdeutlicht die Folge der Polygonkanten (siehe Abbildung A.5(b)). Das neue Polygon N beschreibt eine Fläche mit einem Loch.

Vorgehensweise im Programm:

Alle Polygone einer Struktur werden auf ihre Orientierung untersucht. Polygone, die Löcher beschreiben (Orientierung mathematisch negativ), müssen mit ihrem umfassenden Polygon verschmolzen werden. Dazu wird eine Verbindungslinie vom *Loch*-Polygon zum umfassenden Polygon gezogen, dann werden alle Punkte des *Loch*-Polygon in mathematisch positiver Folge zum umfassenden Polygon hinzugefügt und zuletzt wird auf der alten Verbindungslinie eine neue in umgekehrter Richtung erzeugt. Damit sind die beiden Polygone zu einem Polygon zusammengefaßt.

2. Schritt:

Ein Polygon darf nach GDSII-Definition nicht mehr als 200 Punkte enthalten. Sind zur Beschreibung einer Fläche mehr als 200 Punkte notwendig, so muß die Fläche in mehrere kleinere Flächen zerlegt werden. Diese Vorgehensweise ist in Abbildung A.6 dargestellt.



(a) P mit mehr als 200 Punkten

(b) P1 mit mehr als 200 Punkten P2 mit 6 Punkten

(c) P1.1 mit weniger als 200 Punkten und P1.2 mit weniger als 200 Punkten

Abbildung A.6: Aufsplittung von Polygonen

Vorgehensweise im Programm:

Als Algorithmus wird folgendes Verfahren angewandt: durch die Mitte der Fläche wird eine waagrechte Linie gelegt und es werden die Schnittpunkte des umgebenden Polygons mit dieser Linie bestimmt. Die Schnittpunkte werden nach ihrer X-Koordinate sortiert. Das erste Punktepaar (z.B. 1. und 2. Punkt, $(2*n-1)$. Punkt oder $(2*n)$. Punkt ...) legt eine Schnittlinie in der Fläche fest. Ist diese Schnittlinie nicht zur gleichen Zeit auch Randlinie, so zerlegt sie die Fläche in zwei Teile. Das umgebende Polygon wird an der Schnittlinie geöffnet und es werden zwei Polygonkanten unter Beibehaltung des jeweiligen Drehsinns eingefügt. Dadurch entstehen zwei neue Polygone.

Sind diese beiden Schritte durchgeführt, so kann die Ausgabe erfolgen. Über

Optionen beim Aufruf des Programms kann als Ausgabeformat zwischen GDSII (binär, Standard), GDSII (symbolisch) und einem der Eingabe entsprechenden Format gewählt werden. Bei der symbolischen GDS II-Ausgabe werden alle Datensätze in ASCII ausgegeben.

A.4 Die Erzeugung von Vermessungsaufträgen aus dem Geometriemodell

A.4.1 Zuordnung von Bemaßungs- zu Geometrieobjekten

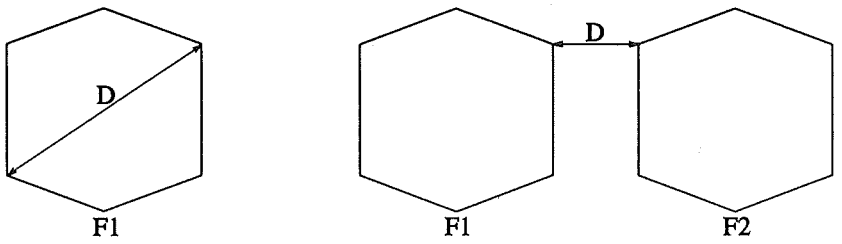
Die Erzeugung von Vermessungsaufträgen erfolgt auf Grundlage der Bemaßungseinträge, die im MCAD-System bei der Konstruktion gemacht wurden. Da bei der Übertragung ins IGES-Format die Zuordnung der Bemaßungen zu den Geometrieobjekten verloren geht, muß zunächst diese Beziehung wiederhergestellt werden. Unter Berücksichtigung der Kennung, des Anfangspunktes und des Endpunktes des Bemaßungseintrags werden die Bemaßungen in der Bemaßungsliste den Kanten, Punkten, Ecken und Figuren zugeordnet, wobei auch der Bemaßungstext zu berücksichtigen ist:

Kennung	Bedeutung	Zuordnung
L	Verifiziere Linie	Linie
S	verifiziere Kreisbogen	Kreisbogen
D	bestimme Distanz	Linie, Ecke, Punkt_auf_Kante, Figur
B	bestimme Linienbreite	zwei gerade Linien
A	überprüfe Vollständigkeit	Linie, Kreisbogen

Zunächst wird die Zuordnung innerhalb einer Figur versucht. Falls es sich um eine Linienbemaßung handelt und der Anfangspunkt sowie der Endpunkt mit der Anfangsecke und der Endecke einer Linie übereinstimmen, bzw. falls es sich um eine Kreisbogenbemaßung handelt, und der Anfangspunkt sowie der Endpunkt der Bemaßung mit dem Mittelpunkt des Kreisbogens und einem Punkt auf dem Kreisbogen übereinstimmen, kann der Bemaßungseintrag dem Geometrieelement zugeordnet werden. Mit den restlichen Bemaßungseinträgen wird eine Zuordnung zu Punkten (nach Kapitel A.2.2 sind dies alles Punkte auf Kantensegmenten) und zu Ecken versucht. Falls der Bemaßungstext eine Linienbreitenmessung (mittlerer Abstand zweier gerader Linien) spezifiziert, wird sichergestellt, daß Anfangspunkt und Endpunkt des Bemaßungseintrags jeweils Punkten auf Kantensegmenten zugeordnet werden können. Spezifiziert der Bemaßungstext eine Vollständigkeitsüberprüfung, werden für alle Kanten der Figur Verifikationsbefehle erzeugt. Falls nun immer noch Bemaßungseinträge vorhanden sind, die nicht zugeordnet werden konnten, so handelt es sich um Vermessungsaufträge zwischen Figuren, etwa der Abstandsüberprüfung zwischen zwei Figuren (Abbildung A.7). Die beschriebene Vorgehensweise wird rekursiv für alle Figuren durchgeführt.

A.4.2 Erzeugung von Aufsetzpunkten

Im Bemaßungstext wird durch Angabe einer Toleranz spezifiziert, mit welcher Genauigkeit das Messergebnis bei Längen-, Distanz- oder Linienbreitenmessung



(a) Abstandsmessung innerhalb der Figur F1

(b) Abstandsmessung zwischen den Figuren F1 und F2

Abbildung A.7: Vermessungsaufträge zwischen Figuren

bestimmt werden soll[2]. Aus der vorgegebenen Genauigkeit ergibt sich die Vergrößerung des Objektives, mit dem die (optische) Vermessung durchgeführt wird. Je höher die geforderte Genauigkeit, umso größer muß die Objektivvergrößerung sein, desto kleiner wird allerdings der zu einem Zeitpunkt unter dem Mikroskop sichtbare Ausschnitt der Geometrie, das *Bildfeld.*

Die Vermessungsaufträge in der Auftragsliste werden nach Möglichkeit so zusammengefaßt, daß mehrere Vermessungsaufträge in einem Bildfeld abgearbeitet werden können. Ein solches Bildfeld mit den darin liegenden Vermessungsaufträgen, den Koordinaten des Mittelpunktes, der Vergrößerung des Objektives und der Kantenlänge des Bildfeldes ist ein Objekt vom Typ *Aufsetzpunkt.* Es wird in der Aufsetzpunktliste der Figur gespeichert.

Falls es nicht möglich ist, alle Vermessungsaufträge einem Aufsetzpunkt zuzuordnen, werden weitere Aufsetzpunkte erzeugt. Falls ein Vermessungsauftrag nicht innerhalb eines Bildfeldes abgearbeitet werden kann, wie etwa die Verifikation einer Linie, die länger ist als die Bildfeldkanten, so wird ein rechteckiges Raster von Aufsetzpunkten erzeugt, das die gesamte Figur überlagert (siehe Abbildung A.8) Von diesen Aufsetzpunkten werden nur diejenigen in die Aufsetzpunktliste übernommen, die tatsächlich mit dem im Vermessungsauftrag referenzierten Objekt einen nichtleeren Durchschnitt haben. In Abbildung A.8 sind nicht berücksichtigte Bildfelder grau gekennzeichnet. In diesem Fall ergeben sich die *W-Ecken,* d.h. Schnittpunkte von Kanten mit dem Bildfeldrand (Abbildung A.8).

Nachdem dies für alle Vermessungsaufträge durchgeführt wurde, wird versucht die durch die Rasterung erzeugten Aufsetzpunkte mit den anderen wieder zusammenzulegen. Dies geschieht nicht nur innerhalb einer Figur, sondern über alle Figuren. Als Ergebnis entsteht eine Liste, die alle Aufsetzpunkte zur Vermessung aller spezifizierter Geometrieobjekte beinhaltet, die *globale Aufsetzpunktliste.* Diese Liste läßt sich noch bezüglich verschiedener Kriterien sortieren, unter anderem:

[2] Falls keine Angabe gemacht wird, wird ein Vorgabewert von 2% des Abstandswertes genommen, der vom MCAD System erzeugt wird.

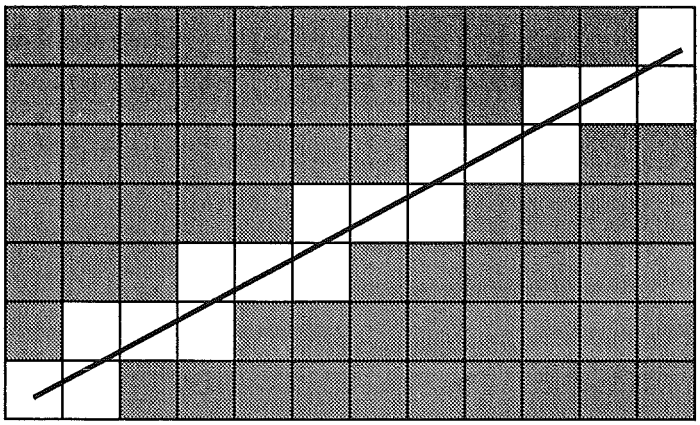


Abbildung A.8: Überlagerung einer Linie mit Bildfeldern

- Zusammenfassen aller Vermessungsaufträge, die mit der gleichen Vergrößerung durchgeführt werden können, um die Zeit die durch Objektivänderungen verloren geht (Drehung des Objektivrevolvers, Nachfokussieren) zu minimieren.
- Minimieren des Verfahrweges des Objektivtisches (Travelling Salesman Problem).
- Kombination von beidem, falls das Verfahren des Objektivtisches oberhalb einer bestimmten Entfernung länger dauert als ein Objektivwechsel.

A.4.3 Auftragserteilung an das Bildverarbeitungssytem

Eine ASCII-Datei bildet schließlich die Schnittstelle zum Bildverarbeitungssystem. Alle Aufsetzpunkte mit Position, Vergrößerung und den zugehörigen Vermessungsaufträgen werden in diese Datei geschrieben. Dabei muß die Abhängigkeit der Vermessungsaufträge untereinander berücksichtigt werden. Basis jeder Vermessung ist die Eckendetektion. Zur Verifikation einer Linie müssen vorher deren Anfangsecke und Endecke bestimmt werden.

Eine Zeile in dieser Datei besteht aus einer Anweisung (codiert mit einem Buchstaben) und der zur Durchführung dieser Anweisung notwendigen Information.

Anhang B

Arbeiten mit dem Partialmodell der Fertigungssicht

In diesem Kapitel des Anhangs wird zunächst die Architektur und anschließend die Handhabung des Systems beschrieben, das prototypisch implementiert wurde.

B.1 Die Architektur des prototypischen Systems zur Visualisierung des Partialmodells

Für die Implementierung des Prototypen zur Visualisierung des Partialmodells wurde eine Marktanalyse durchgeführt, um ein geeignetes objektorientiertes Datenbanksystem auszuwählen.

Die Anforderungen an das objektorientierte Datenbanksystem liegen in der Erfüllung des Standards ODMG-93 der Object Database Management Group, der in Kapitel 4.2 eingeführt wurde. Als Rechnerplattform eine Sun SparcStation mit dem Betriebssystem: Solaris 1.1 (mit Upgrade auf Solaris 2.1), sowie den Oberflächen MIT X11 und OpenWindows.

Grundlage der Marktanalyse bildete Literatur, die zu Systemen oder Teilfunktionalitäten veröffentlicht wurden ([BDK91, Loo91, Loo92, DeW90, Ver93]). Anhand dieser Literatur war eine Negativauswahl möglich, d.h. Systeme wurden nicht in die engere Wahl genommen, die a priori entscheidende Funktionalität nicht angeboten haben. Das Validationsergebnis war das objektorientierte Datenbanksystem VERSANT[1].

Die Gesamtarchitektur des implementierten Prototyps zur Visualisierung des Partialmodells für die Fertigungssicht ist wie in Abbildung B.1 dargestellt auf-

[1]VERSANT wird von Versant Object Technology Corporation entwickelt ([Ver]).

gebaut. Die drei Hauptkomponenten sind:

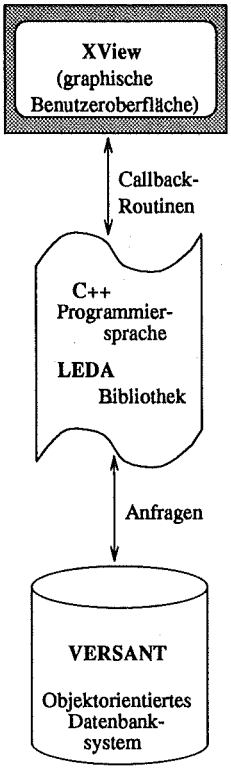


Abbildung B.1: Die Architektur des implementierten Prototyps

1. Die graphische Benutzeroberfläche **XView** [Hel].
2. Die Programme, implementiert in der Programmiersprache **C++** [Str92] unter Zuhilfenahme der Bibliothek **LEDA** [Näh92].
3. Das objektorientierte Datenbanksystem **VERSANT** (Version 2.1.4 mit C-Front 2.1).

Die Programmsteuerung erfolgt durch den Anwender, der auf der graphischen Oberfläche Aktionen (z.B. Betätigung eines Buttons, o.ä.) ausführt. Diese Aktionen verursachen Ereignisse (sogenannte Events), die von der Hauptschleife der Oberfläche ausgewertet werden. An die unterschiedlichen Events sind verschiedene Callback-Routinen gekoppelt, die beim Eintreten eines Events automatisch ausgeführt werden. Diese Callback-Routinen können direkt Aufrufe an die Datenbank enthalten, werden aber in der Regel, gemäß dem objektorientierten Paradigma der Kapselung folgend, Methoden der eigentlichen Klassen aufrufen. Diese Methoden greifen auf die in der Datenbank persistent gespeicherten Objekte zu.

Das beschaffte Datenbanksystem arbeitete über weite Strecken gesehen zufriedenstellend. Dennoch sind dazu einige Anmerkungen zu machen:

- Während der Implementierungszeit von etwas weniger als einem Jahr gab es nicht weniger als vier unterschiedliche Versionen des Datenbanksystems (2.1.3, 2.1.4, 3.0.8, 3.0.10). Dabei war der Übergang von einer Version zur nächsten keinesfalls problemlos. Besonders die Umstellung von 2.X auf 3.X hat erhebliche Änderungen erzwungen, da unter der neuen Version das Konzept von Templates unterstützt ist.
- Die Übernahme von bereits in C++ implementiertem Quellcode in den Unterstützungsbereich durch das Datenbanksystem erwies sich als komplexer als erwartet. Das lag zum einen an den Anwendungen selbst[2], zum anderen an Konzepten des Datenbanksystems[3].

Die Probleme mit dem objektorientierten Datenbanksystem sind keinesfalls systeminhärent, wie in [AASW94] bestätigt wird. Sie sind vor dem Hintergrund zu sehen, daß es sich bei objektorientierten Datenbanksystemen um sehr neue Systeme handelt, deren Funktionalität und Stabiliät oftmals nicht vollständig ausgereift ist. Es bleibt zu hoffen, daß zumindest die Einführung der Norm der ODMG [Cat93] in diesem Punkt einen Fortschritt bringt.

B.2 Die Handhabung des prototypischen Systems zur Visualisierung des Partialmodells

Es folgt die Beschreibung der zum Prototyp gehörenden Programmdateien sowie die Sequenz ihrer Verarbeitung zur Erzeugung eines ablauffähigen Programms. *Dateinamen* sind dabei kursiv geschrieben, **Kommandos** werden fettgedruckt.

Abbildung B.2 verdeutlicht den Zusammenhang der Werkzeuge mit den dabei benötigten und erzeugten Dateien.

Die gesamte Definition der Benutzeroberfläche ist in der Datei

limes.G

enthalten. Diese Datei wird ausschließlich mit dem interaktiven Oberflächenwerkzeug

guide

verändert und gespeichert. Die Datei *limes.G* wird anschließend mit dem Tool und den Parametern

[2] Eine generelle Frage bei der Übernahme von Quellcode ist, wie eine transistente Verzeigerung der Daten persistent gemacht werden kann. Das wird bei den meisten Systemen durch die Einführung von eigenen Datentypen (im vorliegenden Falle von Links) erreicht, so daß bei der Übernahme die entsprechenden Programmteile geändert werden müssen.

[3] Um eine möglichst systemunabhängige Implementierung zu erhalten, bietet es sich an, anstelle der in C++ üblichen Datentypen die vom Datenbanksystem angebotenen Datentypen mitsamt deren Operatoren anzuwenden. Das erzwingt entsprechende Änderungen im vorhandenen Programmcode.

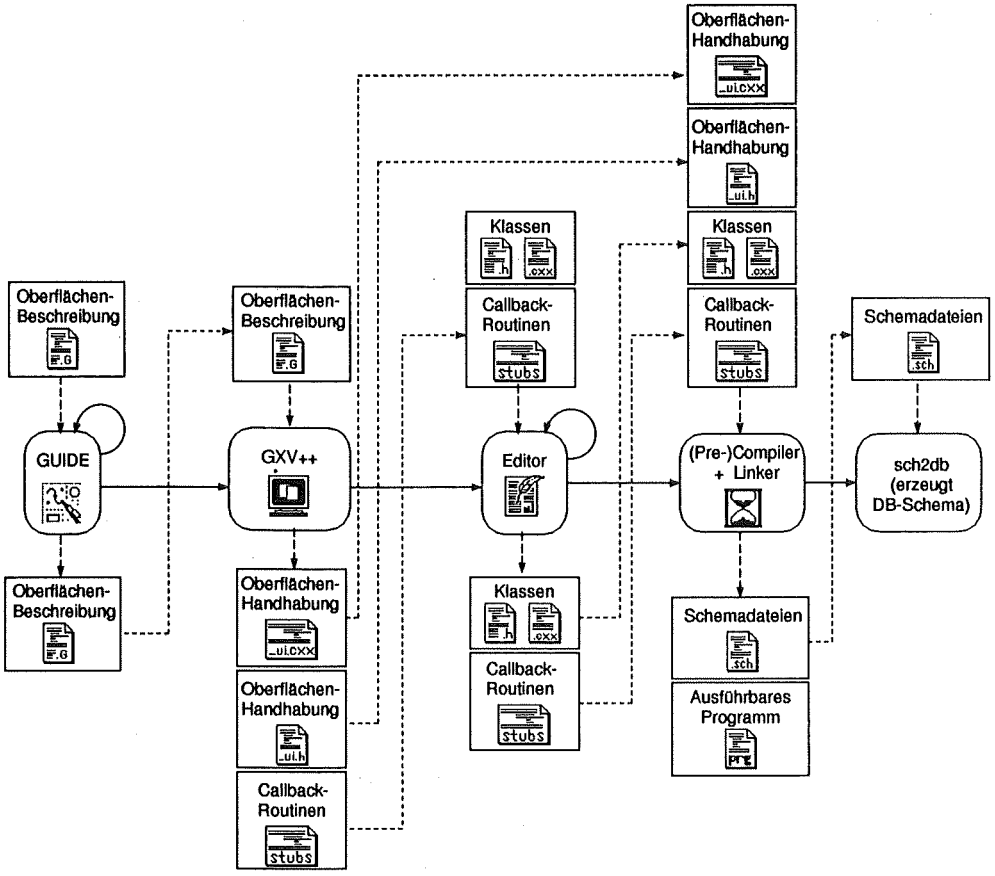


Abbildung B.2: Ablauf der Aktivitäten bei der Modifikation von Dateien für den Prototyp

gxv++ -csuffix cxx limes

in C++-Code übersetzt, der die gesamte Oberflächenbehandlung übernimmt. Das Suffix ist wichtig für den Versant-Precompiler. Die von **gxv++** erzeugten Dateien sind

limes_ui.cxx und *limes_ui.h*

Sie enthalten die C++-Routinen zur Erzeugung der Oberflächen-Objekte (Fenster, Buttons, u.ä.), sowie die C++-Methoden zum Umgang mit diesen Objekten. Die beiden Dateien *limes_ui.cxx* und *limes_ui.h* werden jedes Mal neu erzeugt, wenn die Oberflächenbeschreibung *limes.G* geändert wurde.

Eine andere von **gxv++** erzeugte Datei ist

limes_stubs.cxx

Sie enthält die benötigten Callback-Routinen, die bei der Selektion der Oberflächenobjekte automatisch gestartet werden. Diese Callback-Routinen können vom

Programmierer verändert werden. Diese Änderungen werden bei erneuter Erzeugung von *limes_ stubs.cxx* berücksichtigt, wenn nach Änderung von *limes.G* die Datei *limes_ stubs.cxx* erneut erzeugt wird.

Neben diesen Dateien, die durch die Koppelung des Programms mit der Benutzeroberfläche existieren, gibt es die eigentlichen Klassen des Systems. Sie werden im Abschnitt 5.2.1 getrennt vorgestellt.

Nachdem die Datei *limes_ stubs.cxx* erzeugt und ggfs. geändert wurde, werden die C++-Programm wie gewohnt übersetzt, allerdings unter Verwendung des VERSANT Precompilers. Dieser erzeugt für alle Klassen sogenannte Schemadateien, z.B.

kbase.sch

die benötigt werden, um die in diesen Klassen angelegte Objekte persistent in die Datenbank aufzunehmen. Für die Speicherung der Daten anhand dieser Schemadateien muß zunächst eine Datenbasis erzeugt werden, was durch die Befehle

makedb -g < *dbname* > (erzeugt eine Gruppen-DB)

createdb < *dbname* >

geschieht. Anschließend wird das Datenbankschema anhand der Information aus den Schemadateien durch den Befehl

sch2db -d < *dbname* > ***.sch**

erstellt. Für die Datenbasis muß jetzt noch über das Werkzeug

vutil -d < *dbname* >

der Sperrmechanismus (Locking) und Recoverymechanismus (Logging) aktiviert werden. Anschließend kann nach fehlerfreier Übersetzung das Programm aufgerufen werden:

limes < *dbname* >

Erster Aufrufparameter ist hierbei der Name der Datenbasis.

B.3 Programmablauf

Nach dem Aufruf des Programms erscheint zunächst das Kontrollfenster (Abbildung B.3). Dort ist zunächst der Button *Wissensbasis* gesperrt, da die Wissenbasis erst initialisiert werden muß. Das geschieht über das Pulldown-Menu *Programm.*

Abbildung B.3: Kontrollfenster

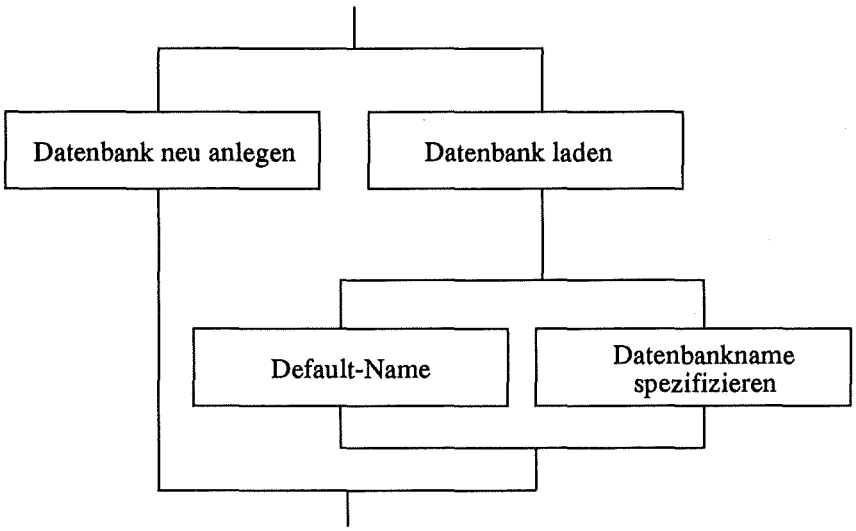


Abbildung B.4: Ablauf zum Initialisieren der Wissenbasis

Beim Initialisieren des Programms besteht die Option, entweder auf der aktuellen Arbeitsdatenbank weiterzuarbeiten oder aber eine zuvor gesicherte Datenbasis zu laden (siehe Abbildung B.4). Die Auswahl erfolgt im aufgeblendeten Initialisierungsfenster (Abbildung B.5) mit den Buttons *neu* oder *laden.* Ist das Laden einer Sicherungsdatei gewünscht, so wird im selben Fenster ein Textfeld sichtbar, in das der Anwender den Namen der Sicherungsdatei eintragen kann. Gleichzeitig wird ein Button für die Übernahme des Default-Dateinamens aufgeblendet.

Analoges Vorgehen wird beim Beenden des Programms verfolgt (Abbildung B.6). Auch dort erhält der Anwender die Option, die aktuelle Arbeitsdatenbank in einer Sicherungsdatei zu speichern, oder das Programm ohne Sicherung zu be-

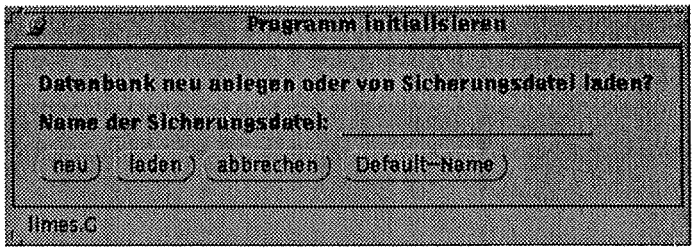


Abbildung B.5: Initialisierungsfenster

enden. Auch hier ist der Name der Sicherungsdatei spezifizierbar, bzw. der Default-Name übernehmbar (Abbildung B.7).

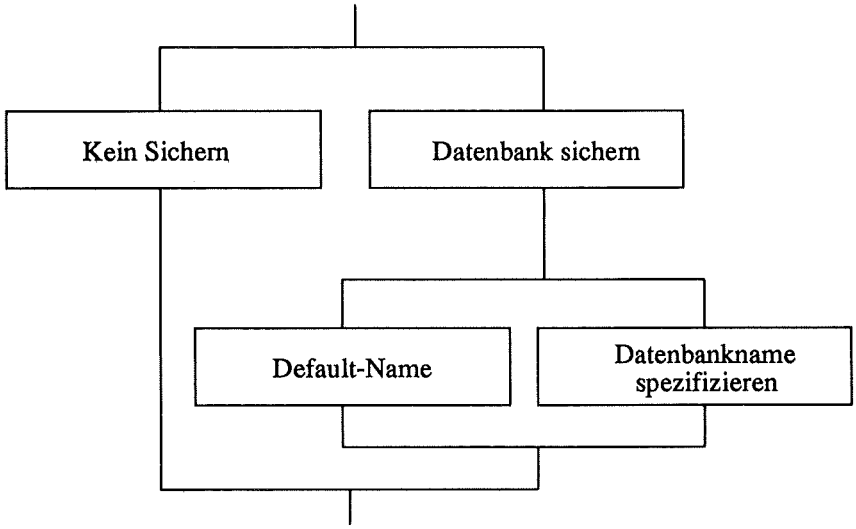


Abbildung B.6: Ablauf zum Sichern der Wissensbasis

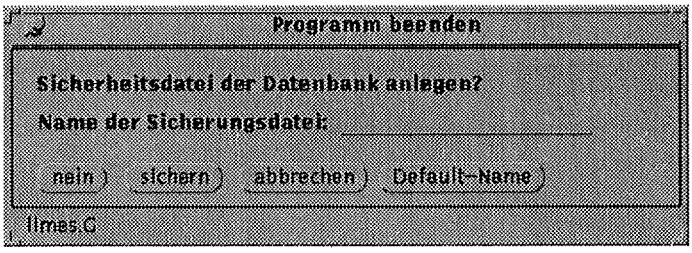


Abbildung B.7: Fenster für Sicherungsdatei

Ist die Wissensbasis initialisiert, wird der *Wissensbasis*-Button im Limes Kontrollfenster (Abbildung B.3) anwählbar. Dort hat man die Möglichkeit, die Wissensbasis zu bearbeiten oder anzuzeigen (siehe Abbildung B.9). Bei Anwahl von *Wissensbasis bearbeiten* im Pulldown-Menu wird das Fenster zur Bearbeitung der Wissensbasis (Abbildung B.8) aufgeblendet. Dort ist bisher ausschließlich

das Menu hinter dem Button *Prozeßschritt* anwählbar. Von dort aus kann man Prozeßschritt *einfügen, löschen* oder *auflisten.*

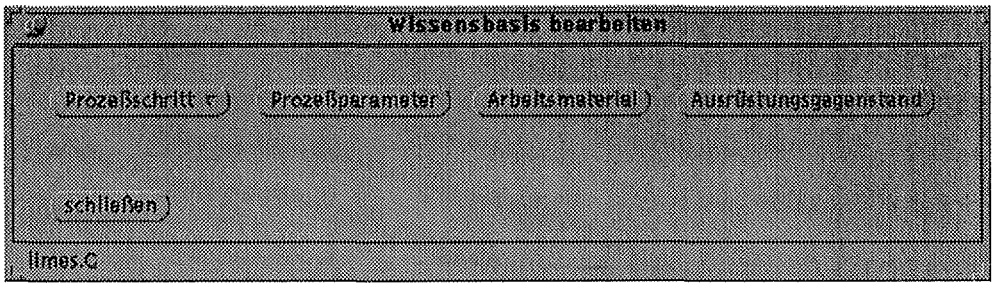


Abbildung B.8: Bearbeitungsfenster

Will man einen *Prozeßschritt einfügen*, so wird das Fenster zum Einfügen von Prozeßschritten aufgeblendet (Abbildung B.11). Dort spezifiziert der Anwender den Namen des neu einzufügenden Prozeßschritts sowie den Name des Prozeßschritts, von dem der neue Schritt eine Konkretisierung (Unteraktivität) sein soll. Die Liste der Vorgänger und die Liste der Nachfolger wird aus dem Fenster zur Visualisierung des Prozeßschrittnetzes (Abbildung B.10) durch Selektieren und anschließendes Auswählen des *Übernehmen*-Buttons spezifiziert.

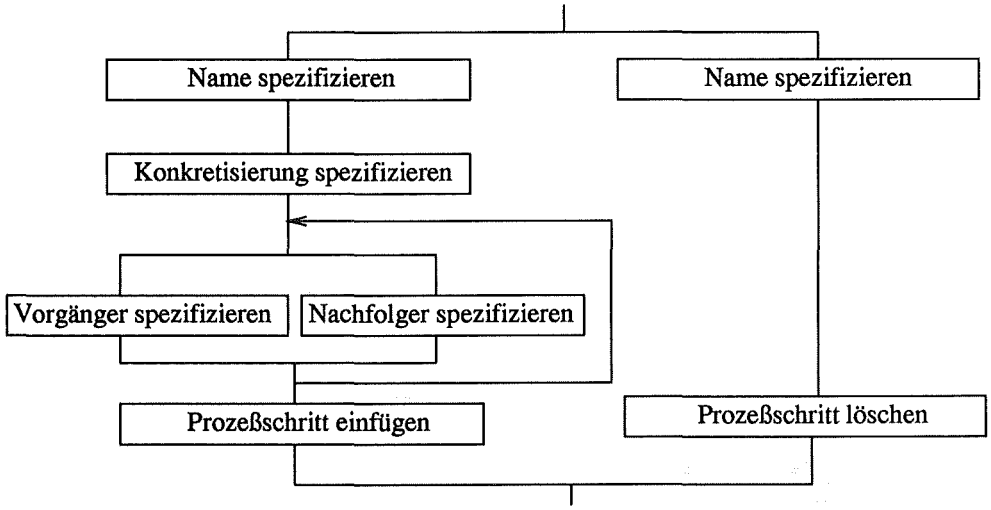


Abbildung B.9: Ablauf zum Bearbeiten der Wissenbasis

Erst durch Auswählen des *Einfügen*-Buttons wird der Prozeßschritt in dieser Ausprägung in die Datenbank gespeichert, so daß der Bearbeitungsvorgang noch abgebrochen werden kann.

Will man *Prozeßschritte löschen*, so wird das Fenster zum Löschen von Prozeßschritten (Abbildung B.12) aufgeblendet. Dort kann man einen Namen direkt eintragen oder aber einen selektierten Schritt aus dem Fenster zur Visualisierung des Prozeßschrittnetzes (Abbildung B.10) übernehmen. Erst durch auswählen des *Löschen*-Buttons wird der Prozeßschritt in dieser Ausprägung aus

Abbildung B.10: Fenster zur Visualisierung des Prozeßschrittnetzes

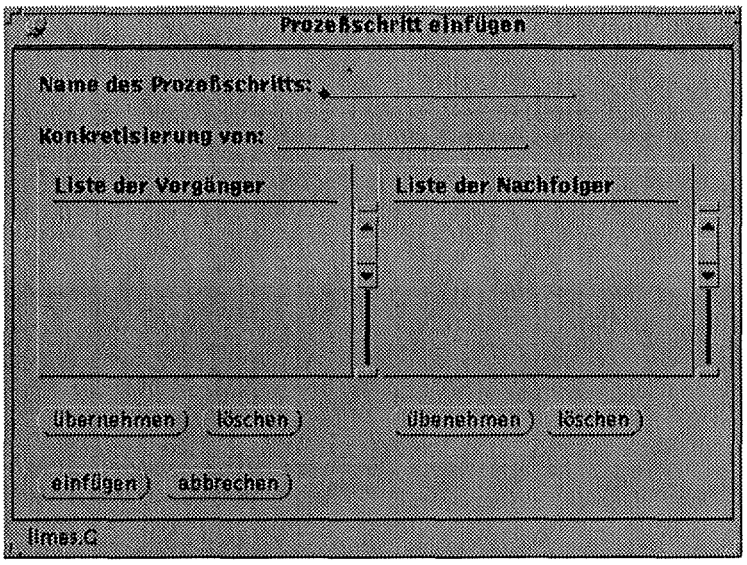


Abbildung B.11: Fenster zum Einfügen von Prozeßschritten

der Datenbank gelöscht, so daß der Bearbeitungsvorgang noch abgebrochen werden kann.

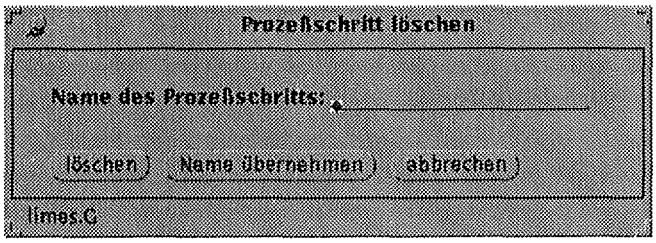


Abbildung B.12: Fenster zum Löschen von Prozeßschritten

Aus dem Fenster zur Visualisierung des Prozeßschrittnetzes (Abbildung B.10) kann zu jeden beliebigen Prozeßschritt die Veränderung der Liste seiner Attribute angestossen werden (siehe Abbildung B.13). Dazu wird das Fenster *Pro-*

zeßschritt (Abbildung B.14) aufgeblendet.

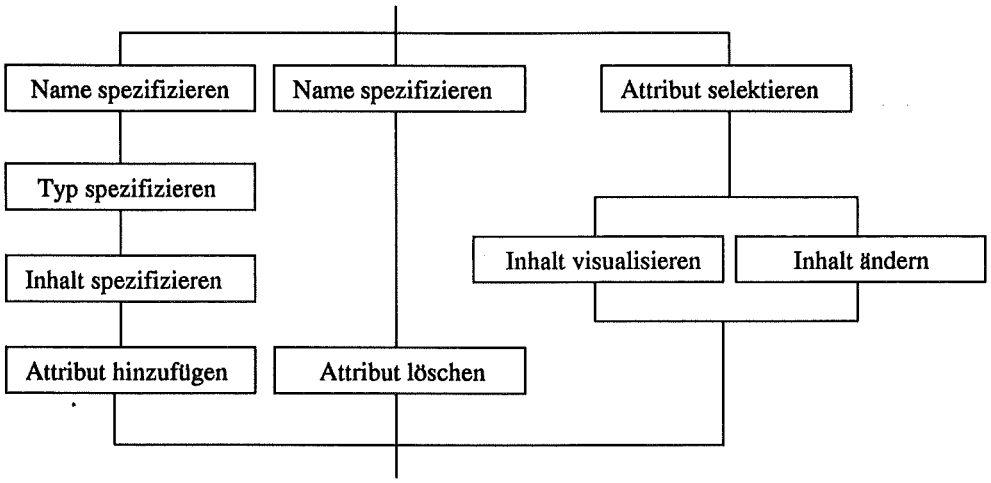


Abbildung B.13: Ablauf zum Darstellen und Ändern der Prozeßschrittinformation

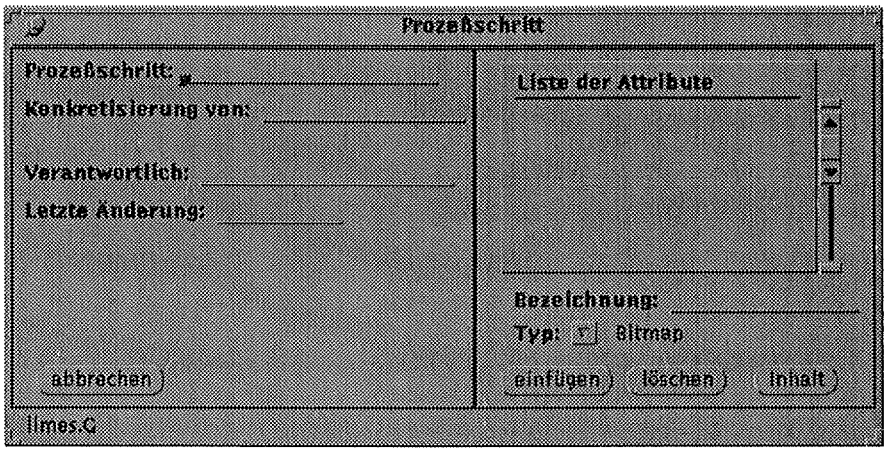


Abbildung B.14: Prozeßschrittfenster

Nach Eintragen des Prozeßschritt-Namens wird der Prozeßschritt eine Hierarchieebene höher automatisch in das Feld *Konkretisierung von* übernommen. Ebenso werden die bisher für diesen Schritt gespeicherten Attribute in die Attributliste übertragen. Durch einfaches Selektieren eines Attributs in der Liste und anschließendes Auswählen des *Inhalt*-Buttons kann der Inhalt des Attributs angezeigt und auch geändert werden. Dazu werden, je nach Attribut-Typ, das entsprechende Eingabefenster aufgeblendet.

Gleichzeitig wird bei Dokumenten und portable Bitmaps ein Werkzeug zum Anzeigen des aktuell gespeichteren Inhalts gestartet:

- Bei portable Bitmaps ist es das Werkzeug **xv**, das Bilddateien unterschiedlichen Formats anzeigen und weiterverarbeiten kann.

- Bei Dokumenten wird zur Bearbeitung als Editor ein Client-Fenster von **lemacs** gestartet. Dazu muß zuvor das Werkzeug **lemacs** gestartet werden und im Hintergrund laufen.

Beim Ändern von Bitmaps kann eine Datei im PBM-Format in die Datenbank importiert werden (Abbildung B.15), geänderte Dokumente können gespeichert werden, numerische Werte und Zeichenketten können direkt im Anzeigefenster geändert werden (für numerische Werte z.B. in Abbildung B.16), um sie anschließend persistent in der Datenbank zu speichern.

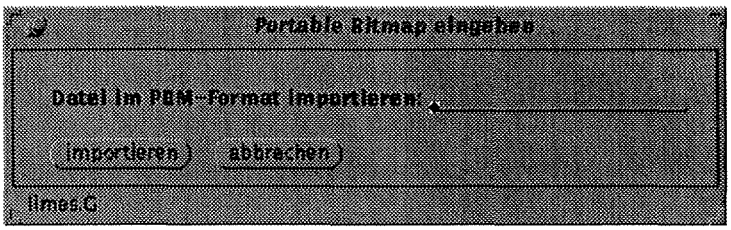


Abbildung B.15: Fenster zum Spezifizieren von portablen Bitmap-Dateien

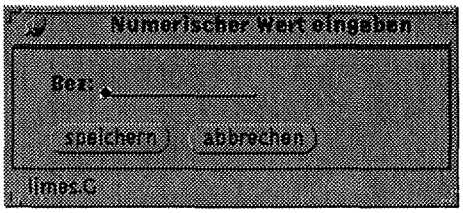


Abbildung B.16: Fenster zum Spezifizieren von numerischen Werten

Zusätzlich hat der Anwender die Möglichkeit, neue Attribute in die Attributliste *einzufügen* bzw. bestehende Attribute (und damit auch deren Inhalt) aus der Datenbasis zu *löschen*. Vor dem Eintragen von neuen Attributen muß der Name sowie der gewünschte Datentyp spezifiziert sein. Anschließend kann, für das neu hinzugefügte Attribut, der Inhalt wie oben beschrieben geändert werden.

Anhang C

Klassenbeschreibung der objektorientierten Modellierung von PRAXIS

Im Anschluß erfolgt die Beschreibung der bei der objektorientierten Modellierung von PRAXIS definierten Klassen.

Die Notation entspricht der von der ODMG [Cat93] festgelegten Norm der Objektdefinitionssprache (ODL).

C.1 Die Klasse *praxis_diagram*

```
interface praxis_diagram: praxis_object
{
  extent diagrams;
  keys name;

  attribute string name;

  relationship  praxis_page  contains
    inverse praxis_page::is_part_of

  insert_page(praxis_page );

  delete_page(praxis_page );

  load(out praxis_diagram) raises (no_such_praxis_diagram);

  save(in praxis_diagram ) raises (no_such_praxis_diagram);
}
```

C.2 Die Klasse *praxis_page*

```
interface praxis_page
{
  extent pages;
  keys name;

  attribute string name;

  relationship List<praxis_containers> display_praxis_containers
    inverse praxis_container::displayed_on

  relationship List<praxis_activities> display_praxis_activities
    inverse praxis_activity::displayed_on

  relationship praxis_diagram is_part_of
    inverse praxis_diagram::contains

  relationship praxis_hierarchical_activity has_alternatives
    inverse praxis_hierarchical_activity::alternatives

  add_container(in praxis_container);

  remove_container(out praxis_container);

  add_praxis_activity(in praxis_activity);

  remove_praxis_activity(out praxis_activity);
}
```

Hinweise:

Liste von *praxis_ activities*, die auf dieser Seite visualisiert werden.

Liste aller auf dieser Seite visualisierten *praxis_ containers* für *praxis_ things*, *praxis_ attributes* und *praxis_ parameters*.

C.3 Die Klasse *praxis_ container*

```
interface praxis_container
{

  extent containers;
  keys name;

  attribute string name;

  attribute string validity;
```

```
  attribute Type basetype

  relationship Set<praxis_thing> contains
    inverse praxis_thing::is_in

  relationship Set<praxis_container> references
    inverse praxis_container::global_container

  relationship praxis_container  global_container
    inverse praxis_container::references

  relationship praxis_page displayed_on
    inverse praxis_page::display_praxis_container

  relationship List<praxis_activity> is_used_by
    inverse praxis_activity::uses_praxis_container

  add (in praxis_thing);

  remove(out praxis_thing);

  get (out praxis_thing);

  list (out praxis_thing);

}
```

Hinweise:

validity enthält die Gültigkeit des Containers (lokal, global)

references enthält Referenzen auf Container, die diesen Container als globalen Container nutzen

global_container enthält Referenz auf globalen Container (falls vorhanden)

list() listet alle im Container enthaltenen Objekte

C.4 Die Klasse *praxis_activity*

```
interface praxis_activity
{

  extent activities;
  keys name;

  attribute string name;

  attribute string action;
```

```
attribute string executor;

attribute string time;

attribute string tools;

relationship List<praxis_activities> successors
  inverse praxis_activity::predecessors

relationship List<praxis_activities> predecessors
  inverse praxis_activity::successors

relationship List<praxis_container> uses_praxis_container
  inverse praxis_container::is_used_by

relationship List<praxis_ressources> uses_praxis_ressources
  inverse praxis_ressources::plan_used_by

relationship List<praxis_materials> uses_praxis_materials
  inverse praxis_materials::plan_used_by

relationship praxis_page displayed_on
  inverse praxis_page display_praxis_activity

add_successor(praxis_activity);
remove_successor(praxis_activity);
list_successors();

add_predecessor(praxis_activity);
remove_predecessor(praxis_activity);
list_predecessor();

add_io_object(praxis_thing);
remove_io_object(praxis_thing);
list_io_object();

add_attribute_object(praxis_variable);
remove_attribute_object(praxis_variable);
list_attribute_object();

add_parameter_object(praxis_variable);
remove_parameter_object(praxis_variable);
list_parameter_object();

add_praxis_ressources(praxis_ressource);
remove_praxis_ressources(praxis_ressource);
list_praxis_ressources();

add_praxis_material(praxis_material);
remove_praxis_material(praxis_material);
list_praxis_material();
```

```
  execute();

}
```

Hinweise:

execute() führt die *praxis_activity* aus. Diese Ausführung besteht entweder aus der Erzeugung einer *praxis_execution* (bei *praxis_executable_activity*), oder im Öffnen des Verfeinerungfensters (bei *praxis_hierarchical_activity*)

C.5 Die Klasse *praxis_executable_activity*

```
interface praxis_executable_activity : praxis_activity
{

  extent containers;
  keys name;

  attribute string name;

  relationship List<praxis_executions> executes
    inverse praxis_executions::is_executed_by

  add_praxis_executions(praxis_execution);

  execute();
}
```

Hinweise:

execute() dient zur Ausführung der *praxis_executable_activity* durch Erzeugung einer *praxis_execution*

C.6 Die Klasse *praxis_hierarchical_activity*

```
interface praxis_hierarchical_activity : praxis_activity
{

  relationship List<praxis_pages> alternatives
    inverse praxis_page::has_alternative

  attribute string selected;

  select_page(praxis_page);

  insert_page(praxis_page);

  delete_page(praxis_page);
```

```
  execute(praxis_page);

}
```

Hinweise:

execute() öffnet die Alternativseite

C.7 Die Klasse *praxis_execution*

```
interface praxis_execution
{

  extent executions;
  keys name;

  attribute string executor;

  attribute string time;

  attribute string tool;

  relationship List<praxis_things> io
    inverse praxis_things::is_io_of

  relationship List<praxis_attribute> attributes
    inverse praxis_attribute::is_attribute_of

  relationship List<praxis_parameter> parameters
    inverse praxis_parameter::is_parameter_of

  relationship List<praxis_ressources> uses_praxis_ressources
    inverse praxis_ressources::is_used_by

  relationship List<praxis_materials> uses_praxis_materials
    inverse praxis_materials::is_used_by

  relationship praxis_executable_activity is_executed_by
    inverse praxis_executable_activity::executes


  info();

}
```

Hinweise:

info() zeigt Information über diese Ausführung an

C.8 Die Klasse *praxis_things*

```
interface praxis_thing :  praxis_object
{

  extent things;
  keys name;

  attribute string name;

  relationship praxis_container is_in
    inverse praxis_container::contains

  relationship praxis_execution is_io_of
    inverse praxis_execution::io

  display();

  change();

}
```

C.9 Die Klasse *praxis_parameter*

```
interface praxis_parameter :
{

  extent parameters;
  keys name;

  attribute string name;

  attribute string value;

  relationship praxis_execution is_parameter_of
    inverse praxis_execution::parameters
  display();

  change();

}
```

C.10 Die Klasse *praxis_attribute*

```
interface praxis_attribute :
{

  extent attributes;
```

```
  keys name;

  attribute string name;

  attribute string value;

  relationship praxis_execution is_attribute_of
    inverse praxis_execution::attributes
  display();

  change();

}
```

C.11 Die Klasse *praxis_material*

```
interface praxis_material :  praxis_object
{

  extent materials;
  keys name;

  attribute string name;

  attribute string description;


  relationship praxis_execution is_used_by
    inverse praxis_execution::uses_praxis_materials

 relationship praxis_activity planned_used_by
    inverse praxis_activity::plan_uses_praxis_materials

  display();

  change();

}
```

C.12 Die Klasse *praxis_ressource*

```
interface praxis_ressource :  praxis_object
{

  extent ressources;
  keys name;

  attribute string name;
```

```
attribute string description;


relationship praxis_execution is_used_by
  inverse praxis_execution::uses_praxis_ressources

relationship praxis_activity planned_used_by
  inverse praxis_activity::plan_uses_praxis_ressources

display();

change();
```

Anhang D

Sprachbeschreibung von PRAXIS

Im folgenden wird eine Sprache für die die Konzepte von *PRAXIS* beschreiben.

Die Sprache von PRAXIS wird durch eine kontextfreie Grammatik[1] [ASU86] durch Angabe von syntaktischen Konstrukten und Ableitungsregeln formuliert. In Abschnitt D.1 wird zunächst die Grammatik der Sprache in BNF spezifiziert.

Abschnitt D.2 enthält ein kurzes Beispiel, in dem eine Sicht in der vorgestellten Sprache beschrieben wird.

D.1 Grammatik in BNF

```
( 1) <praxis_diagram>          ::=   praxis_diagram: <praxis_diagram_identifier>
                                     {
                                      <praxis_page>
                                     }

( 2) <praxis_page>             ::=   praxis_page: <praxis_page_identifier>
                                     {
                                     activities: ( <alt_praxis_activity> )
                                     containers: ( <alt_praxis_container> )
                                     }

( 3) <alt_praxis_activity>     ::=   /* empty */
                                   |
                                      <praxis_activity> <alt_praxis_activity>

( 4) <alt_praxis_container>    ::=   /* empty */
                                   |
                                      <praxis_container> <alt_praxis_container>

( 5) <praxis_activity>         ::=   praxis_activity: <praxis_activity_identifier>
                                     {
                                     executor: <identifier>
                                     execution_time: <time>
                                     execution_tool: <tool>
```

[1] Die Sprache ist zusätzlich LALR [ASU86].

successor: (<alt_praxis_activity_identifier>)
input: (<alt_praxis_container_identifier>)
output: (<alt_praxis_container_identifier>)
attribute: (<alt_praxis_container_identifier>)
parameter: (<alt_praxis_container_identifier>)
material: (<alt_praxis_material>)
ressource: (<alt_praxis_equipment>)
activates: <praxis_activity_def>
}

(6) <praxis_activity_def> ::= <praxis_hierarchical_activity>
|
<praxis_executable_activity>

(7) <praxis_hierarchical_activity> ::= **selected:** <praxis_page_identifier>
alternative: (<alt_praxis_page>)

(8) <praxis_executable_activity> ::= **execution:** (<alt_praxis_execution>)

(9) <praxis_execution> ::= **praxis_execution:** <praxis_execution_identifier>
{
executor: <identifier>
execution_time: <time>
execution_tool: <tool>
input: (<alt_praxis_thing_identifier>)
output: (<alt_praxis_thing_identifier>)
attribute: (<alt_praxis_attribute_identifier>)
parameter: (<alt_praxis_parameter_identifier>)
material: (<alt_praxis_material>)
ressource: (<alt_praxis_equipment>)
}

(10) <praxis_container> ::= **praxis_container:** <praxis_container_identifier>
{
validity: <validity>
global: <global>
basetype: <basetype>
}

(11) <validity> ::= **local**
|
global

(12) <global> ::= <praxis_container_identifier>
|
null

(13) <basetype> ::= **file**
|
geometry
|
bitmap
|
document

(14) <praxis_container_def> ::= <praxis_io_container>
|
<praxis_variable_container>

(15) <praxis_io_container> ::= **io:** (<alt_praxis_thing_identifier>)

(16) <praxis_variable_container> ::= **var:** (<alt_praxis_attribute_identifier>)
| **par:** (<alt_praxis_parameter_identifier>)

(17) <time> ::= **year:** <digit> <digit> <digit> <digit>
month: <digit> <digit>
day: <digit> <digit>
hour: <digit> <digit>
minute: <digit> <digit>
second: <digit> <digit>

(18) <tool> ::= <praxis_tool_identifier>

(19) <alt_praxis_activity_identifier> ::= <praxis_activity_identifier>
| <praxis_activity_identifier>
; <alt_praxis_activity_identifier>

(20) <alt_praxis_container_identifier> ::= <praxis_container_identifier>
| <praxis_container_identifier>
; <alt_praxis_container_identifier>

(21) <alt_praxis_material> ::= <praxis_material>
| <praxis_material>
; <alt_praxis_material>

(22) <alt_praxis_equipment> ::= <praxis_equipment>
| <praxis_equipment>
; <alt_praxis_equipment>

(23) <alt_praxis_page> ::= **/* empty */**
| <praxis_page>
; <alt_praxis_page>

(24) <alt_praxis_execution> ::= **/* empty */**
| <praxis_execution>
; <alt_praxis_execution>

(25) <alt_praxis_thing_identifier> ::= <praxis_thing_identifier>
| <praxis_thing_identifier>
; <alt_praxis_thing_identifier>

(26) <alt_praxis_parameter_identifier> ::= <praxis_parameter_identifier>
| <praxis_parameter_identifier>
; <alt_praxis_parameter_identifier>

(27) <alt_praxis_attribute_identifier> ::= <praxis_attribute_identifier>
| <praxis_attribute_identifier>
; <alt_praxis_attribute_identifier>

(28) <praxis_activity_identifier> ::= **/* empty */**
| <identifier>

(29) <praxis_attribute_identifier> ::= **/* empty */**
| <identifier>

(30) <praxis_container_identifier> ::= **/* empty */**
| <identifier>

(31) <praxis_diagram_identifier> ::= **/* empty */**
| <identifier>

(32) <praxis_execution_identifier> ::= **/* empty */**
| <identifier>

(33) <praxis_page_identifier> ::= **/* empty */**
| <identifier>

(34) <praxis_parameter_identifier> ::= **/* empty */**
| <identifier>

(35) <praxis_thing_identifier> ::= **/* empty */**
| <identifier>

(36) <praxis_tool_identifier> ::= **/* empty */**
| <identifier>

(37) <identifier> ::= **/* empty */**
| <identifier>

(38) <praxis_material> ::= **/* empty */**
| <identifier>

(39) <praxis_equipment> ::= **/* empty */**
| <identifier>

(40) <identifier> ::= **[a-zA-Z][a-zA-Z0-9_-]***

(41) <digit> ::= **[0-9]**

D.2 Beispiel

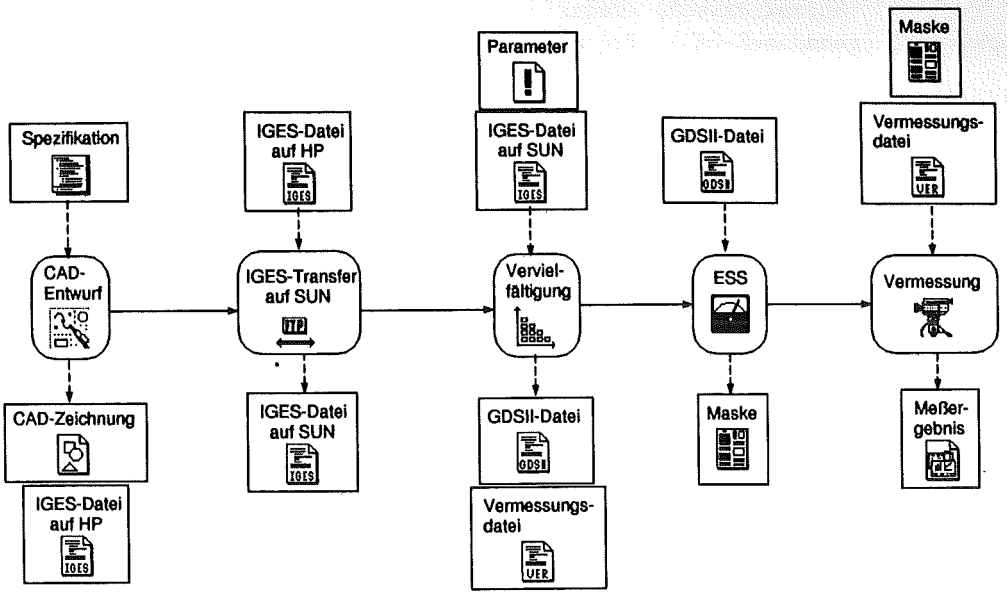


Abbildung D.1: Definiertes Beispiel (Prozeßsicht)

```
praxis_diagram: Beispielsicht {
      praxis_page : Beispielseite {
          activities: (
             praxis_activity: Vermessung {
                                    executor: Peter_Wieland
                                    execution_time: year   : 1995
                                                    month  : 02
                                                    day    : 15
                                                    hour   : 08
                                                    minute: 20
                                                    second: 00
                                    execution_tool: cosmos2d
                                    successor: ()
                                    input: (Maske; Vermessungsdatei)
                                    output: (Messergebnis)
                                    attribute: ()
                                    parameter: ()
                                    material: ()
                                    ressource: ()
                                    activates: execution: ()
                                           }
               praxis_activity: ESS {
                                    executor: Peter_Wieland
                                    execution_time: year   : 1995
                                                    month  : 02
                                                    day    : 15
                                                    hour   : 08
                                                    minute: 20
                                                    second: 00
```

```
                                execution_tool: ess
                                successor: (Vermessung)
                                input: (GDSII-Datei)
                                output: (Maske)
                                attribute: ()
                                parameter: ()
                                material: ()
                                ressource: ()
                                activates: execution: ()
                                        }
        praxis_activity: Vervielfaeltigung {
                                executor: Peter_Wieland
                                execution_time: year  : 1995
                                                month : 02
                                                day   : 15
                                                hour  : 08
                                                minute: 20
                                                second: 00
                                execution_tool: igestogdsii_script
                                successor: (ESS)
                                input: (IGES-Datei_auf_Sun)
                                output: (GDSII-Datei;Vermessungsdatei)
                                attribute: ()
                                parameter: ()
                                material: ()
                                ressource: ()
                                activates: execution: ()
                                        }
         praxis_activity: IGES_Transfer_auf_Sun {
                                executor: Peter_Wieland
                                execution_time: year  : 1995
                                                month : 02
                                                day   : 15
                                                hour  : 08
                                                minute: 20
                                                second: 00
                                execution_tool: ftp
                                successor: (Vervielfaeltigung)
                                input: (IGES-Datei)
                                output: (IGES-Datei_auf_Sun)
                                attribute: ()
                                parameter: ()
                                material: ()
                                ressource: ()
                                activates: execution: ()
                                        }
        praxis_activity: CAD_Entwurf {
                                executor: Peter_Wieland
                                execution_time: year  : 1995
                                                month : 02
                                                day   : 15
                                                hour  : 08
                                                minute: 20
```

```
                                                second: 00
                                    execution_tool: MCAD
                                    successor: (IGES_Transfer_auf_Sun)
                                    input: (Spezifikation)
                                    output: (CAD-Zeichnung; IGES-Datei)
                                    attribute: ()
                                    parameter: ()
                                    material: ()
                                    ressource: (MCAD-System)
                                    activates: execution: ()
                                            }
                            )
containers: (
     praxis_container: Spezifikation {
                                    validity: local
                                    global: null
                                    basetype: document
                                    }
     praxis_container: CAD-Zeichnung {
                                    validity: local
                                    global: null
                                    basetype: document
                                    }
     praxis_container: IGES-Datei {
                                    validity: local
                                    global: null
                                    basetype: file
                                    }
     praxis_container: IGES-Datei_auf_Sun {
                                    validity: local
                                    global: null
                                    basetype: file
                                    }
     praxis_container: GDSII-Datei {
                                    validity: local
                                    global: null
                                    basetype: file
                                    }
     praxis_container: Vermessungsdatei {
                                    validity: local
                                    global: null
                                    basetype: file
                                    }
     praxis_container: Maske {
                                    validity: local
                                    global: null
                                    basetype: file
                                    }
     praxis_container: Messergebnis {
                                    validity: local
                                    global: null
                                    basetype: file
                                    }
```

```
                praxis_container: Parameter {
                                        validity: local
                                        global: null
                                        basetype: file
                                        }
                        )
                }
        }
```

Danksagungen

An dieser Stelle möchte ich die Gelegenheit nutzen, den vielen Leuten zu danken, die ihren Teil zum Gelingen dieser Arbeit beigetragen haben.

Zunächst Dank an meine Eltern, die mir auf dem langen Weg durch das Studium und die Promotion ständig die notwendige Unterstützung zukommen ließen. Ohne ihre Unterstützung wäre diese Arbeit nie entstanden.

Für die Übernahme des Referats und der damit verbundenen fachlichen Betreuung der Arbeit möchte ich Herrn Prof. Dr. W. Menz vom Institut für Mikrostrukturtechnik (IMT) des Forschungszentrums Karlsruhe danken. Die Arbeit entstand am Institut für Angewandte Informatik (IAI) des Forschungszentrums Karlsruhe in den Jahren 1992 – 1995. Dank an den Institutsleiter, Herrn Prof. Dr. H. Trauboth für die Übernahme des Korreferats.

Mein Dank für die intensivsten Gespräche und Diskussionen über inhaltliche Aspekte der Arbeit gilt Herrn Dr. Horst Eggert, Abteilungsleiter am IAI.

Meinem Zimmergenossen und Kletterfreund Martin Goik muß ich ganz besonders danken, nicht nur für 3 Jahre mit mir am selben Schreibtisch, sondern auch für unzählige LaTeX-Tips und -installationen sowie Sicherungen im Fels bzw. Begleittransporte ins Krankenhaus.

Meinem Kollegen Klaus Lindemann möchte ich für die Versorgung mit süßen Getränken und Leckereien zu unchristlichen Stunden und an Wochenenden danken, Peter Stiller für die sichere Begleitung auf dem Nachhauseweg.

Allen anderen Mitarbeitern der Abteilung gebührt Dank für die ständige Bereitschaft zu Gesprächen, insbesondere Dr. Clemens Düpmeier und meinem Gruppenleiter Dr. Klaus-Peter Scherer für die Diskussionen über die Klassen- und Sprachbeschreibung.

Nicht zu vernachlässigen sind meine Freunde, die mich während der drei Jahre mit der notwendigen Motivation und der oft noch wichtigeren Ablenkung versorgt haben. Herausheben möchte ich hierbei Doris und Joachim Schmid, die trotz Streß im Kinderzimmer lange Abende mit der Abfassung inhaltlicher Tips und der Beseitigung von Rechtschreibefehlern zubrachten.