Modellierung und Handhabung versionierter Objekte

Christoph Hübel, Wolfgang Käfer

SFB 124
Forschungsbericht Nr. 26/89
Universität Kaiserslautern
Erwin-Schrödinger-Straße
6750 Kaiserslautern

Dezember 1989

Kurzfassung

Die Modellierung, die Handhabung und die Kontrolle von Objektversionen bestimmen in vielen Entwurfsanwendungen zusätzliche Anforderungen an die Verwaltung und Organisation der Entwurfsdaten. An einem Beispiel aus dem VLSI-Entwurfsbereich werden die wesentlichen Merkmale aufgezeigt: die "Gegenstände" der Versionenbildung sind komplex-strukturiert und teilweise überlappend; ebenso stehen sie in vielfältigen Beziehungen zueinander, die bei der Verarbeitung beachtet werden müssen. Hieran angelehnt diskutieren wir Basismechanismen zur Versionsverwaltung, die die Festlegung eines anwendungsspezifischen Versionsbegriffes unterstützen, und skizzieren eine mögliche Realisierung auf der Grundlage konventioneller Datenbanksysteme.

1. Einleitung

Ähnlich wie in betriebswirtschaftlich-administrativen Anwendungsbereichen stellen Datenbanksysteme (DBS) auch für das Gebiet der rechnergestützten Ingenieuranwendungen eine prinzipielle Lösung der anfallenden Datenverwaltungsprobleme dar. Sie gewährleisten die Sicherheit der Daten, bieten die Möglichkeit des Mehrbenutzerzugriffs sowie "hohe", anwendungsunabhängige Abfragesprachen und erlauben ein Herauslösen der Datenbeschreibung bzw. der Datenhandhabungsfunktionen aus den jeweiligen Anwendungsprogrammen.

Diesem prinzipiellen Nutzen stehen allerdings die praktischen Erfahrungen gegenüber, die beim Einsatz konventioneller DBS in sog. Nicht-Standard-Anwendungsbereichen gemacht wurden. Die Komplexität der in Ingenieuranwendungen meist vorherrschenden Objekte führt bei der Verwendung konventioneller Datenmodelle zu einer unangemessenen, unvollständigen Objektbeschreibung und damit, beim Einsatz entsprechender Datenbanksysteme, zu einer äußerst ineffizienten Objektverarbeitung. Die Komplexität der betrachteten Objekte ergibt sich zum einen aus ihrer internen Struktur: so setzen sich beispielsweise komplexe Elektronikschaltkreise wiederum aus Schaltkreisen zusammen, bzw. bestehen aus einer Vielzahl von Basisbauelementen. Diese sind zwar elementar bzgl. ihrer Zusammensetzung, dennoch können zur ihrer vollständigen Beschreibung nicht-elementare Eigenschaften erforderlich sein (z.B. das Kennlinienfeld eines Transistors). Betrachtet man zum anderen speziell den Bereich der Entwurfsanwendungen, so ist zu beobachten, daß die dort zu modellierenden Entwurfsobjekte eine zusätzliche Komplexität aufweisen: abhängig von der Organisation des jeweiligen Entwurfsprozesses können verschiedene Darstellungen eines einzigen Objektes entstehen (Versionen), deren Zusammenhang und deren Abhängigkeiten es zu beschreiben und verwalten gilt. Gerade bei großen Entwurfsaufgaben wird die Koordination dieser Objektdarstellungen zu einem eigenständigen Problem, dessen Lösung spezielle Beschreibungs- und Handhabungsmechanismen erfordert.

Auf dem Gebiet der DB-Forschung fließen diese neueren Anforderungen in die Konzeption und die Realisierung von Prototyplösungen sog. Nicht-Standard-DBS (NDBS) ein. Allerdings steht die praxisgerechte Umsetzung dieser Prototypen in kommerziell einsetzbare Systeme noch aus. In diesem Beitrag wird daher ein Ansatz der Versionsmodellierung vorgestellt, der sowohl für konventionelle Datenmodelle und verfügbare Systeme als auch für Non-Standard-Datenmodelle bzw. deren Realisierungen geeignet ist. Um die im dritten und vierten Kapitel diskutierten Konzepte an durchgängigen Beispielen illustrieren zu können, wird in Kapitel zwei zunächst ein realistischer Anwendungsfall, nämlich der VLSI-Entwurfsbereich, skizziert und die wesentlichen Begriffe erläutert.

2. Entwurfsstrukturen im VLSI-Anwendungsbereich

Mit wachsender Integrationsdichte und der damit verbundenen Komplexitätserhöhung beim VLSI-Chip-Entwurf gewinnt die Strukturierung des Entwurfsablaufes und die Entwicklung einer Entwurfsmethodologie zunehmend an Bedeutung. Die in [Zim86] beschriebene Entwurfsmethodik unterscheidet verschiedene **Entwurfsbereiche**, die jeweils durch hierarchisch angeordnete **Beschreibungsebenen** schrittweise verfeinert werden. Der Bereich "VERHALTEN" spezifiziert die Funktion (Operation, Algorithmus, Prozeß) des betreffenden Entwurfsobjektes möglichst genau. Im Bereich "STRUKTUR" wird das Objekt in seiner realisierungsunabhängigen Zusammensetzung beschrieben; er umfaßt vor allem Schaltplan, Netz- und Komponentenliste. Weitere Bereiche machen Aussagen über den konkreten Aufbau des Entwurfsobjektes. Dazu gehört ein Floorplan (Bereich "TOPOGRAPHIE"), der als Grobstruktur anschließend schrittweise bis zum Masken-Layout (Bereich "PHYSIKALISCHE REALISIERUNG") konkretisiert wird.

Für einen Entwurf sind in allen vier Bereichen hierarchisch verfeinerte Beschreibungen und Darstellungen des betreffenden Entwurfsobjektes erforderlich, die jeweils entsprechende Sichtweisen der verschiedenen Ebenen innerhalb des Bereiches auf das Objekt festhalten. Dadurch ergeben sich vielfältige Integritätsbeziehungen zwischen diesen verschiedenen Repräsentationsformen des gleichen Entwurfsobjektes. Für die Gestaltung und Optimierung des Entwurfs in den einzelnen Bereichen sowie für die Übergabe von Spezifikations- und Schnittstellendaten zwischen den Bereichen gibt es eine Vielzahl von VLSI-Werkzeugen, deren Anwendung durch Präzedenzstrukturen geregelt wird. Die Werkzeuge aktualisieren oder transformieren dabei die Objektdarstellungen. Durch Parametrisierung und interaktive Nutzung der Werkzeuge werden häufig Versionen zu einer gegebenen Repräsentationsform abgeleitet, die natürlich dadurch vom Ausgangsobjekt abhängig werden. Neben dem Begriff der Version, der in seiner Bedeutung eine Weiterentwicklung einer bestimmten Objektdarstellung beschreibt, wird in [Zim86] der Begriff der Alternative eingeführt, nach dem "gleichwertige" Objektrepräsentationen ohne einen zeitlichen Bezug als Alternativen bezeichnet werden. Es kann in allen Entwurfsbereichen und auf allen Beschreibungsebenen zur Bildung von Versionen und Alternativen kommen, so daß die Notwendigkeit von Basismechanismen zum Anlegen, Auffinden und Manipulieren von Versionen und Alternativen und zur Kontrolle der Abhängigkeitsbeziehungen evident ist.

Gegenstand der Alternativen- und Versionenbildung

Zur näheren Charakterisierung der Datenstrukturen, die den eigentlichen Gegenstand der Versionen- und Alternativenbildung ausmachen, wollen wir im folgenden die in den Entwurfsbereichen STRUKTUR und TOPOGRAPHIE vorliegenden Informationsstrukturen etwas genauer beschreiben. Der Zusammenhang zwischen der Struktur und der Topographie einer VLSI-Zelle ist im wesentlichen durch den Prozeß des **Chip-Planning** bestimmt, in dem für eine gegebene Struktur eine nach gewissen Kriterien optimale Topographie ermittelt wird [Sch88]. Als Besonderheit ist hierbei zu berücksichtigen, daß dieser Planungsvorgang (rekursiv) auf verschiedenen Hierarchieebenen abläuft, wobei wegen der Notwendigkeit, mit Schätzgrößen arbeiten zu müssen, tiefere Hierarchieebenen eine Revision und Anpassung der Planung auf höheren Ebenen erzwingen können. Die gerade in Bearbeitung befindliche Zelle heißt "cell under design" (CUD), setzt sich aus Subzellen zusammen und gehört zu einer Superzelle.

Eingabeinformation, Ablauf und Ergebnis des Chip Planning diskutieren wir anhand von Bild 2.1. Als Beschreibung aus dem Bereich STRUKTUR wird eine Komponentenliste, die aus einer Modulliste und einer Netzliste besteht, zur Verfügung gestellt. In der Modulliste werden die Moduln (Zellen) beschrieben, die innerhalb einer CUD als Subzellen zu plazieren sind. Die Modulbeschreibung enthält neben dem Namen weitere Parameter wie u.a. die voraussichtliche Modulfläche und die Außen-/Innenanschlüsse. Typischerweise existieren für die Modulflächen lediglich Abschätzungen, bzw. Flächenfunktionen [KSZ87], die die möglichen Flächenformen und Dimensionen eines Moduls beschreiben. Außen- bzw. (beim Obermodul) Innenanschlüsse werden aufgrund ihrer Funktionalität zu Anschlußbündeln, den sog. Pins, zusammengefaßt. In der Netzliste wird die Struktur der Zelle, d.h. die Verbindungen der Subzellen untereinander beschrieben. Das sogenannte Netzgewicht dient als Maß für die Breite der Verbindung und muß stets mit der Breite der angeschlossenen Pins übereinstimmen. Zur genaueren Beschreibung der Einbettung der Subzellen in die CUD, d.h. also der Verbindung der Subzellen-Pins mit den Pins der CUD, sind Modul- und Netzliste um die entsprechenden Einträge erweitert.

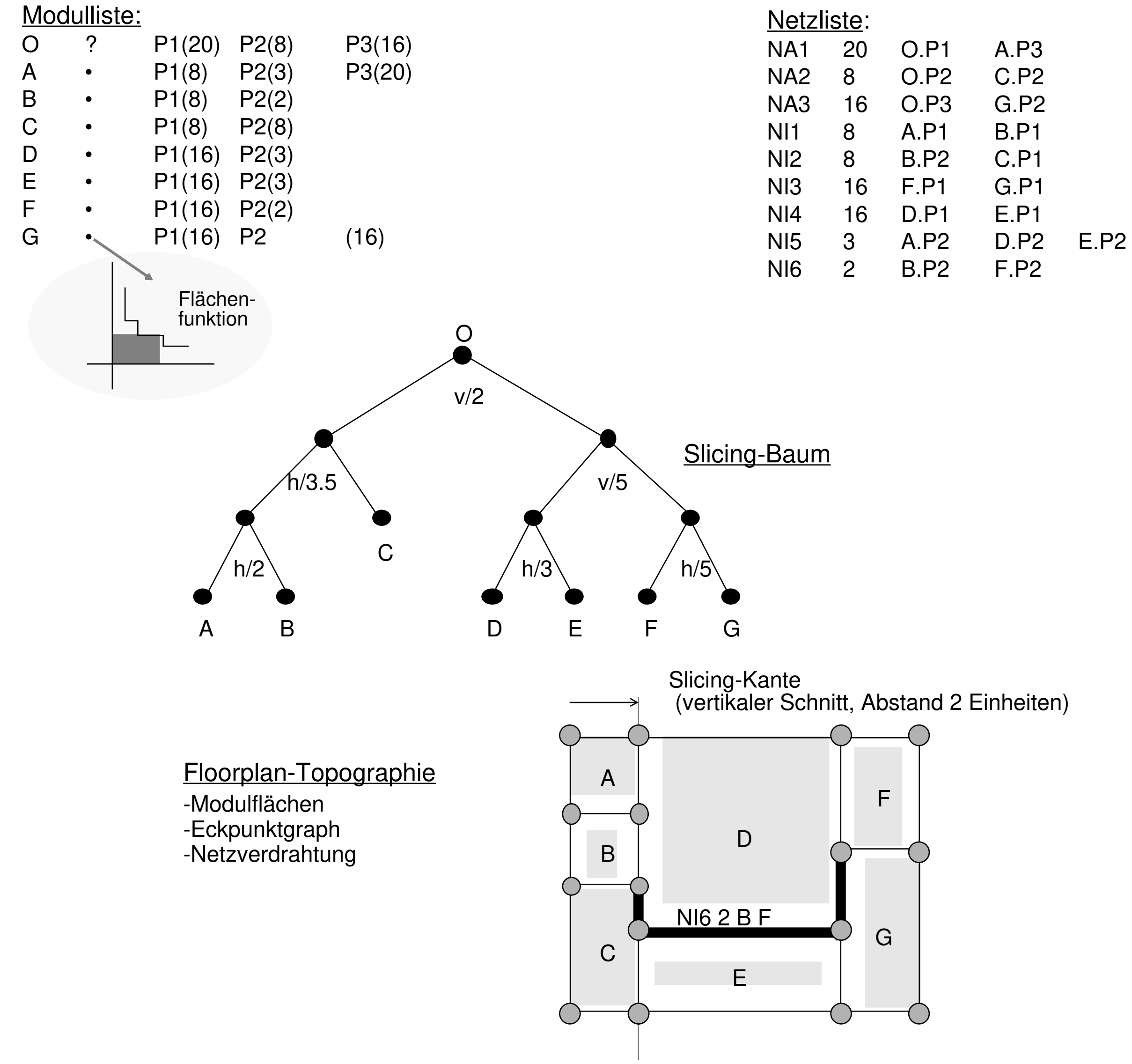

Bild 2.1: Arbeitsschritte des Chip Planning

Im Rahmen des Chip Planning wird zunächst eine Floorplan-Topologie generiert, die durch einen Slicing-Baum dargestellt wird. Die dazu benötigten Algorithmen versuchen Kostenfunktionen zu minimieren, deren Parameter aus Modulfläche und Netzlängen bestehen. Hieran anschließend wird ebenfalls über spezielle Heuristiken eine Horizontal/Vertikal- bzw. eine Rechts/Links-Orientierung der Subzellen ermittelt und in Form zusätzlicher Knotenattribute im Slicing Baum abgelegt. Der so erweiterte Slicing-Baum beschreibt die vorläufige Gestalt des zu ermittelnden Floorplans. Hierauf aufbauend wird nun eine globale Verdrahtung der Subzellen gemäß der vorgegebenen Netzliste durchgeführt. Dazu wird zunächst ein sog. Eckpunktgraph abgeleitet. Eine Ecke ist dabei durch genau zwei Slicing-Kanten bestimmt. Im Anschluß hieran erfolgt die eigentliche Verdrahtungsplanung, durch die einem Netz eine Folge von Verdrahtungsstrecken zugeordnet wird. Die einzelnen Verdrahtungsstrecken werden durch zwei benachbarte Ecken begrenzt und ggf. an einen Pin oder eine weitere Verdrahtungsstrecke angeschlossen. Die Verdrahtungsstrecken mit gemeinsamen Ecken bilden einen Kanal. Eine Abschätzung der Kanalbreite und damit der Kanalgeometrie kann durch Summation der über die Verdrahtungsstrecken assoziierten Netzgewichte ermittelt werden. Die so berechnete Kanalfläche geht dann entweder als eigenes Flächenstück in die Flächenbedarfsberechnung der CUD ein oder wird anteilig den Flächen der angrenzenden Subzellen zugeschlagen. Da die Planung zunächst mit Schätzgrößen begonnen wurde, ist nun, nachdem eine genauere Zellgeometrie bekannt ist, eine iterative Optimierung erforderlich.

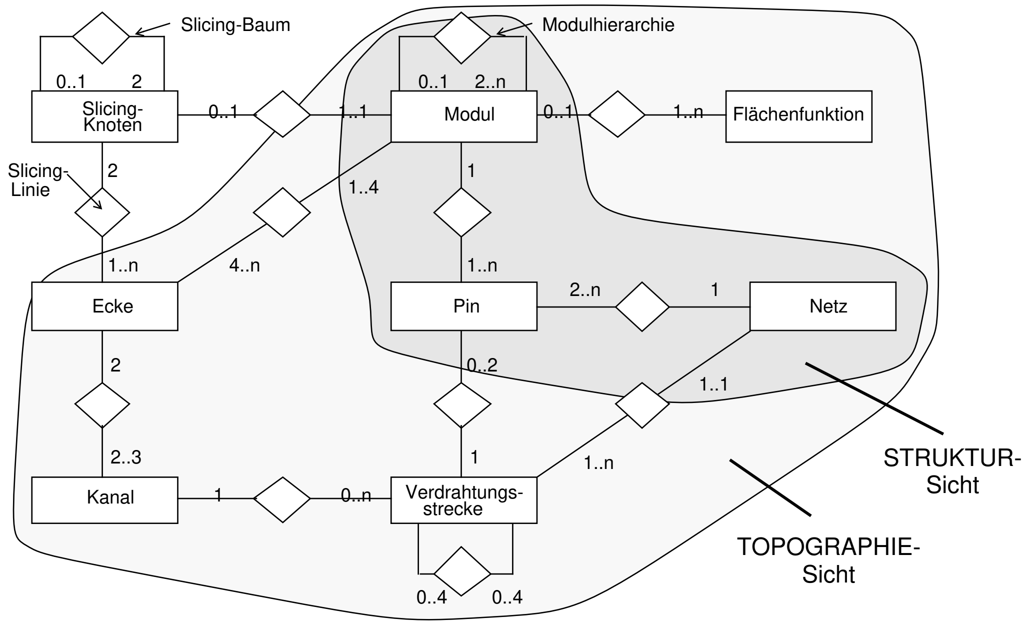

Bild 2.2: E/R-Diagramm der Informationsstrukturen für das Chip Planning

Bild 2.2 stellt die informell beschriebenen Informationsstrukturen der Entwurfsbereiche STRUKTUR und TOPOGRAPHIE und damit die Ein- und Ausgabeobjekte der entsprechenden VLSI-Entwurfswerkzeuge in Form eines E/R-Diagramms dar. Eine Zuordnung der einzelnen Entity-Typen zu den Entwurfsbereichen ist anhand der Hintergrundkennzeichnung möglich. Es wird deutlich, daß zum einen die Informationsstrukturen und damit die Objektrepräsentationen der einzelnen Entwurfsbereiche überlappen, und daß zum anderen diejenigen Objektrepräsentationen, zu denen Versionen und Alternativen entstehen, eine erhebliche strukturelle Komplexität aufweisen. Die "Gegenstände" der Versions- und Alternativenbildung lassen sich demnach als "Sichten" auf ein entwurfsbereichsübergreifendes konzeptuelles Schema auffassen.

Konkrete Ausprägungen dieser Sichten bestimmen komplexe Objekte, die alle diejenigen Daten umfassen, die für ein Entwurfswerkzeug in einem bestimmten Zusammenhang benötigt werden, d.h., sie bilden die Einheiten der Verarbeitung. Die Verarbeitung selbst geschieht durch die Werkzeuge, die ein oder mehrere Objekte aus der DB extrahieren, um diese zu verändern oder neue Objekte abzuleiten. Verfeinert man diese Sicht auf **Objektversionen**, so müssen diese gelesen und geändert werden können. Weiterhin müssen neue Versionen angelegt und Versionen von anderen Objekten erzeugt werden können. Dabei wird vorausgesetzt, daß jede Version das Objekt vollständig (aus Sicht der verarbeitenden Werkzeuge) beschreibt, da ja nach der Versionierung eines Objektes die Objektversion den Gegenstand der Verarbeitung darstellt. Die Ableitung von Objektversionen aus Versionen anderer Objekte kann häufig parametrisiert werden, so daß eine Menge von unabhängigen Versionen des abgeleiteten Objekts entsteht. Jede dieser Versionen, die im weiteren als **Objektalternativen** bezeichnet werden, kann anschließend durch Werkzeuge weiterentwickelt werden. Diese Weiterentwicklung führt zu Objektversionen des gleichen Objektes. Die dabei entstehenden Beziehungen werden in Form eines Abstammungsgraphen beschrieben. Bezogen auf Bild 2.3 kann ein Chip-Planning-Werkzeug (W_1) anhand von Parametern (Form und Ausdehnung der umgebenden Zelle) aus einer Komponentenliste (Objekt 1) eine Anzahl alternativer Floorplan-Topographien (Objekt 2) erzeugen. Anschließend kann der Floorplan beispielsweise mit einem Graphikeditor (W_2) manuell weiter optimiert werden. Dabei entsteht eine Abstammungsbeziehung ("entwickelt aus") zwischen den Versionen des Floorplans. Zwischen den Objekten (der STRUKTUR-Sicht und der TOPOGRAPHIE-Sicht) muß auch eine Beziehung (z.B. "abgeleitet aus") verwaltet werden, die ggf. auf Objektversionen verfeinert werden kann. Die Beziehungen zwischen Objekten, Objektversionen oder Objektalternativen sind sehr vielfältig. Zum einen spiegeln sich diese Beziehungen auf Schemaebene wider, zum anderen kommen sie erst auf der Ebene der Werkzeug-Ein-/Ausgabeobjekte zu tragen: so

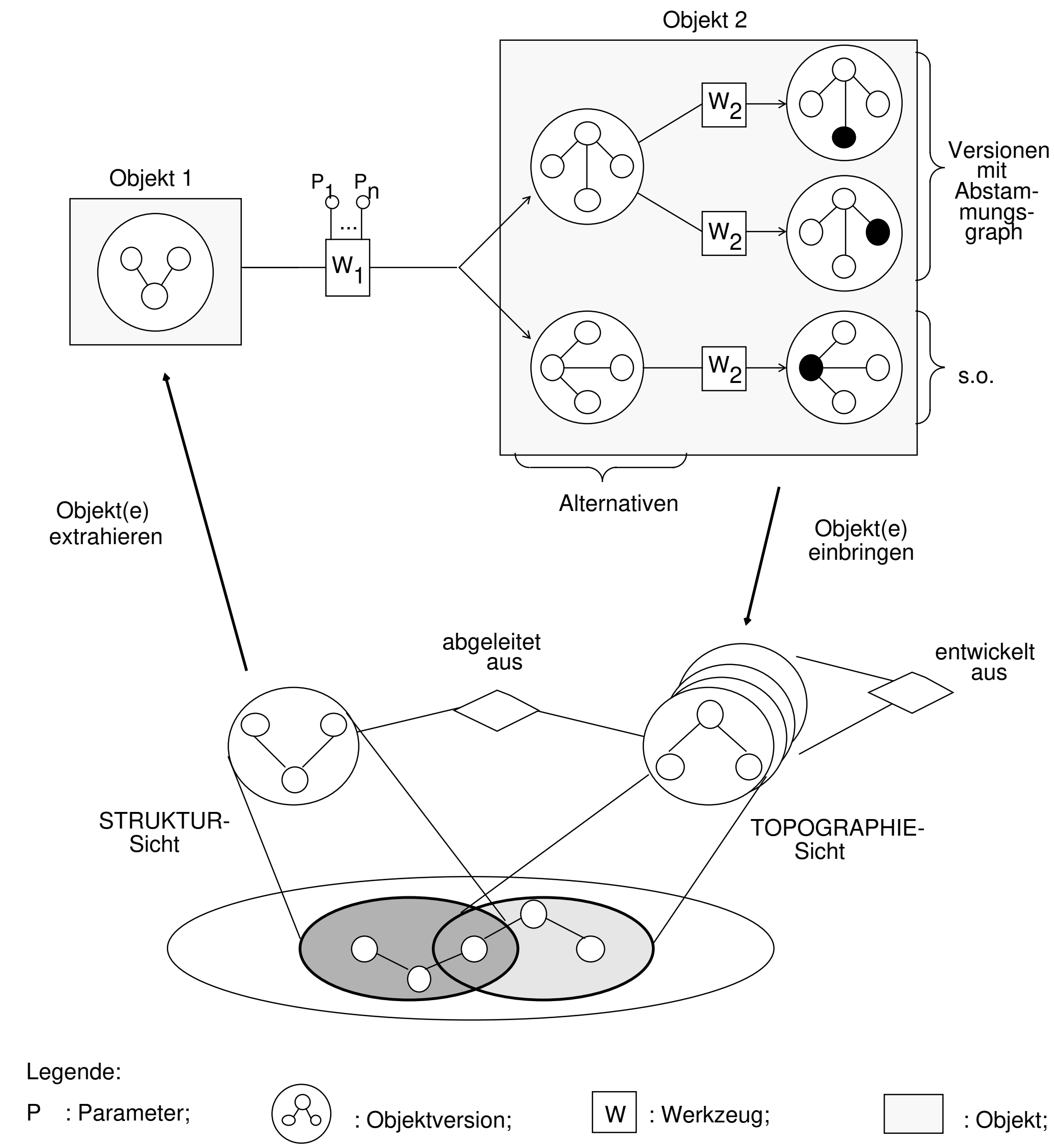

Bild 2.3: Objektversionen als Sichten auf einem konzeptuellen Schema

stehen Objektversionen beispielsweise in einer "abgeleitet-aus" Beziehung, die sich im Prinzip aus der Anwendung eines entsprechenden VLSI-Entwurfswerkzeuge ergibt. In natürlicher Weise existieren Wechselwirkungen zwischen Beziehungen, die auf Schemaebene implizit existieren (Überlappungen), und Beziehungen, die auf Objektversionen explizit definiert sind (abgeleitet-aus, ist-verträglich-mit, ...). Diese Wechselwirkungen sind bei allen Operationen auf Objektversionen/-alternativen zu beachten.

Insgesamt können aus dem hier beschriebenen Anwendungsscenario die folgenden Anforderungen an eine Basisversionsverwaltung abgeleitet werden:

- aufbauend auf einem konzeptuellen Schema müssen die Gegenstände der Versionen- und Alternativenbbildung als komplexstrukturierte Sichten beschreibbar sein,
- darüber hinaus muß es möglich sein, Beziehungen zwischen versionierten Objekten zu spezifizieren; insbesondere sollten Mechanismen angeboten werden, die eine Kontrolle der Wechselwirkungen zwischen expliziten und impliziten Beziehungen ermöglichen.

3. Ein Basisversionsmodell

Das in Kapitel 2 implizit unterstellte "Versionsverständnis" ist relativ speziell und stimmt in vielerlei Hinsicht nicht mit dem Versionsbegriff anderer Anwendungsbereiche überein. Unser Ziel ist es daher zunächst ein Basisversionenmodell zu entwickeln, von dem ausgehend die unterschiedlichen Versionsbegriffe unterstützt werden können. Die in der Literatur vorgeschlagenen Versionsmodelle, z.B. [Bato85, Blan87, Chou86, Ditt88, Katz84, Katz86, Klah86] leisten dies in der Regel nicht. Teilweise sind sie zu speziell auf einen bestimmten Anwendungsbereich zugeschnitten und kommen daher nicht als ein Basismodell in Frage. Andere erlauben keine adäquate Behandlung der Beziehungen zwischen Objektversionen oder sehen eine zu enge Verknüpfung mit dem Transaktionsbegriff vor. Unser Ansatz in Form eines Kern- bzw. Erweiterungssytems stellt Basisoperationen zur Modellierung und Handhabung von Versionen und Alternativen zur Verfügung, die dann zu einer der jeweiligen Anwendung angepaßten Versionsverwaltung ausgebaut bzw. kombiniert werden können. Hierzu führen wir das Konzept der Versionsklasse ein, wobei diese so beschrieben werden kann, daß sie den anwendungsspezifischen Versionsbegriff widerspiegelt. Dieser Ansatz erscheint besonders deshalb vielversprechend, weil er zum einen auch auf konventionellen und damit auf dem Markt befindlichen Systemen realisierbar ist und zum anderen eine Fortführung des "Erweiterungsgedankens" darstellt: viele Prototypen der in der Forschung befindlichen NDBS [Bato87, Bato88, Care86, Care87, Härd88, McPh87, Sch87, Schö89] basieren auf einem Baukastensystem oder einer Kernarchitektur, die durch Zusatzebenen an die jeweiligen Anwendungsbereiche angepaßt werden müssen.

3.1 Versionierte Objekte

Aus den in Kapitel 2 beschrieben Anforderungen ergeben sich zusammenfassend folgende Eigenschaften von Objekten und Versionen:

- Objekte:
  - sind durch Ausschnitte aus dem DB-Schema bestimmt,
  - sind Einheiten der Verarbeitung,
  - können mit anderen Objekten verbunden sein,
  - können überlappen,
  - sind durch Attribute beschrieben,
  - sind eindeutig identifizierbar,
  - bestehen aus Versionen.
- Versionen:
  - beschreiben eine im Sinne des Datenmodells vollständigen Ausschnitt des Schemas,
  - erfüllen damit die durch die Schemabeschreibung definierten Integritätsbedingungen,
  - sind innerhalb des Objekts eindeutig identifizierbar,
  - gehören zu genau einem Objekt,
  - sind in einem Abstammungsgraph geordnet.
- Alternativen
  - sind Versionen, die im Abstammungsgraph keine Vorgänger besitzen.

Dieser rudimentäre Alternativenbegriff erfüllt die in Kapitel 2 gestellten Anforderungen: Alternativen stellen unabhängige, parallel nebeneinander existierende Lösungen dar. Jede Alternative kann weiterentwikkelt werden und bildet somit die Wurzel eines Abstammungsgraphen. Insgesamt gesehen besteht ein Objekt aus einem Wald von Abstammungsgraphen.

Die Definition eines versionierten Objekts ist in Bild 3.1 mit Hilfe einer vereinfachten Syntax dargestellt. Die CLASSES-Klausel dient zur Beschreibung möglicher Versionsklassen und wird in Abschnitt 3.2 ausführlich behandelt.

```
DEFINE <object> AS <Ausschnitt des DB-Schemas> VERSIONED
(    <attribute>       : TYPE,
                       :      Definition der Objekt beschreibenden Attribute
     <attribute>       : TYPE )
DERIVATION IS LIST / TREE / DAG
     (* Abstammungsgraph als Liste,Baum oder azyklischer Graph *)
NUMBER OF ALTERNATIVES IS <0..*>   (*Anzahl der möglichen Alternativen *)
CLASSES ARE <classes>.   (* Aufzählung der möglichen Versionsklassen *)
```

Bild 3.1: Definition eines versionierten Objekts

Zur Beschreibung des "Ausschnitts des DB-Schemas" können die Sprachklauseln des zugrundeliegenden DBS verwendet werden. In Bild 3.2, das die Definition der Objekte entsprechend zu Bild 2.2 aufzeigt, verwenden wir eine intuitive Schreibweise. Wir zählen alle Entity-Typen auf, die das Objekt konstituieren. Eventuell notwendige Rollennamen werden den Entity-Typen vorangestellt.

```
DEFINETopographie AS
Ober(Modul), Unter(Modul), Pin, Netz   VERSIONED
( . . .  Definition der objektbeschreibenden Attribute  . . . )
DERIVATION IS TREE
NUMBER OF ALTERNATIVES IS <0 . . *> ;

DEFINEStruktur AS
Ober(Modul), Unter(Modul), Flächenfunktion,
Pin, Netz, Super(Verdrahtungsstrecke), Sub(Verdrahtungsstrecke,
Kanal, Ecke   VERSIONED
( . . .  Definition der objektbeschreibenden Attribute  . . . )
DERIVATION IS LIST
NUMBER OF ALTERNATIVES IS <0 . . 0> ;
```

Bild 3.2: Definition des Topographie- und Strukturobjekttyps.

Topographie-Objekte können beliebig viele Alternativen besitzen. Jede Alternative kann wiederum einen baumartigen Abstammungsgraphen aufspannen (vgl. Bild 2.3, Objekt 2). Struktur-Objekte hingegen besitzen keine bzw. genau eine Alternative. Die Versionen sind linear geordnet.

Zur Handhabung der versionierten Objekte werden vier Gruppen von Operatoren benötigt:

- Operationen auf Objekten

  Es werden Operationen zum Erzeugen, Löschen und Selektieren von Objekten benötigt. Bei der Erzeugung des Objekts werden die objektbeschreibenden Attribute mit Werten versorgt, die über die Lebenszeit des Objekts nicht geändert werden können. Es muß keine Version des Objekts erzeugt werden. Die Selektion basiert entweder auf den objektbeschreibenden Attributen oder auf der Selektion von Versionen des Objekts.
- Operationen auf Versionen

  Es müssen Versionen erzeugt, gelöscht, geändert und selektiert werden können. Die Selektion einer Version kann auf das zugehörige Objekt eingeschränkt werden; es werden gesonderte Operationen zur Selektion von Versionen über den Abstammungsgraphen zur Verfügung gestellt.
- Operationen auf dem Abstammungsgraph

Eine Version muß in den Abstammungsgraphen eingegeliedert oder wieder entfernt werden können. Dabei kann die Beschreibung der Vorgänger/Nachfolger deskriptiv über die Selektionsoperationen erfolgen.

- Operationen auf Klassen

  Versionen müssen den Klassen zugeordnet bzw. aus Klassen entfernt werden können. Weiterhin ist eine Operation zum Testen der Klassenzugehörigkeit bereitzustellen.

```
CREATE   OBJECT <data> FROM <object>
         erzeugt ein Objekt (ohne Versionen) und gibt die Objekt-Id zurück.

<object_selection> :=
         ID IS <object-id> /
         <Suchausdruck über den objektbeschreibenden Attributen> /
         VERSIONS <version_selection>(* jede Version gehört zu genau einem Objekt *)

<version_selection> :=
         ID IS <version-id> /
         <Suchausdruck über den Versionen eines Objekts> /
         <über die Position in einem Graph>

DELETE   FROM <object> WHERE <object-selection>
         löscht das Objekt mit allen Versionen.

SELECT   OBJECT / VERSION <projection> FROM <object> WHERE <object_selection> / <version_selction>
         selektiert Objekte (mit den objektbeschteibenden Attributen aber ohne Versionen) bzw. Versionen.

CREATE   VERSION <data> FROM <object> WHERE <object_selection>
         erzeugt eine neue Version des Objekts und gibt die Versions-Id zurück.

UPDATE   VERSION <data> FROM <object> WHERE <object_selection>
         ändert die Daten einer existierenden Version.

CONNECT <object> TO <graph> ID IS <version_id>
         PREDECESSORS ARE      <version_id_list> /
                               <version_selection>
         SUCCESSORS ARE        (* s.o. *)

DISCONNECT <object> FROM <graph>WHERE <version_selection>
         es werden alle Vorgänger mit allen Nachfolgern verbunden.

PREDECESSORS OF <object> <version_selection> FROM <graph>      LEVELS ARE <1..MAX> /
                                                               version_selection
selektiert alle Vorgänger im angegeben Graph. Mit LEVEL = 1 werden die direkten Vorgänger, mit LEVEL = MAX die Wurzeln
und mit LEVEL = 1.. MAX alle Vorgänger selektiert.

SUCCESSORS ... (* analog PREDECESSOR *)

ASSIGN   <object> <version_selection> TO <class>;

        Versionen eines Objekts werden einer Klasse zugewiesen. Dabei werden die mit der Klasse definierten Konstistenzbe-
        dingungen überprüft.

REMOVE ... (* analog ASSIGN *)
```

Bild 3.3: Operationen für versionierte Objekte

In Bild 3.2 sind die Operationen in einer vereinfachten Syntax dargestellt. Sie bilden die Basis zur Handhabung von Objekten und Objektversionen. Im folgenden sollen nur die Operationen zur Manipulation von Versionen kurz erläutert werden. Bei der Erzeugung einer Version müssen alle Daten, die diese Version konstituieren sowie alle Versionen, von denen diese Daten abgeleitet wurden angegeben werden. Bleibt dabei die Struktur des Abstammungsgraphen konsistent, wird die Differenz (vgl. Kapitel 4) der neuen Version zur zeitlich neuesten Version ihrer Vorgänger berechnet und abgespeichert. Im Gegensatz zu temporalen Anwendungen muß davon ausgegangen werden, daß die Werkzeuge in Ingenieuranwendungen iterativ arbeiten, d.h., nicht jede Änderung der Daten darf zu einer neuen Version führen. Die Operation UPDATE VERSION erlaubt deshalb die Änderung von Daten einer (bereits gespeicherten) Version. Sind die geänderten Daten in mehreren Objekten enthalten (vgl. Bild 2.3), d.h. die Objekte überlappen, so muß für jedes einzelne der berührten Objekt angegeben werden, ob die Änderung in der aktuellen Version (dieses Objekts) durchgeführt werden soll oder ob eine neue Version des Objekts zu erstellen ist.

Angewendet auf das Beispiel aus Kapitel 2 (Bild 2.3) sind die in Bild 3.4 vorgestellten Operationen auszuführen. Dabei wird angenommen, daß die Struktur schon in der Datenbank vorhanden ist. Die Topographie hingegen muß erst erzeugt werden.

Lesen der neuesten Version der Struktur des Addierers:
SELECT OBJECT ALL FROM Struktur
WHERE Name = "Addierer" ⇒ Struktur_1

SELECT VERSION ALLFROM Struktur
WHERE V_id = TOPICAL⇒ V_7

Erzeugen des Topographie-Objekts:
CREATE OBJECT Name = "Addierer", . . . FROM Topographie ⇒ Topographie_1

Ausführung des Werkzeugs W_1 und erzeugen der ersten Version der Topographie:

CREATE VERSION <data> FROM Topographie
WHERE Top_id = Topographie_1 ⇒ V_1

Lesen der 1. Version und anschließendes ausführen des Werkzeugs W_2 :

SELECT VERSION ALLFROM Topographie
WHERE Top_id = Topographie_1 and V_id = V_1

Erzeugen einer 2. Version der Topographie:
CREATE VERSION <data> FROM Topographie
WHERE Top_id = Topographie_1 ⇒ V_2

Die neue Version wird als Nachfolgeversion von V_1 eingetragen:
CONNECTTopographie TO DERIVATION ID IS V_2
PREDELESSORS ARE V_1

Bild 3.4: Operationsfolge zum Erzeugen von Objekten und Versionen.

Anschließendes Ausführen von Werkzeug W_1 (analog oben) liefert eine neue Version (V_3) der Topographie. Diese Version ist unabhängig von den bisherigen Versionen (V_1, V_2) und bildet somit eine Alternative. Dies wird dadurch gekennzeichnet, daß sie mit keinen anderen Versionen der Topographie verbunden wird.

3.2 Unterstützung von anwendungsspezifischen Versionsbegriffen

Die Frage nach der Semantik von anwendungsspezifischen Versionen ist die Frage nach den Bedingungen, die erfüllt sein müssen, damit eine Menge von Daten (die ein Objekt beschreiben) im Sinne der Anwendung als Version bezeichnet wird. Darüberhinaus ist das aktuelle Versionsverständnis auch vom Standpunkt des Betrachters abhängig. Der Entwerfer eines Objekts möchte vielleicht die Daten "vor dem

Mittagessen" und "nach Freitag dem 13." als Versionen zur Verfügung haben, während sein Vorgesetzter nur Daten, die technisch korrekte Objekte beschreiben, als Versionen bezeichnen möchte. Als Mindestanforderung müssen die datenmodellinhärenten Integritätsbedingungen (z.B. referentielle Integrität, Kardinalitätsrestriktionen usw.) erfüllt sein, da die Verarbeitung der Daten mit den Manipulationsoperationen des Datenmodells erfolgt. Weitergehende Anforderungen können durch ein Klassenkonzept, unterstützt durch ein Autorisierungskonzept, erfüllt werden. Klassen dienen dazu, alle Versionen mit "gemeinsamen Eigenschaften" zusammenfassen und damit einen anwendungsspezifischen Versionsbegriff zu bilden. Das Autorisierungskonzept wird benötigt, um die Zuordnung der Versionsbegriffe, respektive der Versionsklassen zu Benutzern beschreiben zu können. Aus Platzgründen kann hier nicht näher auf das Autorisierungskonzept eingegangen werden, das den in SQL [Date83] verwendeten Konzepten genügt. Die gemeinsamen Eigenschaften der Versionen einer Klasse können im einfachsten Fall durch die Angabe einer Konsistenzbedingung im Sinne der ASSERT-Klausel von SQL (bei der Definition der Klasse) beschrieben werden. Viele technische Anforderungen lassen sich allerdings damit nicht formulieren, so daß benutzerdefinierte Programme (Werkzeuge) herangezogen werden müssen. Diese Werkzeuge müssen nach dem Aufruf (idealerweise durch das DBS) und der Eingabe eine Objektversion entscheiden, ob die Version die gestellten Anforderungen erfüllt, d.h. der Klasse zugehörig ist.

Dieses Klassenkonzept in Verbindung mit einem Autorisierungskonzept erlaubt sehr leicht die Nachbildung von üblichen (nicht rechnergestützten) Freigabeverfahren, die meist auf der Vergabe eines Stempels durch den Abteilungsleiter beruhen: man definiert die Klasse "gestempelt" mit dem Werkzeug "Stempel", das die Zugehörigkeit zur Klasse kontrolliert und beschränkt das Ausführungsrecht von "Stempel" auf den Abteilungsleiter. Ebenso kann die Sichtbarkeit von Versionen dediziert für bestimmte Benutzer und Benutzergruppen auf Versionen in bestimmten Klassen reduziert werden, so daß für jeden Benutzer eine eigene Versionssemantik realisiert werden kann.

In Bild 3.5 ist die Definition einer Klasse in vereinfachter Syntax dargestellt. Als Beispiel wird eine Klasse "neueste Versuche" definiert, die die Blätter des Abstammungsgraphen umfaßt. Durch Autorisierungsmaßnahmen ist bspw. leicht zu erreichen, daß nur von diesen Versionen weitere Versionen abgeleitet werden dürfen.

```
DEFINE CLASS <class> FOR <object>
        ASSERT <Zustandsbedingung> /
        TOOL IS <tool_name>;
Beispiel:
DEFINE CLASS Neueste_Versuche FOR Struktur
        ASSERT (SUCCESSORS OF DERIVATION WHERE LEVELS ARE <1..1> = EMPTY);
DEFINE CLASS Erste_Versuche FOR Topographie
        ASSERT (PREDECESSORS OF DERIVATION WHERE LEVELS ARE <1..MAX> = EMPTY);
und für die Autorisierung:
GRANT DERIVE FOR Struktur ON Neueste_Versuche TO Entwerfer_Müller;
```

Bild 3.5: Definition von Klassen

Ein wichtiger Aspekt bei der Festlegung anwendungsspezifischer Versionsbegriffe liegt in dem Modus der Versionserzeugung und -verwendung. In Bild 3.3 wurde die Versionserzeugung an einen bestimmten Benutzer gebunden und liegt somit außerhalb des Systems, d.h., beim Einfügen der Version in eine bestimmte Klasse muß nachträglich überprüft werden, ob die Versionssemantik gewährleistet bleibt. Im Falle des Werkzeugs aus Kapitel 2 (Chip-Planner) müßte also überprüft werden, ob die erzeugte Topographie "konsistent" ist. Geht man davon aus, daß das Werkzeug, bedingt durch seine Arbeitsweise, nur korrekte Daten erzeugt, so ist dieser Test überflüssig. Zur Beschreibung solcher Sachverhalte müssen die Werkzeuge selbst im Schema beschrieben werden. Dies erlaubt eine exaktere Beschreibung bzgl. der

Versionserzeugung als auch eine exaktere Rechtevergabe (Ausführung bestimmter Werkzeuge) für Benutzer. Bild 3.6 zeigt die Definition eines Werkzeuges auf.

```
DEFINE   TOOL <tool>
         READS {VERSION OF <object> IN CLASS <class>}
         GENERATES {VERSION OF <object> IN CLASS <class>}

Angewendet auf das Beispiel (Kapitel 2, Bild 3.3):

DEFINE   TOOL W1
         READS VERSION OF Struktur IN CLASS Neueste_Versuche
         GENERATES VERSION OF Topographie IN CLASS Erster_Versuch

DEFINE   TOOL W2
         READS VERSION OF Topographie
         GENERATES VERSION OF Topographie
```

Bild 3.6: Definition von Werkzeugen

Analog Bild 2.3 liest das Werkzeug W_1 eine Version einer Struktur aus der Klasse der "Neuesten Versuche" und erzeugt eine Version einer Topographie. Wie schon erwähnt wurde, stellen die so erzeugten Topographieversionen alternative Lösungen dar. Durch die Zuordnung der Versionen zur Klasse der "Ersten Versuche" ist garantiert, daß die Versionen keine Vorgänger besitzen (vgl. Bild 3.3) und somit Alternativen darstellen.

Das vorgestellte Versionsmodell erlaubt es jedem Benutzer eine bestimmten Versionsbegriff mit der von ihm intendierten Semantik zu realisieren. Weiterhin läßt sich die Art und Weise wie neue Versionen entstehen sollen (durch welche Werkzeuge), wer Versionen (mit welchen Werkzeugen) erzeugen oder ändern darf und von welchen Versionen überhaupt Ableitungen gemacht werden dürfen sehr flexibel beschreiben.

3.3 Beziehungen zwischen versionierten Objekten

Analog zu der Abstraktion bei der Bildung des Objektbegriffs, muß der Beziehungsbegriff neu gefaßt werden. Ein Objekt abstrahiert von der Menge seiner Versionen und trägt damit wesentlich zur Strukturierung und Vereinfachung des Informationsgehalts eines DB-Schemas bei. Analog muß bei Beziehungen zwischen versionierten Objekten verfahren werden: Beziehungen werden zwischen Objekten definiert und können dann ggf. auf Objektversionen verfeinert werden. Im vorgestellten VLSI-Schema wird z.B. aus dem Objekt "Struktur" das Objekt "Topographie" erzeugt. Erst in einem weiteren Schritt ist festzustellen, aus welcher Version der Struktur eine oder eventuell mehrere Versionen des Objekts "Topographie" abgeleitet wurden (vgl. Bild 3.7).

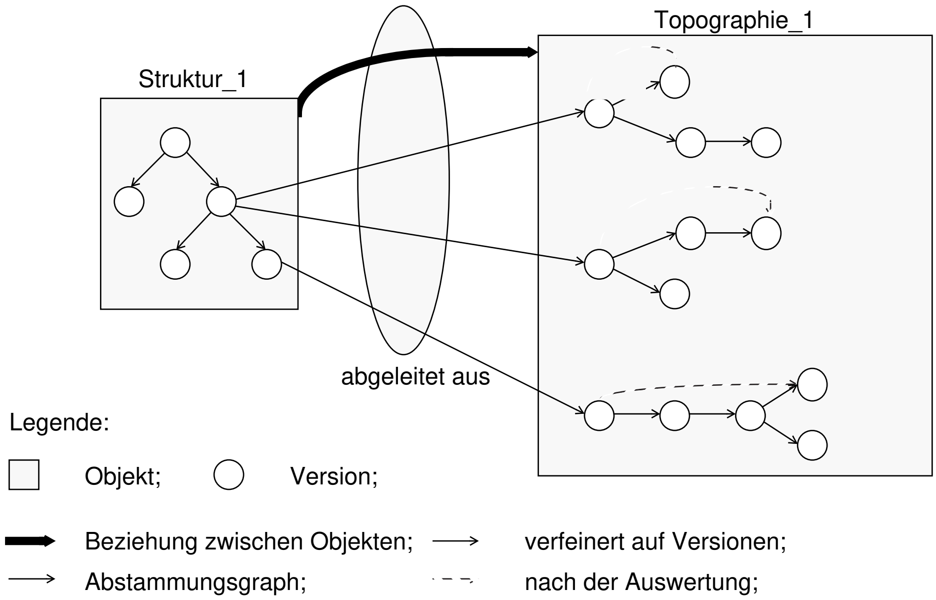

Bild 3.7: Beziehungen zwischen versionierten Objekten

Bei der Definition einer Objektbeziehung (Link) sind also zwei Abstraktionsebenen zu unterscheiden:

- Objektebene

  Die Beziehung wird zwischen den Objekten definiert und nimmt dabei auf die Versionierung keine Rücksicht; die Beziehungen können analog dem E/R-Modell den Typen 1:1, 1:n und n:m zugeordnet werden bzw. durch Kardinalitätsrestriktionen eingeschränkt werden.

- Versionsebene

  Beziehungen zwischen Versionen können (als Verfeinerung von Objektbeziehungen) nur geknüpft werden, wenn zwischen den zugehörigen Objekten eine Beziehung besteht. Analog, aber unabhängig zu den definierten Objektbeziehungen, können die Versionsbeziehungen den Typen 1:1, 1:n oder n:m zugeordnet werden.

Auf der Versionsebene ergeben sich durch die Versionierung, respektive durch die Weiterentwicklung des Objekts, weitere Probleme. Wie in Bild 3.7 dargestellt ist, kann die Topographie durch den Einsatz geeigneter Methoden optimiert werden. Dies bedeutet, daß die Versionsbeziehung zunächst zur "Wurzelversion" eines Abstammungsgraphen führt, indem dann eine geeignete Version bestimmt werden muß. Dieser häufig erwünschte Vorgang kann automatisiert werden, wenn das Auswahlverfahren (zur Bestimmung einer bestimmten Version innerhalb des Objekts) bei der Definition der Versionsbeziehung angegeben wird. Als Auswahlkriterium können Eigenschaften der Versionen, Klassenzugehörigkeit oder die Position im Abstammungsgraphen genutzt werden. Bild 3.8 zeigt in vereinfachter Syntax die Definition von Beziehungen sowie die zugehörigen Operationen auf.

```
DEFINE LINK <link> BETWEEN <object1>, <object2>
                    CARDINALITY          FROM <object1> TO <object2> IS <intervall> FOR VERSIONS IS <intervall>,
                                         FROM <object2> TO <object1> IS <intervall> FOR VERSIONS IS <intervall>
        SELECTION FOR <object1> IS       <version_selection>/
                                         <graph_selection>/
                                         <class_selection>
        SELECTION FOR <object2> IS ...   (* s.o.*)
CREATE LINK <link> BETWEEN
        OBJECTS <object_selection_1>,         <object_selection_2> /
        VERSIONS <version_selection_1>,       <version_selection_2>;
DELETE LINK ... (* s.o.*)
        mit der Objektbeziehung werden alle abhängigen Versionsbeziehungen gelöscht.
```

Bild 3.8: Operationen auf Objektbeziehungen

Bezogen auf das Beispiel aus Kapitel 2 muß eine Beziehung zwischen Strukturobjekten und Topographieobjekten definiert werden. Da zu jeder Struktur genau eine Topographie gehört, ist diese Beziehung den Typ 1:1 zuzuordnen. Auf der Versionsebene hingegen muß die Beziehung vom Typ 1:n sein, da das Werkzeug W_1 (vgl. Bild 2.3) aus einer Strukturversion mehrere Topographieversionen (Alternativen) erzeugt. Bei der Definition der Versionsbeziehung soll weiterhin angenommen werden, daß die jeweils neueste Version der Alternative ausgewählt wird..

```
DEFINE LINK abgeleitet aus BETWEEN Struktur, Topographie
          CARDINALITY FROM Struktur TO Topographie IS <0 . . 1> FOR VERSIONS IS <0..*>,
          CARDINALITY FROM Topographie TO Struktur IS <1 . . 1> FOR VERSIONS IS <1..1>
SELECTION FOR Struktur IS V_id
SELECTION FOR Topographie IS
          SUCCESSORS      OF Topographie V_id FROM DERIVATION
                          WHERE V_id = TOPICAL
```

Bild 3.9: Definition einer versionierten Beziehung

Ergänzend zu Bild 3.4 müssen die dort erzeugten Objekte und Versionen miteinander in Beziehung gesetzt werden. Dabei muß immer die Verbindung zwischen den Objekten vor den Verbindungen zwischen den Objektversionen gesetzt werden. Dies führt zu der in Bild 3.10 dargestellten Operationsfolge.

```
CREATE      LINK abgeleitet_aus BETWEEN
            OBJECTS      Struktur_1, Topographie_1;
CREATE      LINK abgeleitet_aus BETWEEN
            OBJECTS      Struktur_1,  Topographie_1
            VERSIONS     V_7,         V_1
CREATE      LINK abgeleitet_aus BETWEEN
            OBJECTS      Struktur_1,  Topographie_1
            VERSIONS     V_7,         V_3
```

Bild 3.10: Operationsfolge zur Erzeugung versionierter Beziehungen

4. Abbildung auf DBS

Das vorgestellte Versionsmodell erlaubt sowohl die Modellierung und Handhabung versionierter Objekte als auch die Festlegung eines anwendungsspezifischen Versionsbegriffs. Es basiert in seinen Grundannahmen auf den in herkömmlichen DBS verwendeten Konzepten und kann somit als Zusatzebene auf diese DBS aufgesetzt werden. Damit ist eine Evaluierung des Modells mit den heute auf dem Markt befindlichen Systeme möglich. In Abhängigkeit von der Mächtigkeit des verwendeten DBS variiert allerdings sowohl der Implementierungsaufwand als auch die Leistungsfähigkeit des Gesamtsystems.

Eine wesentlich höhere Effizienz (sowohl bzgl. des Implementierungsaufwandes als auch der Leistungsfähigkeit) kann bei dem Einsatz von NDBS erwartet werden. Diese Systeme befinden sich jedoch noch im Forschungs- oder Entwicklungsstadium, so daß sie für kommerzielle Anwendungen noch nicht zur Verfügung stehen. Die meisten dieser NDBS bieten einerseits meist überhaupt keine oder nur rudimentäre Versionsunterstützung, erlauben aber andererseits mindestens eine strukturelle oder sogar eine verhaltensmäßige Objektorientierung. Dies hat zur Folge, daß das Versionsmodell weiterhin als "Zusatzebene" (Erweiterung des Kern-Systems; Anwendungsorientierte Schnittstelle, realisiert durch "Baukastenteile", etc.) gesehen werden kann.

Abbildung 4.1 zeigt den prinzipiellen Aufbau der Zusatzebene in Form einer Schichtenarchitektur. Im weiteren Verlauf von Kapitel 4 werden die wesentlichen Abbildungsschritte der versionierten Objekte auf nicht-versionierte, elementare "Tupel" aufgezeigt. Es ist dabei zu beachten, daß viele dieser Abbildungsschritte bei Verwendung eines NDBS ganz entfallen können (da sie implizit im NDBS erfolgen) oder durch die Modellierungsmittel des NDBS (einfach) zu formulieren sind.

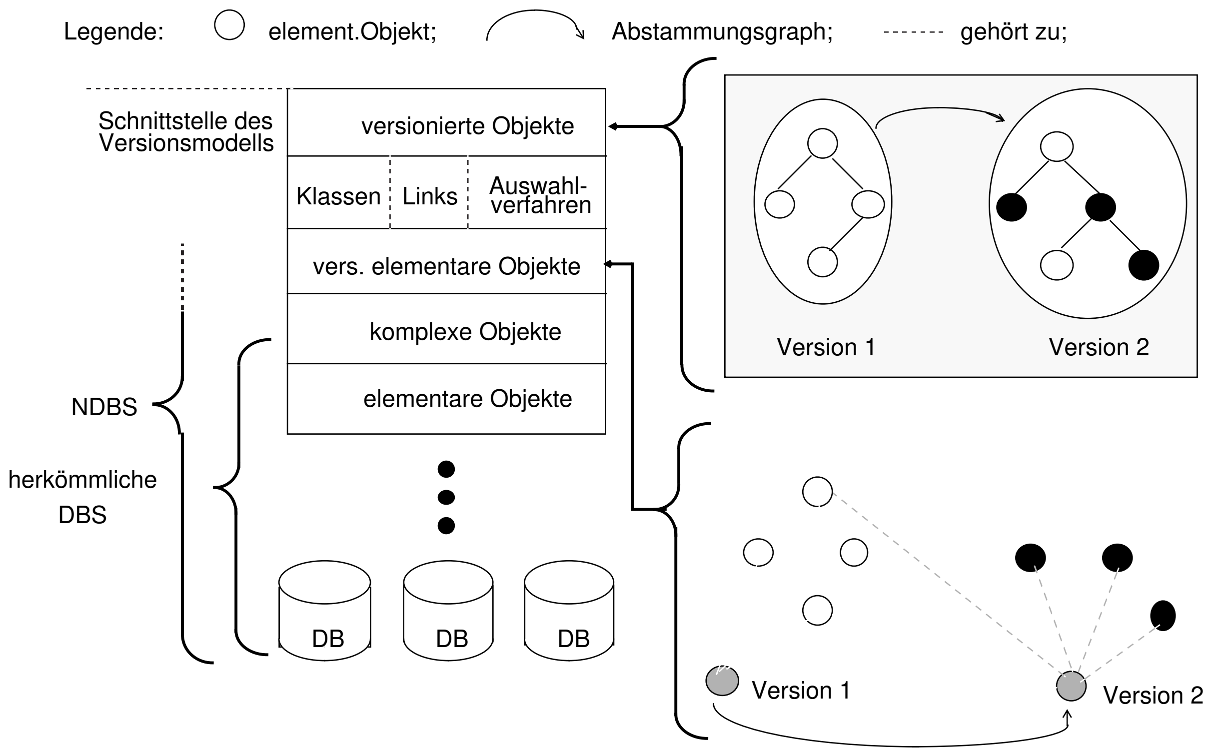

Bild 4.1: Schichtenarchitektur der Zusatzebene

Abbildung der versionierten Objekte

Wie in den vorangegangenen Kapiteln beschrieben wurde, sind AW-Objekte (z.B. Struktur, Topographie) als Ausschnitt des DB-Schemas beschrieben und bestehen damit aus einer Menge von elementaren Objekten (z.B. Modul, Pin, Netz usw.) und Verbindungen zwischen ihnen. Um eine möglichst einfache Abbildung dieser versionierten AW-Objekte auf die Datenelemente des verwendeten DBS zu erreichen, wird jedes (zusammengesetzte, versionierte) AW-Objekt durch genau ein elementares (versioniertes) Objekt repräsentiert, das sowohl die objektbeschreibenden Attribute als auch die Verbindungen zu allen elementaren Objekten des AW-Objekts umfaßt (vgl. Bild 4.1). Diese Objekt wird im folgenden als "Repräsentant" bezeichnet. Durch diesen Schritt wird die Versionsbildung von (zusammengesetzten) AW-Objekten auf die Versionsbildung von elementaren Objekten reduziert: jede Version des zusammengesetzten Objekts wird durch eine Version des Repräsentanten dargestellt. Der Abstammungsgraph wird zwischen den Repräsentanten aufgebaut.

Abbildung der Links und Klassen

Die Verbindung zwischen versionierten zusammengesetzten Objekten (Links) kann auf Verbindungen zwischen den Repräsentanten und damit auf Verbindungen zwischen elementaren Objekten reduziert werden. Die Zugriffsinformationen oder -prozeduren werden bei der Definition des Repäsentanten berücksichtigt.

Klassen werden ebenfalls durch elementare Objekte repräsentiert, die wiederum die Informationen enthalten müssen, die notwendig sind, um die Klassenzugehörigkeit von Objektversionen entscheiden zu können. Die Klassenzugehörigkeit wird durch Verbindungen zwischen dem "Klassenobjekt" und den Repräsentanten ausgedrückt. Bild 4.2 zeigt den Informationsgehalt der Repräsentanten mit einem E/R-Diagramm graphisch auf.

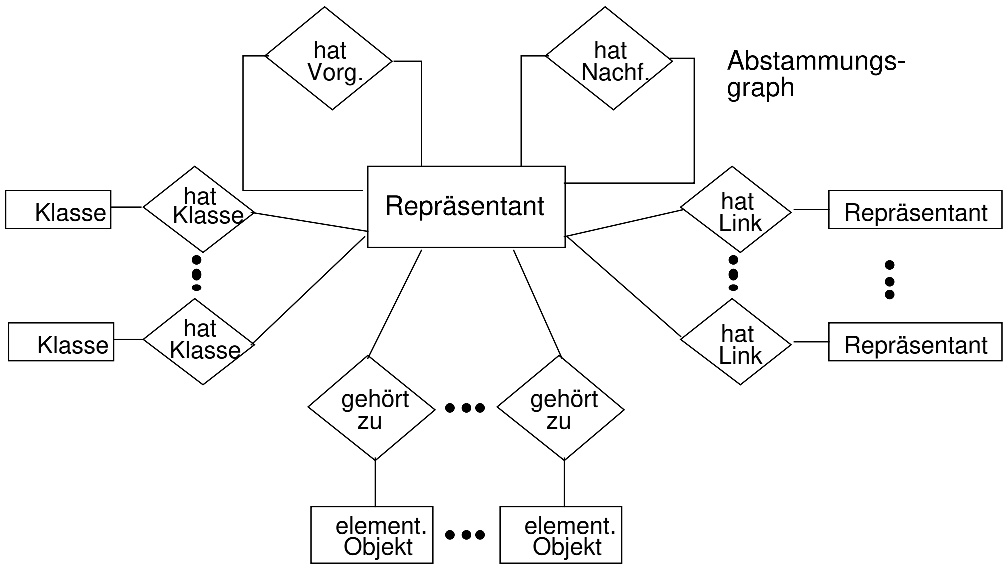

Abb. 4.2: E/R-Diagramm zur Objektrepräsentation

Versionierung elementarer Objekte

Die Versionierung der elementaren Objekte und der Verbindungen zwischen ihnen bildet die Grundlage des Versionsmodells. Ausschlaggebend für die Güte dieser Abbildung ist das zu erwartende Zugriffs- und Änderungsverhalten. Im allgemeinen kann angenommen werden, daß der Zugriff auf die aktuelle Version dominierend ist. Dies läßt sich auch in der in Kapitel 2 vorgestellten Anwendung erkennen. Weiterhin wird auf der aktuellen Version die höchste Änderungswahrscheinlichkeit erwartet. Dies führt insgesamt zu der Forderung, die vergleichsweise geringen aktuellen Daten von den historischen Daten zu trennen. Die zu erwartende geringe Zugriffshäufigkeit in Verbindung mit dem hohen Datenvolumen lassen den Einsatz von Differenztechniken ratsam erscheinen. Aus Platzgründen kann auf diese Abbildung nicht näher eingegangen werden, in [Ahn88, Dada84, Käfe88, Käfe89, Lum84] wird die Problematik jedoch ausführlich behandelt.

5. Zusammenfassung

Ausgehend von für den Bereich VLSI-Chip-Entwurf beispielhaften Informationsstrukturen wurde die Notwendigkeit einer Versions- und Alternativenbildung aufgezeigt und ein anwendungsspezifisches, intuitives Versionsverständnis skizziert. Als wesentliche Eigenschaften der zu versionierenden VLSI-Objekte ist ihre i. allg. komplexe Struktur sowie ihre teilweise Überlappung zu nennen.

Als Ansatz zur Lösung der aufgezeigten Problemstellung wurde ein Basisversionsmodell vorgestellt, das die Festlegung eines anwendungsspezifischen Versionsbegriffes erlaubt. Hierzu sind Modellkonstrukte zur Definition der zu versionierenden Objekte und Operationen zur Handhabung der entstehenden Versionsgraphen vorgesehen. Die eingeführten Abstraktionsebenen führen zur Unterscheidung zwischen Objektversionen und Mengen von zusammenhängenden Objektversionen, die dann als abstrakte Objekte aufgefaßt werden können. Entsprechend erlaubt das Basisversionsmodell die explizite Unterscheidung zwischen Objektversionsbeziehungen und Beziehungen auf Objektebene, wodurch insgesamt eine genauere Modellierung ermöglicht wird.

An zahlreichen Beispielen wurde die prinzipiell Tauglichkeit der eingeführten Konzepte auf der Modellierungsebene motiviert. Daneben konnten erste Realisierungsvorschläge diskutiert werden, die eine Abbildung bzw. Einbettung des Basisversionsmodells auf bzw. in eine DBS-Architektur vorsehen. Die vorgestellten Konzepte befinden sich derzeit in der Implementierungsphase, wobei als DBS-Grundlage der NDBS-Kern PRIMA [Härd88] zum Einsatz kommt. Als eine konkrete Validierung wird eine Verwendung in einer realen VLSI-Entwurfsumgebung [Pahl88, Siep89] angestrebt.

6. Literatur

Abra87 Abramowicz, K., Dittrich, K.R., Gotthard, W., Längle, R., Lockemann, P.C., Raupp, T., Rehm, S., Wenner, T: DAMOKLES: Entwurf und Implementierung eines Datenbanksystems für den Einsatz in Software-Produktionsumgebungen, GI-Fachgruppe "Software-Engineering", Mitteilungen der GI-Fachgruppe "Software-Engineering", Softwaretechnik-Trends, Heft 7-2, Okt. 1987, S. 2-21.

Ahn88 Ahn, I., Snodgrass, R.: Partitioned Storage for Temporal Databases, Information Systems, Vol. 13, No. 4, 1988, pp. 369-391.

Bato85 Batory, D.S., Kim, W: Modeling Concepts for VLSI CAD Objects, ACM TODS, Vol. 10, No. 3, sept. 1985, pp. 322-346.

Bato87 Batory, D.S.: Principles of Database Management System Extensibility, in: IEEE Database Engineering, Vol. 10, No. 2, Special Issue on Extensible Database Systems, Juni 1987, pp. 40-46.

Bato88 Batory, D.S.; Barnett, J.R.; Garza, J.F.; Smith, K.P.; Tsukuda, K.; et al.: GENESIS: An Extensible Database Management System, in: IEEE Transactions on Software Engineering, Vo. 14, No. 11, November 1988, pp. 1711-1731.

Blan87 Blanken, H., Ijbema, A: Storage of Versioned Objects in a CIM Environment, Proc. of the Int. Conf. on Data and Knowledge Systems for Manufacturing and Engineering, Hartford, 1987, pp. 65-74.

CD86 Carey, M.J.; DeWitt, D.J.: Extensible Database Systems, in: On Knowledge Base Management Systems, (Hrsg.: Brodie, M.L.; Mylopoulos, J.), Springer, 1986, pp. 315-330.

CD87 Carey, M.J.; DeWitt, D.J.: An Ovierview of the EXODUS Project, in: IEEE Database Engineering, Vol. 10, No. 2, Special Issue on "Extensible Database Systems", 1987, pp. 47-54.

Chou86 Chou, H.-T., Kim, W: A Unifying Framework for Versions in a CAD Environment, Proc. of the 12th Int. Conf. on VLDB, Kyoto, Japan, 1986, 1986, pp. 336-344.

Dada84 Dadam, P., Lum, V., Werner,H.-D.: Integration of Time Versions into a Relational Database System, Proc. 10th Int. Conf. on VLDB, Singapore, 1984, pp.509-522.

Ditt84 Dittrich, K.R., Kotz, A.M., Mülle, J.A., Lockemann, P.C: Datenbankkonzepte für Ingenieuranwendungen: Eine Übersicht über den Stand der Entwicklung, Proc. 14. GI-Jahrestagung, Braunschweig, 1984, Springer-Verlag, Informatik Fachberichte Nr. 88, 1984, S. 175-192.

Ditt85 Dittrich, K.R., Kotz, A.M., Mülle, J.A., Lockemann, P.C: Datenbankunterstützung für den ingenieurwissenschaftlichen Entwurf, Informatik-Spektrum, Bd. 8, H. 3, 1985, S. 113-125.

Ditt88 Dittrich, K.R., Lorie, R.A: Version Support for Engineering Database Systems, IEEE Transactions on Software Engineering, Vol. 14, No. 4, 1988, pp. 429-437.

Härd87 Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A: PRIMA - A DBMS Prototype Supporting Engineering Applications, Universität Kaiserslautern, Fachbereich Informatik, Forschungsbericht Nr. 22/87 des SFB124, 1987, 20 pp.

Härd88 Härder, T: Overview of the PRIMA Project, (Hrsg: Härder, T.), The PRIMA Project - Design and Implementation of a Non-Standard Database System, University Kaiserslautern, SFB 124, Report No. 26/88, 1988, pp. 1-12.

Härd89 Härder, T: Non-Standard DBMS for Support of Engineering Applications - Requirement Analysis and Architectural Concepts, (Hrsg: Shriver, B.D.), Proc. of the 22nd Hawaii International Conference on System Sciences (HICSS-22), Kailua-Kona, Hawaii, Volume II, 1989, pp. 549-548.

Käfe88 Käfer, W.:Ein Geschichts- und Versionsmodell für komplexe Objekte, Proc. der GI-Fachtagung "Datenbanksysteme für Büro, Technik und Wissenschaft", Zürich, 1988, Springer Verlag, Informatik Fachberichte Nr. 204, Berlin, Heidelberg, 1988.

Käfe89 Käfer, W., Ritter, N.: Modellierung versionierter Objekte auf PRIMA, in Vorbereitung.

Katz84 Katz, R.H., Lehman, T: Database Support for Versions and Alternatives of Large Design Files, IEEE Transactions on Software Engineering, Vol.SE-10, No.2, 1984, pp 191-200.

Katz86 Katz, R.H., Anwarrudin, M., Chang, E: A Version Server for Computer-Aided Design Data, Proc. of the ACM/IEEE 23rd Int. Conf. on Design Automation, Las Vegas, 1986.

Katz86 Katz, R.H., Chang, E., Bhateja, R: Version Modelling Concepts for Computer-Aided Design Databases, (Hrsg: Zaniolo, C.), Proc. SIGMOD Int. Conf. on Management of Data (Washington, D.C., May 28-30, 1986), ACM SIGMOD Record, vol. 15, no. 2, june 1986, pp. 379-386.

Klah86 Klahold, P., Schlageter, G., Wilkes, W: A General Model for Version Management in Databases, Proc. 12th International Conference on VLDB, Kyoto, Japan, 1986, pp. 319-327.

KSZ87 Klein, A., Schreiner, F., Zimmermann, G.: Ein Sizing-Modell für den VLSI-Entwurf, Forschungsbericht Nr. 25/87, SFB 124, Universität Kaiserslautern, 1987.

Lum84 Lum, V.,Dadam, P.,Erbe, R., Guenauer, J., Pistor, P., Walch, G., Werner, H.-D., Woodfill, J.: Designing DBMS Support for the Temporal Dimension, Proc. of the SIGMOD Int. Conf. on Management of Data, 1984, pp. 115-130

Mits88 Mitschang, B: Ein Molekül-Atom-Datenmodell für Non-Standard-Anwendungen - Anwendungsanalyse, Datenmodellentwurf und Implementierungskonzepte, Springer-Verlag, Informatik Fachberichte, 185, 1988.

McPh87 McPherson, J.A.; Pirahesh, H.: An Overview of Extensibility in Starburst, in: Database Engineering, Vol. 10, No. 2, Special Issue on Extensible Database Systems, june 1987, pp. 32-39.

Pahl88

Sche87 Schek, J.-J.: DASDBS: A Kernel DBMS and Application-Specific Layers, in: IEEE Database Engineering, Vol. 10, No. 2, 1987, pp. 62-64.

Schö89 Schöning, H.; Sikeler, A.: Extending and Configuring an Enhanced Database Management System, 1989. eingereicht zur Veröffentlichung

Sch88 Schürmann, B. :Hierarchisches Top Down Chip Planning, in Informatik Spektrum, Bd. 11, H. 2, April 1988, S.57-70.

Siep89

Zim86 Zimmermann, G.: Top-down design of digital systems, in: Logic Design and Simulation, E. Hörbst (Editor), Elsevier Science Publ. B.V. 1986.