in: Tagungsband der GI-Fachtagung 'Datenbanksysteme für Büro, Technik und Wissenschaft',Informatik aktuell,
März 1993, Braunschweig, S. 144-163.

Flexible Entwurfsdatenverwaltung für CAD-Frameworks: Konzept, Realisierung und Bewertung

W. Käfer*, B. Mitschang
Fachbereich Informatik, Universität Kaiserslautern
Postfach 3049, 6750 Kaiserslautern
e-mail: {kaefer | mitsch}@informatik.uni-kl.de

Kurzfassung:

Eine der Hauptzielrichtungen von CAD-Frameworks ist die Integration von einzelnen, eigenständigen CAD-Werkzeugen mittels einer zentralen Datenverwaltung. Wesentliche Aufgaben hierbei sind die Verwaltung aller entwurfsrelevanten Daten sowie die effiziente Bereitstellung werkzeugrelevanter Daten für den werkzeugspezifischen Entwurfsschritt. Hierzu sind die vom jeweiligen Werkzeug zu bearbeitenden Entwurfsdaten bzw. Entwurfsobjekte zu selektieren und in der jeweils benötigten Form bereitzustellen. Nach Beendigung des Entwurfsschrittes sind die geänderten Daten in den aktuellen Datenbestand zu integrieren. Um diese schwierige Aufgabe meistern zu können, ist es nötig, die vorherrschenden Objekt- und Entwurfsstrukturen entsprechend zu berücksichtigen.

Das hier vorgestellte Objekt-Versions-Modell OVM soll diesen hohen Anforderungen entsprechen. Mit OVM werden (versionierte) Objekte aus Elementarobjekten zusammengesetzt; zwischen diesen Objekten können Objekt-, Versions-, und Konfigurationsbeziehungen in flexibler Art und Weise aufgebaut werden. Die zugehörige Manipulationssprache OML erlaubt ein adäquates Arbeiten mit den so strukturierten Objekten. OVM bietet eine hohe Abstraktion von der zugrundeliegenden Datenrepräsentation, so daß eine OVM-Realisierung im Prinzip mit unterschiedlichen DBS durchgeführt werden kann. Unsere OVM-Realisierung mittels des PRIMA-Systems wird vorgestellt, die gewonnenen Erfahrungen berichtet und eine vergleichende Bewertung zu Realisierungsalternativen gegeben.

1. Motivation

Ein wesentlicher Schritt zur Beherrschung der Komplexität des Entwurfs (etwa VLSI-Entwurf oder CAD im Maschinenbau, in der Architektur, im Bau- und Raumwesen), stellt der Einsatz Integrierter Rechnergestützter Entwurfsumgebungen (engl. "integrated computer-aided design environment" oder einfach "CAD Frameworks") dar [RS92, HNST90]. Diese Umgebungen bieten eine Sammlung rechnergestützter Dienste in Form von einzelnen Werkzeugen an, die jeweils miteinander gekoppelt und fein aufeinander abgestimmt sind und alle (wesentlichen) Phasen des Entwurfsprozesses in kontinuierlicher Weise unterstützen. Hierbei stellt die integrierte Datenhaltung eine wesentliche Voraussetzung zur gewünschten Werkzeugintegration dar.

Eine wichtige Aufgabe von CAD-Frameworks ist somit die zentrale Verwaltung aller ***entwurfsrelevanten Daten*** sowie die Bereitstellung der ***werkzeugrelevanten Daten*** für den werkzeugspezifischen Entwurf. Die Anwendung eines CAD-Werkzeuges bedeutet das Ausführen eines sog. *Entwurfsschrittes*, der in mehrere Phasen unterteilt werden kann. Das hier zugrundeliegende *kontext-basierte Verarbeitungskonzept* unterscheidet im wesentlichen drei Phasen. Mittels einer Checkout-Operation werden die werkzeugrelevanten Daten bei der Datenhaltungskomponente angefordert. Nach deren Bereitstellung in einem lokalen Verarbeitungsbereich wird der eigentliche Entwurfsschritt durchgeführt. Dabei wird häufig auf die Daten im Verarbeitungsbereich zugegriffen und diese auch i. allg. geändert bzw. neue Entwurfsdaten erzeugt. Vor Beendi-

*) Zur Zeit als Gastwissenschaftler bei IBM, Almaden Research Center, San Jose, Kalifornien.

gung der Werkzeugausführung werden diese Änderungen mittels einer Checkin-Operation wieder der zentralen Datenhaltung zurückgegeben.

In vielen Fällen dienen Versionierungs- und Konfigurationsstrukturen zur Organisation der Entwurfsdaten [DL88, Ka90, Kä91] entsprechend den Anforderungen des werkzeugspezifischen Entwurfs. Beispielsweise können diese Strukturen den Entwurfsprozeß bzw. dessen Entwurfsschritte reflektieren und erlauben somit insbesondere eine Zuordnung von Werkzeugläufen und Entwurfsobjekten bzw. Entwurfsobjektversionen. Weiterhin dienen sie der Übergabe von aktuellen Entwurfsdaten von einem Werkzeug zum nächsten und erlauben ein kontrolliertes Weiterschreiben von Entwurfsobjektversionen.

Die werkzeugrelevanten Daten stellen den für ein konkretes Werkzeug und einen konkreten Entwurfsschritt relevanten Ausschnitt der Entwurfsdaten dar. Dieser sog. ***Verarbeitungskontext*** setzt sich i. allg. aus verschiedenen Elementarteilen zusammen, die über (sog. strukturelle) Beziehungen miteinander verbunden sind. Dabei sind die Beziehungen dem Werkzeug bekannt und werden auch für die werkzeugbezogene Verarbeitung benutzt, was im wesentlichen ein Entlanglaufen dieser Beziehungen bedeutet. Aus Leistungsgründen ist es daher erforderlich die Navigation entlang der strukturellen Beziehungen durch entsprechend geeignete Repräsentation des Verarbeitungskontextes im Verarbeitungsbereich des Werkzeugs zu unterstützen.

Aus dem Blickpunkt der Datenhaltung entspricht ein Verarbeitungskontext einer *Sicht* (engl. view) auf den Datenbestand, die durch eine (Anfrage-)Sprache spezifiziert ist. Evaluieren der Sicht (bei der Checkout-Operation) bedeutet entsprechend das Extrahieren des spezifizierten Kontextes und dessen Bereitstellung im werkzeuglokalen Verarbeitungsbereich. Um die Komplexität der Datenhaltungsaufgaben eines Werkzeuges weiter zu verringern, wird dem Werkzeug während seiner Verarbeitung eine versionsfreie Sicht auf die Entwurfsdaten geboten. Die Checkout- bzw. die Checkin-Operationen müssen jedoch vorhandene Versionierungs- und Konfigurationsstrukturen berücksichtigen. Schon allein dieser komplexe Extraktionsprozeß legt die Verwendung von Sichten nahe. Durch die Trennung von Spezifikation des Verarbeitungskontextes, eigentlicher Extraktion aus dem Entwurfsdatenbestand und anschließender Bereitstellung erhält die Datenhaltungskomponente die freie Wahl von Datenhaltungsmethoden und Datenrepräsentation sowie von Datenbereitstellungskonzepten im Verarbeitungsbereich. Weiterhin ist zu bemerken, daß verschiedene Werkzeuge i. allg. auf unterschiedlichen Verarbeitungskontexten basieren, die sich häufig auf den zugrundeliegenden Entwurfsdaten überlappen.

All diese Überlegungen ergeben, daß ein Modell und eine zugehörige Sprache erforderlich sind, welche zusammen eine flexible und abstrakte Spezifikation von Verarbeitungskontexten erlauben. Das hier vorgestellte Objekt-Versions-Modell OVM entspricht diesen Anforderungen. Mit OVM werden (versionierte) Objekte aus Elementarobjekten zusammengesetzt. Zwischen diesen Objekten können Objekt-, Versions-, und Konfigurationsbeziehungen in flexibler Art und Weise aufgebaut werden. Die zugehörige Manipulationssprache OML erlaubt ein adäquates Arbeiten mit den so strukturierten Objekten. Die Spezifikation von Verarbeitungskontexten entspricht in OVM der Definition von Sichten, die OVM-Objekte ansprechen bzw. bearbeiten. OVM bietet dabei eine hohe Beschreibungsebene, die insbesondere von der zugrundeliegenden Repräsentation der Elementarobjekte abstrahiert. Dies wiederum bedeutet, daß eine OVM-Realisierung im Prinzip mit unterschiedlichen Datenhaltungssystemen bzw. Datenbanksystemen (DBS) durchgeführt werden kann. Für alle nachfolgenden Diskussionen wird angenommen, daß die Entwurfsdaten durch ein DBS verwaltet werden, dessen Eigenschaften dann ggf. konkretisiert werden.

Bevor allerdings OVM erklärt und an einem Beispiel aus dem VLSI-Entwurf vorgestellt wird (Kapitel 3), sollen in Kapitel 2 grundlegende Eigenschaften und Anforderungen an Objektmodelle für den Entwurf diskutiert werden. In Kapitel 4 wird eine mögliche OVM-Realisierung mittels unseres Non-Standard-DBS PRIMA [HMMS87, GG92] vorgestellt und die dabei gewonnenen Erfahrungen berichtet. Eine vergleichende

Bewertung von alternativen Realisieungsmöglichkeiten und ein kurzes Resümee (in Kapitel 5) schließen die Diskussion ab.

2. Objektmodelle für den Entwurf

Im folgenden werden die Charakteristika der werkzeugbasierten Verarbeitung diskutiert. Dabei ergeben sich zwei unterschiedliche Abstraktionsebenen. Die Berücksichtigung dieser Aspekte in einem Objektmodell wird dann anschließend betrachtet.

2.1 Charakteristika eines werkzeugbasierten Entwurfsschrittes

In Entwurfsanwendungen müssen zwei Arten der Verarbeitung unterschieden werden [GHM92, Kä91]. Auf der Ebene der Entwurfssteuerung, also des Einsatzes bestimmter Entwurfsmethoden zur *Ablaufsteuerung des Entwurfsprozesses*, sind Entwurfsobjekte bzw. mehr noch Entwurfsobjektversionen Gegenstand des Interesses. Hier ist eine abstrakte Sicht auf die Entwurfsdaten vorteilhaft, da lediglich die Abhängigkeiten zwischen Entwurfsobjektversionen bzw. die Zusammenhänge zwischen Entwurfsobjektversionen und einzelnen Werkzeugläufen, also Entwurfsschritten, berücksichtigt werden müssen.

Die zweite Art der Verarbeitung erfolgt durch die *spezialisierten Entwurfswerkzeuge*, die die Entwurfsdaten im Sinne des Entwurfsziels transformieren. Die Art der Datenverarbeitung basiert auf dem 'inneren' Aufbau der Entwurfsobjektversionen. Den Werkzeugen wird daher eine versionsfreie Sicht auf die zu bearbeitenden Objekte geboten. Die Auswahl der zu lesenden Versionen bzw. die Integration der erzeugten Versionen erfolgt somit strikt getrennt von der eigentlichen Verarbeitung der Entwurfsdaten. Diese Trennung in *versionsfreie Werkzeugsicht* und *versionsbehaftete Entwurfssteuerungsebene* ist ein wesentlicher Beitrag zur Beherrschung des Entwurfsprozesses und zur Reduktion der Komplexität der werkzeugbasierten Verarbeitung. Diese beiden Abstraktionsebenen werden auch von CAD-Frameworks forciert und drücken sich in den folgenden sechs Phasen der werkzeugbasierten Verarbeitung aus:

1. Die *Daten- bzw. Versionsauswahl* für den aktuellen Verarbeitungsschritt wird aufgrund entwurfsspezifischer bzw. entwurfsmethodenspezifischer Überlegungen zumeist von einem Entwerfer bzw. durch die Entwurfssteuerung getroffen. Hierzu müssen die betreffenden Ausschnitte (Entwurfsobjektversionen) aus den Entwurfsdaten einfach und kompakt bezeichnet bzw. beschrieben werden können.

2. Der Werkzeuglauf beginnt mit der *Extraktion der Eingabedaten* aus der Datenbank durch eine sog. *Checkout*-Operation. Dies muß einfach und effizient anhand einer deskriptiven Beschreibung erfolgen können. Der ausgewählte Kontext liefert eine versionsfreie Sicht auf die werkzeugrelevanten Daten.

3. Nach der Bereitstellung der werkzeugrelevanten Daten in einem werkzeuglokalen Verarbeitungsbereich wird der eigentliche Entwurfsschritt durchgeführt. Die *Verarbeitung* der Daten im Werkzeug erfolgt i. allg. traversierend und entlang der strukturellen Beziehungen, die die Elementarobjekte miteinander verbinden. Dabei werden Daten im Verarbeitungsbereich geändert bzw. neue Entwurfsdaten erzeugt. Effizienter Datenzugriff und Navigationsunterstützung (etwa durch entsprechende Hauptspeicher-Datenstrukturen) bestimmen die Leistungsfähigkeit der Verarbeitung.

4. Vor Beendigung der Werkzeugausführung werden die durchgeführten Änderungen mittels einer *Checkin*-Operation wieder dem DBS zurückgegeben.

5. Die *Integration der erzeugten Entwurfsversionen* erfolgt nach den für den aktuellen Verarbeitungsschritt geltenden Kriterien. Diese Kriterien sind von der verwendeten Entwurfsmethodik festgelegt und regeln i. allg. Abhängigkeiten zwischen den Entwurfsdaten, wie etwa die Ableitung von neuen Entwurfsversionen aus vorherigen Entwurfsversionen.
6. In einem letzten, *bewertenden Schritt* müssen die favorisierten Lösungen ausgewählt und zusammen mit den Ergebnissen anderer Entwurfsschritte im Hinblick auf das gesteckte Entwurfsziel beurteilt werden. Dies wird häufig unter dem Begriff *Konfigurierung* subsumiert.

Durch diese Phaseneinteilung werden die Trennung von *werkzeugspezifischen und entwurfsspezifischen Aspekten* und damit auch gleichzeitig die zwei unterschiedlichen Abstraktionsebenen verdeutlicht. Die Schritte 2, 3, und 4 bilden das kontext-basierte Verarbeitungskonzept und basieren auf einer versionsfreien Datensicht, wie sie von Entwurfswerkzeugen benötigt wird. Hingegen sind alle entwurfsspezifischen und damit zumeist auch die Versionierung betreffende Aspekte in den Schritten 1, 5 und 6 zusammengefaßt. Im nächsten Abschnitt wollen wir die Organisation der Entwurfsdaten eingehender untersuchen und zu einer klareren Begriffsbildung kommen. Die werkzeugbasierte Verarbeitung wird später (in Kapitel 4) nochmals aufgegriffen und in einem Architekturansatz für CAD-Frameworks aufgezeigt.

2.2 Organisation der Entwurfsdaten

2.2.1 Entwurfsobjekte und Entwurfsobjektversionen

Wir führen den Begriff des Entwurfsobjekts bzw. kurz des ***Objekts*** *als eindeutig benennbare und leicht zu handhabende, strukturierte Ansammlung von elementaren Daten* ein (vgl. Bild 1), der als Grundlage der werkzeugbasierten Verarbeitung dienen soll. Objekte können durch eine Reihe von sog. ***Objektattributen*** in ihrer Gesamtheit beschrieben werden. Die im Objekt zusammengefaßten *elementaren Objekte*, die sog. ***Elementarobjekte***, beschreiben die für die Werkzeugausführung notwendigen Daten und können durch ***Elementarobjektattribute*** vollständig beschrieben werden; sie entsprechen damit in ihrer Ausdrucksmächtigkeit etwa den Tupeln im Relationenmodell. Die Elementarobjekte sind gemäß den sog. ***Strukturbeziehungen*** (die auch als *Elementarobjektbeziehungen* bezeichnet werden) organisiert. Ein Objekt beschreibt somit alle (oder zumindest einen signifikanten Teil) der für ein konkretes Werkzeug relevanten Daten. *Unterschiedliche Zustände der im Objekt zusammengefaßten Daten werden als Objektversionen* oder kurz als ***Versionen*** *bezeichnet*. Analog den Objekten können die in Versionen zusammengefaßten Daten in ihrer Gesamtheit durch sog. ***Versionsattribute****) charakterisiert werden.

Die Ableitung einer neuen Objektversionen erfolgt i. allg. innerhalb eines Entwurfsschrittes durch die Anwendung eines Entwurfswerkzeugs auf eine (oder auch mehrere) bereits bestehende Entwurfsversionen. Die Abhängigkeiten zwischen (Ausgangs-)Versionen und den daraus entwickelten (neuen) Versionen, die die Fortentwicklung der beteiligten Objekte beschreiben, werden in Form sog. ***Abstammungsgraphen*** repräsentiert. Der Abstammungsgraph kann sich als gerichteter, linearer, baumartiger oder allgemein als azyklischer Graph entwickeln.

Neben der beschriebenen Fortentwicklung initialer Versionen ist aber auch die Erzeugung mehrerer initialer Versionen eines Entwurfsobjekts durch ein Werkzeug zu berücksichtigen. Jede dieser Versionen kann dann unabhängig voneinander weiterentwickelt werden. Hierdurch entstehen Gruppen von jeweils abhängigen Objektversionen eines Objekts, dessen Abstammungsgraph sich dann als *unzusammenhängender* Graph aus-

*) In allen nachfolgenden Bildern sind Objekt- und Versionsattribute nicht gezeigt.

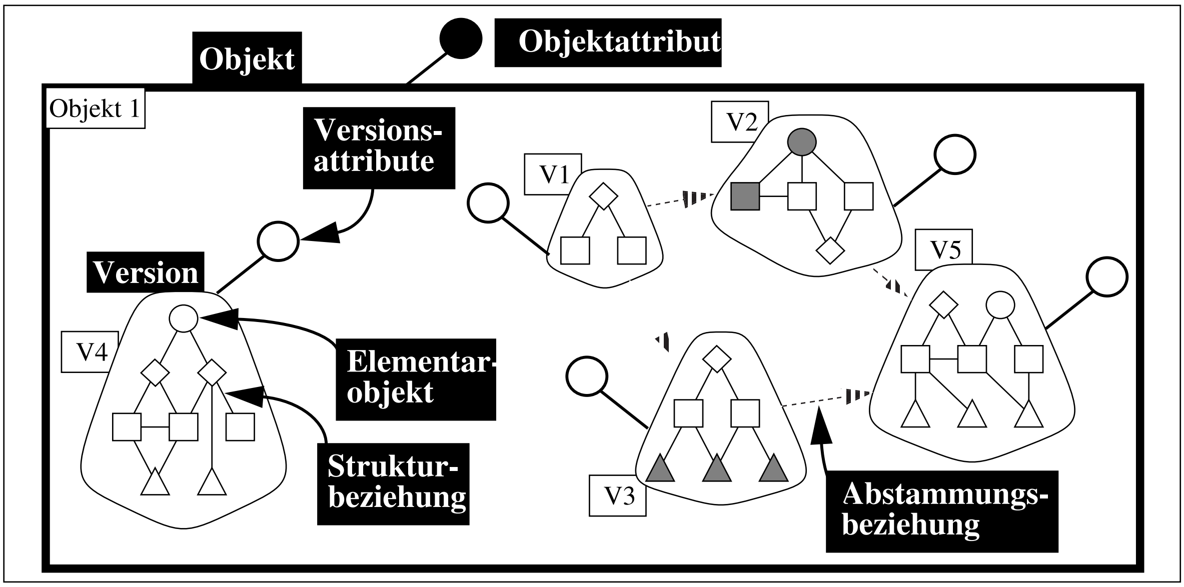

Bild 1:Objekt mit Versionen und Abstammungsgraph

bildet. Versionen, die zu unterschiedlichen Zusammenhangskomponenten des Abstammungsgraphen gehören, also nicht direkt voneinander abhängig sind, bezeichnen wir als ***alternative*** Objektversionen (Versionen 1, 2, 3 und 5 sind "alternativ" zu Version 4), da sie i. allg. durch *alternativ* ausgeführte Verarbeitungsschritte entstanden sind. Ein gutes Beispiel hierfür ist die Anwendung eines Entwurfswerkzeugs auf die gleichen Entwurfsdaten mit unterschiedlichen Steuerungsparametern.

2.2.2 Beziehungen zwischen Entwurfsobjekten bzw. Entwurfsobjektversionen

Die Beziehungen zwischen Objekten, Objektversionen oder auch alternativen Objektversionen sind sehr vielfältig. Zum einen werden sie *explizit* in Form von Objekt- und/oder Versionsbeziehungen definiert, zum anderen ergeben sie sich *implizit* durch objekt- bzw. versionsüberlappende Daten - sind also eine Folge von überlappenden Objektdefinitionen. Bild 2 illustriert die unterschiedlichen Beziehungsbegriffe, die wir im folgenden kurz charakterisieren wollen.

Objektbeziehungen repräsentieren zwischen Objekten bestehende Relationen [DL88, Wi87]. Obwohl Objektbeziehungen auch für sich alleine existieren können, ist es häufig sinnvoll, sie auf Versionen zu verfeinern. Sie werden dann als ***Versionsbeziehungen*** bezeichnet. In diesem Fall können die zugehörigen Objektbeziehungen als *Abstraktion* ihrer Versionsbeziehungen verstanden werden. Auf der anderen Seite, können Versionsbeziehungen nur in Verbindung mit einer zugehörigen Objektbeziehung existieren. Das explizite Nachführen von Versionsbeziehungen bei der Ableitung bzw. Entstehung neuer Versionen kann durch sog. ***generische*** Versionsbeziehungen [BM88, DL88, Sc91] vermieden werden. Hierbei werden an der Beziehung teilnehmende Versionen nicht fest durch ihren Identifikator, sondern dynamisch bei der Evaluierung der Beziehung beispw. mit Hilfe eines Suchausdrucks bestimmt [BM88, Sc91]; so kann etwa die jeweils zuletzt erzeugte Version als Beziehungspartner ermittelt werden.

Zusätzlich zu den beschriebenen expliziten Verknüpfungen zwischen Versionen können weitere implizite Beziehungen vorhanden sein (vgl. Bild 2). Sie ergeben sich durch Datenüberlappungen als Folge von überlappenden Objekttypdefinitionen. Die Datenüberlappungen können dabei über Elementarobjekte selbst oder über Strukturbeziehungen entstehen. Im ersten Fall werden Elementarobjekte mehreren Versionen *unterschiedlicher Objektausprägungen* zugeordnet; wir sprechen demgemäß von ***überlappenden Objektversionen*** bzw. kurz von ***überlappenden Objekten***. Ein Beispiel hierfür ist die Objektüberlappung zwischen Objekt 1 Version 4 und Objekt 3 Version 2 dargestellt in Bild 2. Grundlage des zweiten Falls sind definierte

Strukturbeziehungen. ***Intra-Strukturbeziehungen*** sind auf die aktuelle Objektversion beschränkt. Hingegen überschreiten ***Inter-Strukturbeziehungen*** die Objektversionsgrenzen. Da letztere nur im Kontext der beteiligten Objektversionen vollständig interpretiert werden können, wird durch sie auch eine Art Objektüberlappung definiert. Dieser, zur Interpretation notwendige Kontext, führt direkt zum Begriff der ***Konfiguration*** [BM88, DL88, Ka90].

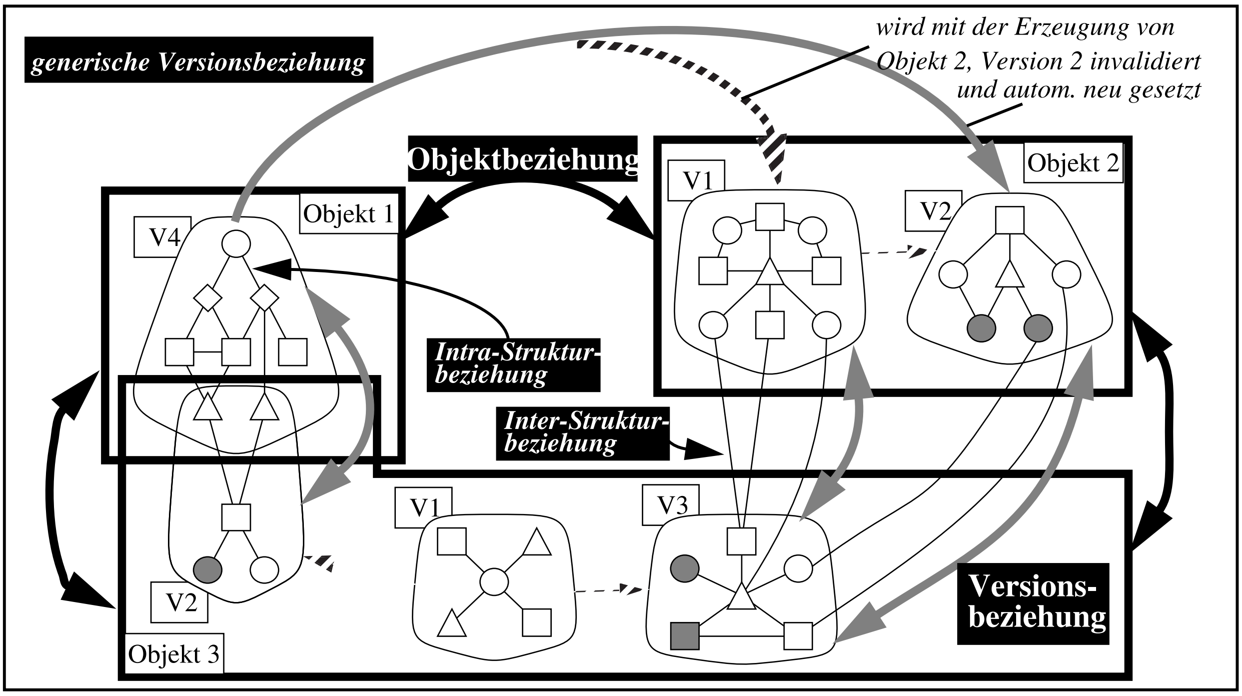

Bild 2: Objekt-, Versions- und Strukturbeziehungen

Beziehen wir uns auf Bild 2 und vergegenwärtigen wir uns, daß jede Objektversion das Objekt vollständig beschreibt; dies bedeutet, daß Version 3 von Objekt 3 entweder nur mit Version 1 oder nur mit Version 2 von Objekt 2 in Beziehung stehen darf. Um eine saubere Trennung der Inter-Strukturbeziehungen zu erreichen, müßte somit eine neue Version von Objekt 3 erzeugt werden. Dies löst jedoch einen Schneeballeffekt aus, falls weitere Inter-Strukturbeziehungen betroffen sind. Um dies zu vermeiden, setzt man zur Trennung der Inter-Strukturbeziehungen Konfigurationen ein, d.h. *Inter-Strukturbeziehungen dürfen nur innerhalb einer Konfiguration interpretiert werden*. In unserem Fall müßten also zwei Konfigurationen (Objekt 2, Version 1 mit Objekt 3, Version 3 und Objekt 2, Version 2 mit Objekt 3, Version 3) gebildet werden.

2.3 Zusammenfassung

Damit haben wir alle wesentlichen Begriffe eingeführt, die zur Strukturierung der Entwurfsdaten notwendig sind. Elementarobjekte und Elementarobjektbeziehungen erlauben die Beschreibung und Handhabung der werkzeugrelevanten Daten und ermöglichen somit eine werkzeugbasierte Verarbeitung gemäß dem in Abschnitt 2.1 vorgestellten kontext-basierten Verarbeitungskonzept. Diese sog. ***Elementarobjektebene*** realisiert die werkzeugspezifischen Aspekte und stellt eine erste Abstraktion von der eigentlichen Repräsentation der Entwurfsdaten dar. Objekte, Objektversionen und die zwischen ihnen definierten Objekt- und Versionsbeziehungen erlauben eine nochmals abstraktere Sichtweise auf die Entwurfsdaten. Diese Abstraktionsebene nennen wir auch ***Objektversionsebene***. Sie unterstützt die entwurfsspezifischen Aspekte und damit die Entwurfssteuerungsebene.

Eine wesentliche Aufgabe bei der Entwurfsdatenmodellierung besteht in der Konsistenzsicherung der Daten. Wir haben hierzu Konfigurationen eingeführt, die eine konsistente, versionsfreie Sicht auf die Entwurfsdaten garantieren. Zusätzlich müssen jedoch auch Abhängigkeiten zwischen den beiden Datenabstraktionsebenen berücksichtigt werden. Liegt eine Objektüberlappung bzw. eine Objektversionsüberlappung auf Elementarobjektebene vor, so muß diese Abhängigkeit auch auf der höheren Abstraktionsebene, also auf der Objektversionsebene sichtbar sein. Aus diesem Grund, ist eine Objektüberlappung nur dann erlaubt, wenn zwischen den beteiligten Objekten bzw. Objektversionen eine Objekt- bzw. Versionsbeziehung existiert. Damit ist auf semantisch höherer Ebene, also auf der Objektversionsebene, die Objektüberlappung auf Elementarobjektebene manifestiert und kann somit auch entsprechend interpretiert werden.

3. Das Objekt-Versions-Modell OVM

Nachdem wir die grundlegenden Strukturen von Entwurfsdaten identifiziert haben, wollen wir nun das Objekt- und Versionsmodell OVM vorstellen, das entsprechend den zuvor erarbeiteten Vorgaben konzipiert wurde.

3.1 Beispielanwendung VLSI-Entwurf

Zur Illustration und gleichzeitig auch zur Evaluierung der Adäquatheit des Objekt-Versions-Modells (OVM) wollen wir eine (vereinfachte) VLSI-Entwurfsumgebung verwenden, die wir im folgenden kurz einführen wollen. Eine ausführlichere Beschreibung dieser Entwurfsumgebung sowie des OVM findet sich in [Kä92]. Ziel des Kapitels ist es, ein vereinfachtes Informationsmodell für einige wichtige Bereiche des VLSI-Entwurfs zu entwickeln, die anschließend durch die Sprachmittel von OVM beschrieben werden sollen.

Die in [Zim86] eingeführte Entwurfsmethodik unterscheidet verschiedene Entwurfsbereiche, die jeweils durch hierarchisch angeordnete Beschreibungsebenen schrittweise verfeinert werden. Der Bereich "Verhalten" spezifiziert die Funktion (Operation, Algorithmus, Prozeß) des betreffenden Entwurfsobjekts möglichst genau. Im Bereich "Struktur" wird das Objekt in seiner realisierungsunabhängigen Zusammensetzung beschrieben; dieser umfaßt u.a. Schaltplan und Komponentenliste. Letztere besteht aus einer Modulliste und einer Netzliste. In der Modulliste werden die Module (Zellen) beschrieben, die innerhalb der aktuell zu entwerfenden Zelle als Subzellen zu plazieren sind. Die Modulbeschreibung enthält neben dem Namen weitere Parameter wie u.a. die voraussichtliche Modulfläche und die Außen-/Innenanschlüsse. In der Netzliste wird die Struktur der Zelle, d.h. die Verbindungen der Subzellen untereinander beschrieben. Weitere Bereiche machen Aussagen über den konkreten Aufbau des Entwurfsobjekts. Dazu gehört ein Floorplan (Bereich "Topographie"), der als Grobstruktur anschließend schrittweise bis zum Masken-Layout (Bereich "physikalische Realisierung") konkretisiert wird.

Da typischerweise die Anzahl der Standardzellen für eine zu entwerfende Zelle sehr groß ist (~ 10^6 Standardzellen), wird der Entwurf *hierarchisch* durchgeführt, d.h., man entwirft hierarchisch aufgebaute Zellen, die jeweils aus ca. 50 bis 100 Subzellen bestehen. Weiterhin wird ausgenutzt, daß häufig Subzellen mit gleicher Funktionalität mehrfach in einer Zelle verwendet werden. Diese Subzellen unterscheiden sich jedoch beispw. durch ihre geometrische Position innerhalb der Zelle bzw. durch ihre geometrische Form. Zur Modellierung dieser Aspekte verwendet man eine spezielle Art von Typisierung, auf die wir später nochmals eingehen werden.

Im Rahmen des sog. Chip Planning [ASZ92] wird eine Modul aus dem Bereich Struktur in den Bereich Topographie überführt. Diese Aufgabe wird i. allg. durch eine Gruppe von Entwurfswerkzeugen ausgeführt. In einem ersten Schritt wird zunächst eine Floorplan-Topologie generiert, die lediglich die relative Lage der Subzellen zueinander festlegt. Die dazu benötigten Algorithmen versuchen Kostenfunktionen zu minimieren, deren Parameter aus Modulfläche und Netzlängen bestehen. Hierauf aufbauend wird nun eine globale Verdrahtung der Subzellen gemäß der vorgegebenen Netzliste durchgeführt. In einem abschließenden Schritt kann dann die Floorplan-Topographie bestimmt werden.

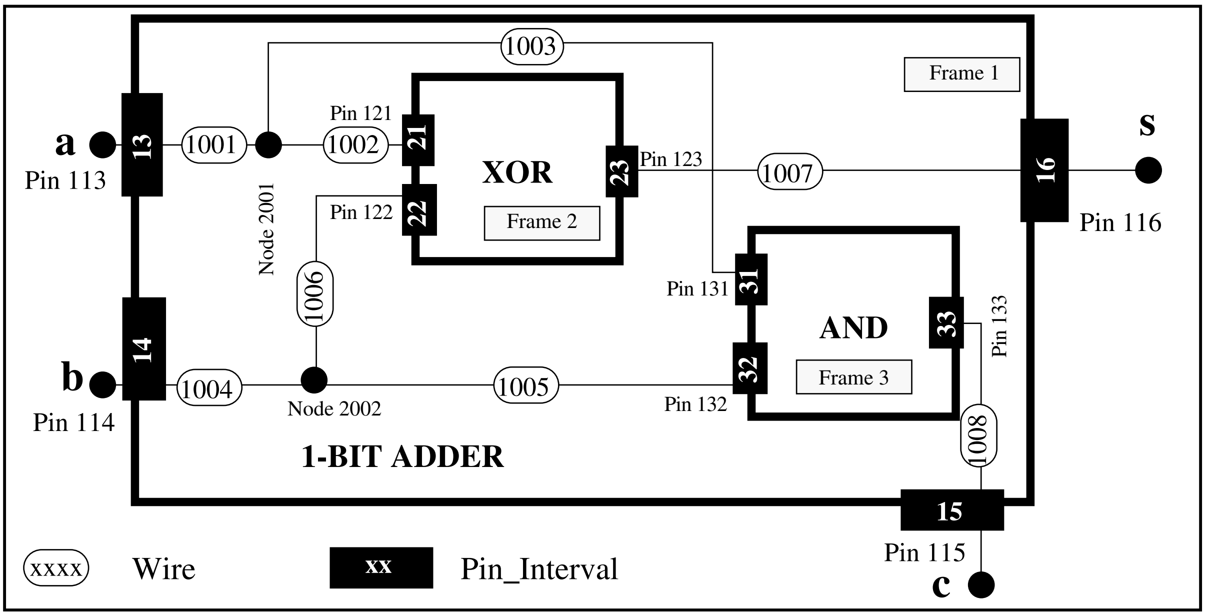

Bild 3: (Vereinfachte) graphische Darstellung des 1-Bit-Addierers

Bild 3 illustriert die vorgestellten Informationsstrukturen anhand einer vereinfachten Darstellung eines 1-Bit-Addierers. Zunächst läßt sich die ***Hierarchisierung*** innerhalb der Entwurfsdaten gut erkennen: Die Beschreibung der Zelle des 1-Bit-Addierers setzt sich direkt in der Beschreibung der Subzellen (XOR- und AND-Zelle) fort. Aus der gewählten Darstellung sind direkt ersichtlich die Modulliste (der 1-Bit-Addierer besteht aus einem AND- und einem XOR-Modul) sowie die Netzliste (also die Verbindungen zwischen den Modulen). Damit wird dann auch die topographische Anordnung der Subzellen und der Verbindungsleitungen suggeriert. Weiterhin ist das Prinzip der ***Typisierung***, also der Aufspaltung der Zellbeschreibung in ihre Schnittstellen-, ihre Implementierungs- und ihre Verwendungsinformation gut erkennbar. Die *Schnittstellen* der Zellen sind dabei jeweils durch einen Rahmen und Anschlußbereiche für Leitungen gegeben. Die *Implementierung* plaziert die Subzellen (*Instanzen*) in die aktuell zu entwerfende Zelle und verdrahtet die einzelnen Anschlußbereiche mittels durch Polygonzüge dargestellter Verdrahtungsstrecken. Die Verdrahtungsstrecken können über Kontaktstellen miteinander verknüpft werden.

3.2 Objektdefinition in OVM

Ausgehend von Bild 3 lassen sich die wesentlichen Beschreibungselemente leicht identifizieren. Zur Beschreibung der Zellen benötigen wir ihren Rahmen, sowie die darauf befindlichen Anschlußintervalle. Zur Beschreibung der Leitungen ist zum einen der Leitungsverlauf und zum anderen die Beschreibung der Kontaktstellen notwendig. Diese Elementarobjekte, werden im folgenden Kapitel mittels OVM beschrieben.

3.2.1 Elementarobjekt- und Beziehungstypen im OVM

Elementarobjekttypen werden durch Attribute unterschiedlichen Typs beschrieben und enthalten jeweils ein Attribut, das einen systemvergebenen Identifikator aufnehmen kann. Sie *können nicht versioniert werden* und entsprechen somit in ihrer Beschreibungsmächtigkeit etwa den Relationen des Relationenmodells.

Strukturbeziehungen zwischen Elementarobjekten werden durch ***Strukturbeziehungstypen*** definiert; sie können vom Typ 1:1, 1:n oder n:m sein, dürfen keine Attribute enthalten, sind jeweils auf maximal zwei Elementarobjekttypen beschränkt und können durch Kardinalitätsrestriktionen eingeschränkt werden*. Zur Beschreibung einer Zelle, wie sie beispw. in Bild 3 graphisch dargestellt ist, benötigen wir die in Bild 4 gezeigten Elementarobjekt- und Beziehungstypen. Der Rahmen einer Zelle wird durch den Elementarobjekttyp 'Frame' beschrieben. Auf dem Rahmen können in den durch 'Pin_Interval' beschriebenen Bereichen Anschlüsse erfolgen. Jeder konkrete Anschluß und dessen Position wird im Elementarobjekttyp 'Pin' vermerkt. Die Anschlüsse werden durch Leitungsbahnen ('Wire') in Form von Polygonzügen oder über Kontakte ('Node') miteinander verbunden. Die Leitungen einer Zelle werden zu Netzlisten ('Netlist') zusammengefaßt. Neben diesen sechs Elementarobjekttypen werden sechs Strukturbeziehungstypen benötigt: Zu jedem Rahmen ('Frame') müssen die zugeordneten Instanzen von 'Pin_Interval' und 'Pin' gebunden werden. Weitere zwei Beziehungstypen werden benötigt, um die Leitungsbahnen alternativ mit 'Pin'-Instanzen oder 'Node'-Instanzen verbinden zu können. Der letzte Beziehungstyp erlaubt die Zusammenfassung aller Leitungsbahnen zu einem Netz.

```
DEFINE ELEMENTARY_TYPE Frame                DEFINE ELEMENTARY_TYPE Pin_Interval
    (f_id:        IDENTIFIER,                   (pt_id:       IDENTIFIER,
     width:       REAL,                          name:        LIST_OF (BYTE),
     height:      REAL);                         position:    HULL(2));

DEFINE ELEMENTARY_TYPE Pin                  DEFINE ELEMENTARY_TYPE Netlist
    (p_id:        IDENTIFIER,                   (n_id:        IDENTIFIER);
     position:    HULL (2));

DEFINE ELEMENTARY_TYPE Node                 DEFINE ELEMENTARY_TYPE Wire
    (n_id:        IDENTIFIER,                   (w_id:        IDENTIFIER
     position:    HULL (2));                     polygon:     LIST_OF (HULL (2)));

DEFINE ELEMENTARY_LINK_TYPE Frame.to_Pin_Interval (1,32), Pin_Interval.to_Frame (1,1);
DEFINE ELEMENTARY_LINK_TYPE Pin_Interval.to_Pin (0, *), Pin.to_Pin_Interval (1,1);
DEFINE ELEMENTARY_LINK_TYPE Frame.to_Pin (1,32), Pin.to_Frame (1,1);
DEFINE ELEMENTARY_LINK_TYPE Wire.to_Pin (0,2), Pin.to_Wire (0,1);
DEFINE ELEMENTARY_LINK_TYPE Wire.to_Node (0,2), Node.to_Wire (3,*);
DEFINE ELEMENTARY_LINK_TYPE Netlist.to_Wire (1,1), Wire.to_Netlist (1,*);
```

Bild 4: Elementarschema des VLSI-Entwurfsbeispiels

Das in Bild 4 gezeigte ***Elementarschema*** modelliert nicht die Sicht eines bestimmten Werkzeugs, sondern beschreibt alle werkzeugrelevanten Entwurfsdaten für den gesamten Entwurfsbereich "Struktur". Der nächste Abschnitt beschäftigt sich mit der für den Entwurfsablauf relevanten abstrakteren Ebene der Entwurfsobjekte bzw. der Entwurfsobjektversionen.

3.2.2 Objekte, Versionen und Beziehungen im OVM

Zur Steuerung des Entwurfsablaufs, also insbesondere zur Koordinierung von Werkzeugläufen wird eine abstraktere Sicht auf die Entwurfsdaten benötigt. Hierdurch angesprochen ist die Objektversionsebene, auf der die entwurfsspezifischen Aspekte modelliert werden. Wir realisieren diese abstraktere Ebene durch die Definition ***komplexer Objekttypen***. Hierbei stehen zwei Aspekte im Vordergrund: welche entwurfsrelevanten

*) Diese Form von Beziehungen ist in Entwurfsanwendungen dominant; eine Erweiterung auf n-äre, attributierte Beziehungen ist im Rahmen des vorgestellten Modells jedoch durchaus möglich.

Daten sind für den Entwurfsablauf von Interesse und welche werkzeugrelevanten Daten sind ihnen zuzuordnen? Bild 5 zeigt das auf dem Elementarschema basierende, abstraktere ***Objektschema***. Es werden drei komplexe Objekttypen definiert:

- ’Interface’ zur Beschreibung der Schnittstelle, bestehend aus den Elementarobjekttypen ’Frame’ und ’Pin_Interval’,
- ’Instance’ zur Beschreibung der Verwendung einer Schnittstelle, bestehend aus ’Frame’ und ’Pin’ sowie
- ’Contents’ zur Beschreibung der Implementierung, also im wesentlichen der Verdrahtung einer Zelle bestehend aus ’Netlist’, ’Wire’ und ’Node’.

```
DEFINE OBJECT_TYPE Interface AS Frame, Pin_Interval VERSIONED
        (OBJECT_ATTRIBUTES:   if_id:                 IDENTIFIER,
                              if_v_no:               VERSION_NO,
                              name:                  LIST_OF (BYTE),
                              function:              LIST_OF (BYTE),
                              no_of_pins:            INTEGER,
        VERSION_ATTRIBUTES:   if_v_id:               IDENTIFIER,
                              scale_factor:          REAL,
                              no_of_feedthroughs:    INTEGER)
        VERSION DERIVATION GRAPH IS LIST;

DEFINE OBJECT_TYPE Instance AS Frame, Pin NOT_VERSIONED
        (OBJECT_ATTRIBUTES:   ist_id:                IDENTIFIER,
                              position:              HULL(2));

DEFINE OBJECT_TYPE Contents AS Netlist, Wire, Node VERSIONED
        (OBJECT_ATTRIBUTES:   c_id:                  IDENTIFIER,
                              name:                  LIST_OF (BYTE),
        VERSION_ATTRIBUTES:   ct_v_id:               IDENTIFIER,
                              ct_v_no:               VERSION_NO,
                              no_of_wires:           INTEGER,
                              length_of_wires:       REAL,
                              no_of_feedthroughs:    INTEGER)
        VERSION DERIVATION GRAPH IS TREE;
```

Bild 5: Objektschema des VLSI-Entwurfsbeispiels

Jeder dieser komplexen Objekttypen besteht i. allg. aus einer Reihe von Elementarobjekttypen[*] und wird durch Attribute in seiner Gesamtheit beschrieben (s. Klausel OBJECT_ATTRIBUTES). Jeder Objekttyp muß ein Attribut zur Aufnahme des systemvergebenen eindeutigen Objektidentifikators enthalten. Die Strukturbeziehungen zwischen den Elementarobjekttypen eines Objekttyps sind bzgl. der gewünschten Abstraktion nicht von Bedeutung und demzufolge auch nicht Teil der Objekttypdefinition. Falls Elementarobjekttypen zur Beschreibung mehrerer Objekttypen benötigt werden und entsprechend in deren Definitionen verwendet werden, so sprechen wir von überlappenden Objekttypen. In unserem Beispiel wird der Elementarobjekttyp ’Frame’ sowohl zur Beschreibung der Schnittstelle einer Zelle (’Interface’) als auch zur Beschreibung einer Zellinstanz (’Instance’) benötigt. Diese typmäßige Objektüberlappung führt zu überlappenden Objekten[†], da die Positionsangaben in den Elementarobjekten ’Pin_Interval’ und ’Pin’ nur bzgl. des konkreten Zellrahmens (’Frame’) von Bedeutung sind. Dies bedeutet, daß alle Werkzeuge, die mit diesen Positionsangaben arbeiten, auch die entsprechenden Rahmendaten benötigen. Da Objekte Einheiten der Verarbeitung darstellen, müssen somit sowohl die ’Interface’- als auch die ’Instance’-Objekte die entsprechen-

*) Die Angabe von Elementarobjekttypen zur Definition eines Objekttyps ist optional.
†) Nicht jede typmäßige Überlappung muß zu überlappenden Objekten (also Objekte, die die gleichen Elementarobjekte benutzen) führen, noch kann von nicht-überlappenden Objekttypen auf nicht-überlappende Objekte geschlossen werden.

den Informationen enthalten. Die durch die Objektüberlappung implizit gebildete Beziehung zwischen den Objekten muß durch eine entsprechend gebildete explizite Beziehung (s.u.) legitimiert werden.

Objekte können im Gegensatz zu Elementarobjekten versioniert werden, d.h., es können sowohl versionierte als auch nicht-versionierte Objekttypen definiert werden. Bei versionierten Objekten (Schlüsselwort VERSIONED) kann der Inhalt des Objekts (also nicht die Objektattribute) in mehreren unterschiedlichen Zuständen, die wir als Objektversionen bezeichnen, vorliegen. Nicht-versionierte Objekte können somit als versionierte Objekte mit genau einer Objektversion verstanden werden. Die einzelnen Objektversionen eines Objekts können analog dem Objekt selbst durch eine Reihe von Attributen beschrieben werden, die wir als Versionsattribute bezeichnen (s. Klausel VERSION_ATTRIBUTES). Eines der Versionsattribute wird wiederum zur Aufnahme eines systemvergebenen eindeutigen Identifikators benötigt. Zusätzlich ist die Definition eines zweiten Attributs zur Aufnahme einer systemvergebenen Versionsnummer notwendig, die alle Versionen *eines Objekts* aufsteigend (mit 1 beginnend) durchnumeriert[*]. Es kann zusätzlich ein Attribut des Typs TIMESTAMP zur Abspeicherung des Entstehungszeitpunktes der Version angegeben werden, das dann systemseitig gesetzt wird.

3.2.3 Objekt- und Versionsbeziehungen im OVM

Die ***Objektbeziehungstypen*** besitzen zunächst die gleichen Eigenschaften wie die schon vorgestellten Elementarbeziehungstypen zwischen Elementarobjekttypen, d.h., sie sind symmetrisch, attributfrei und binär. Es können Beziehungstypen des Typs 1:1, 1:n und n:m direkt modelliert werden. Im Gegensatz zu den Elementarbeziehungstypen können Objektbeziehungstypen jedoch von der Versionierung der Beziehungspartner betroffen sein. Da Objektversionen einem bestimmten Zustand des Objekts entsprechen, ist es zumeist notwendig, auch die Verknüpfung des Objekts in diesem Zustand zu anderen Objektversionen zu beschreiben. Dies bedeutet, daß Objektbeziehungen i. allg. zu Beziehungen zwischen Objektversionen, den sog. ***Versionsbeziehungen*** verfeinert werden. Dies bedeutet, daß V*ersionsbeziehungen nur in Verbindung mit einer Objektbeziehung existieren können*. Entsprechend können Versionsbeziehungstypen nur im Zusammenhang mit Objektbeziehungstypen definiert werden. Die Handhabung von Versionsbeziehungen verlangt nach speziellen Mechanismen, da die unabhängige Versionierung der verschiedenen Objekte auf Ebene der Versionsbeziehungen zu besonderen Problemen führt, die wir im folgenden untersuchen wollen. Betrachten wir hierzu in Bild 6 die Definition der Beziehungstypen für unser Entwurfsbeispiel.

```
DEFINE LINK_TYPE   Instantiation BETWEEN Interface, Instance
     CARDINALITY   FROM Interface TO Instance IS (0, *), FOR VERSIONS (0, *),
                   FROM Instance TO Interface IS (1, 1), FOR VERSIONS (1, 1);
DEFINE LINK_TYPE   Aggregation BETWEEN Contents, Instance
     CARDINALITY   FROM Contents TO Instance IS (1, 1000), FOR VERSIONS (1, 1000
                   FROM Instance TO Contents IS (1, 1), FOR VERSIONS (1, *);
DEFINE LINK_TYPE   Implementation BETWEEN Interface, Contents
     CARDINALITY   FROM Interface TO Contents IS (0, 1), FOR VERSIONS (0, *),
                   FROM Contents TO Interface IS (1, 1), FOR VERSIONS (1, *);
```

Bild 6: Objekt- und Versionsbeziehungen des VLSI-Entwurfsbeispiels

Zunächst wird der Objektbeziehungstyp 'Instantiation' zwischen dem versionierten Objekttyp 'Interface' und dem nicht-versionierten Objekttyp 'Instance' definiert. Durch die Klausel FOR VERSIONS wird die Verfeinerung des Beziehungstyps auf Versionsebene erreicht. Aus diesem Grund müssen neben den Kardi-

*) Die Versionsnummer TOPICAL ist die höchste vergebene und erlaubt zusammen mit dem Objektidentifikator die Adressierung der neusten Version eines Objekts.

nalitätsrestriktionen auf Objektebene auch die Restriktionen auf Versionsebene angegeben werden. Wie die Definition des Beziehungstyps 'Implementation' zeigt, können hierbei unterschiedliche Angaben* sinnvoll sein. Beispielsweise soll ein 'Instance'-Objekt in genau einem 'Contents'-Objekt verwendet werden. Liegen aber unterschiedliche Versionen des 'Contents'-Objekts vor, so sollen hierfür nicht immer wieder neue 'Instance'-Objekte erzeugt werden müssen, da anzunehmen ist, daß viele Instanzen beim Versionsübergang überhaupt nicht verändert werden. Auf Versionsebene kann deshalb ein 'Instance'-Objekt mehreren 'Contents'-Versionen zugeordnet werden. Die gleichen Überlegungen gelten für die Beziehung zwischen 'Interface' und 'Contents': Jede Schnittstelle wird durch genau eine Implementierung realisiert. Es dürfen jedoch mehrere (alternative) Versionen der Implementierung existieren. Im Gegensatz zu bisher, werden bei ***generischen Versionsbeziehungen*** die an der Beziehung teilnehmenden Versionen dynamisch, nach einem vorgegebenen Verfahren (Auswahlbedingung), bestimmt. Beispielsweise kann damit ein bequemer Zugriff von der Schnittstelle auf die jeweils neuste Version der Implementierung ermöglicht werden. Eine ausführlichere Diskussion dieses Konzeptes kann in [Kä92] nachgelesen werden.

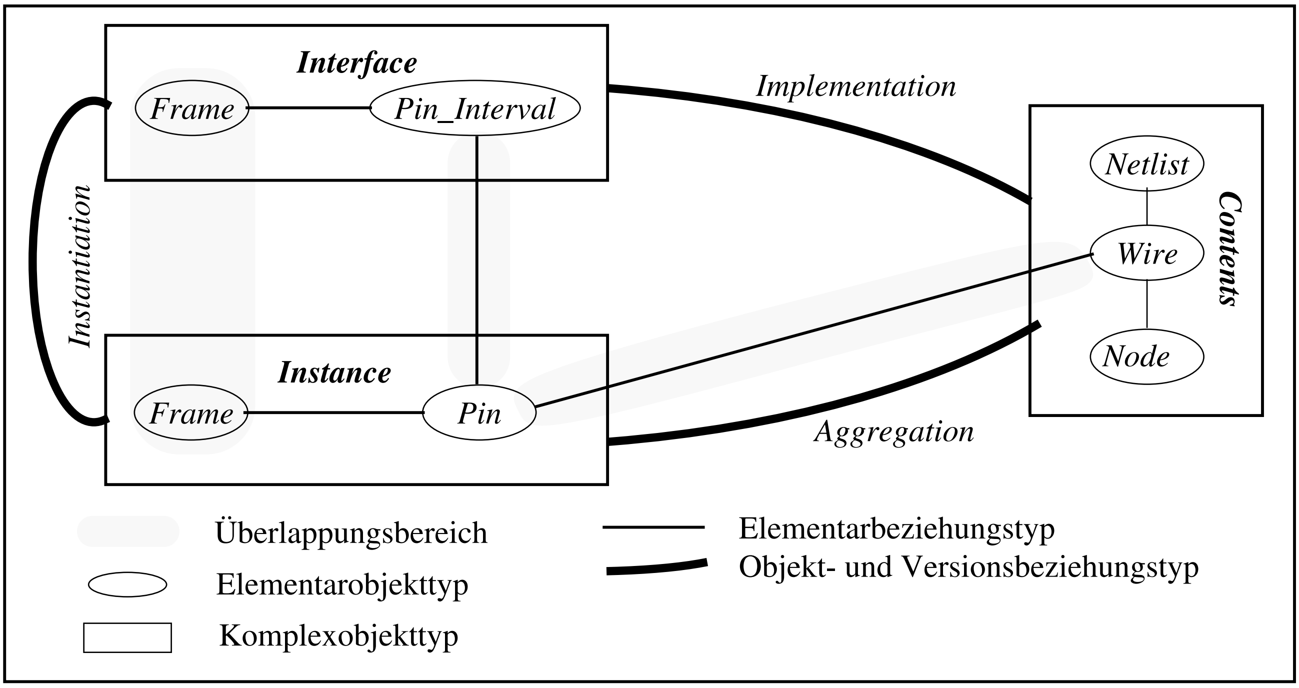

Bild 7: Graphische Darstellung des Gesamtschemas

Bild 7 zeigt eine graphische Gesamtdarstellung des entwickelten OVM-Schemas: die Elementarobjektebene abstrahiert von der eigentlichen Repräsentation der Entwurfsdaten und berücksichtigt die werkzeugspezifischen Aspekte; die Objektversionsebene unterstützt die entwurfsspezifischen Aspekte und damit die Entwurfssteuerungsebene. Die grauen Hinterlegungen heben Objekt- und Versionsüberlappungen hervor. Die in Kapitel 3.1 beschriebenen *Entwurfsbereiche* werden reflektiert durch mehr oder minder vollständige Objektversionen. Das dort ebenfalls angesprochene Prinzip der *Hierarchisierung* wird mittels der drei in Bild 7 aufgezeigten Objekt- bzw. Versionsbeziehungen realisiert: Ausgehend von einer Zelle ('Interface') wird über ihre Implementierung ('Contents') die darin enthaltenen Subzellen ('Instance') bzw. deren Schnittstellen ('Interface') erreicht. Ein solches Objektversions-Ensemble stellt einen typischen Verarbeitungskontext (in diesem Fall für das "Chip Planning"-Werkzeug) dar. Eine Sprache zur Bereitstellung solcher Verarbeitungskontexte wird im nächsten Abschnitt vorgestellt.

*) Wegen der Existenzbindung der Versionsbeziehung an die Objektbeziehung müssen die Kardinalitätsrestriktionen der Versionsbeziehung immer die Kardinalitätsrestriktionen der Objektbeziehung einschließen.

3.3 Objektmanipulation mit OML

In diesem Kapitel werden die wesentlichen Konzepte der Sprache OML (Object Manipulation Language) beschrieben. OML ermöglicht die Manipulation der Objekte, Versionen und Beziehungen in OVM. OML ist eine deskriptive, mengenorientierte Sprache. Die Form der Anweisungen sind an SQL angelehnt, d.h., für jede Operation werden in der FROM-Klausel die betroffenen Objekt- oder Beziehungstypen spezifiziert, deren Ausprägungen dann von der Operation betroffen sind. Diese können zusätzlich durch die Angabe einer WHERE-Klausel eingeschränkt werden. Bild 8 zeigt die Datenmanipulationsoperationen von OML im Überblick, von denen wir einige im folgenden näher besprechen wollen.

Operation	Beschreibung
CREATE OBJECT	Erzeugen eines Objekts, das durch seine Objektattribute beschrieben wird. Diese Operation erlaubt gleichzeitig die erste Version dieses Objekts, die durch ihre Versionsattribute und die Elementarobjekte beschrieben ist, zu erzeugen.
CREATE VERSION	Erzeugen einer Objektversion. Sie wird durch die Angabe der Versionsattribute und der Elementarobjekte beschrieben. Die Angabe der zugehörigen Objektausprägung bzw. der Vorgängerversion(en) im Abstammungsgraphen ist notwendig.
DELETE OBJECT	Löschen eines Objekts (inklusive seiner Versionen).
DELETE VERSION	Löscht eine einzelne Version eines Objekts mit allen Elementarobjekten.
UPDATE OBJECT	Ändern der Objektattribute (die “alten” Werte werden überschrieben).
UPDATE VERSION	Ändern der Versionsattribute (die “alten” Werte werden überschrieben).
CONFIGURE	Konfigurieren von Versionen.
CREATE LINK	Setzen von Objekt- und Versionsbeziehungen.
DELETE LINK	Auflösen von Objekt- und Versionsbeziehungen.

Bild 8: Die Datenmanipulationsanweisungen von OML

In der ersten Anweisung von Bild 9 wird ein Schnittstellenobjekt erzeugt, das keine Version enthält. Einige der Objektattribute werden mit Werten belegt. Die zweite Anweisung erzeugt eine Schnittstellenversion, die mittels der Angabe in der WHERE-Klausel explizit zu dem zuvor kreierten Schnittstellenobjekt gebunden

```
CREATE Interface  (OBJECT:    name := "1-Bit Adder",                  ⇒ if_id = 4711
                              function := "s, c := a + b",
                              no_of_pins := 4)
FROM Interface;
⇒ das Schnittstellenobjekt erhält den Identifikator 4711 (if_id).
CREATE Interface  (VERSION:   scale_factor := 1.0,                    ⇒ if_v_no = 1
                              no_of_feedthroughs := 0)
                  Frame       (width := 2.1,                          ⇒ f_id = 1
                               height := 5.8)
                  Pin_Interval (name := "x",                          ⇒ pt_id = 13
                               position := [(0,1.4),(0,1.7)])
                  :
                  Pin_Interval (name := "s",                          ⇒ pt_id = 16
                               position := [(5.8, 1.2),(5.8,1.6)]);
FROM Interface
WHERE if_id = 4711;
```

Bild 9: Erzeugen eines Schnittstellenobjekts und seiner ersten Version wird. Neben der Angabe der Versionsattribute müssen auch alle Elementarobjekte mit Werten belegt und abgespeichert werden; dies umfaßt auch die zwischen den Elementarobjekten bestehenden Strukturbeziehungen. Da die Version nur ein Elementarobjekt des Typs Frame umfaßt und nur ein Beziehungstyp zwischen 'Frame' und 'Pin_Interval' definiert ist, werden alle Elementarobjekte des Typs 'Pin_Interval' automatisch mit diesem verbunden.

Zur Datenwiedergewinnung stehen in OML die Operationen SELECT OBJECT und SELECT VERSION zur Verfügung, die auf der Objekt- bzw. auf der Objektversionsebene arbeiten und entsprechend Daten auf Objekt- oder Versionsebene aus der Datenbank extrahieren. Bild 10 zeigt zwei unterschiedliche Selektionsanweisungen, von denen die erste Anweisung bewußt die Tatsache der Versionierung zur Gewinnung einer abstrakteren Sicht auf die Daten unterbindet. Die zweite Anweisung operiert hingegen explizit auf der Versionsebene.

```
(1)  SELECT OBJECT Interface (function, no_of_pins), Contents (ALL)
     FROM    Interface - <Implementation> Contents
     WHERE   Interface.name = "1-Bit Adder";

(2)  SELECT VERSION Interface (ALL),
             Contents (name, length_of_wires, Wire (ALL), Node (position) ), Instance (ALL)
     FROM    Interface - (<Implementation> Contents, <Instantiation> Instance)
     WHERE   (Interface.name = "1-Bit Adder") AND (Contents.ct_v_no = 3);
```

Bild 10: Selektion eines (versionierten) Objekts

In der Projektionsliste werden analog SQL die Attribute von Elementarobjekten, Versionen und Objekten aufgelistet, die im Ergebnis enthalten sein sollen. Hierbei ist es möglich, auch Objekte und Versionen auszublenden, falls diese nicht den Zusammenhang der in der FROM-Klausel spezifizierten Objekttypstruktur zerstören. Die FROM-Klausel beschreibt einen gerichteten Teilgraphen des OVM-Schemas. Hierzu werden abwechselnd die benutzten Objekt- und Beziehungstypen aufgezählt*. Natürlich können hierbei gerichtete Versionsbeziehung nur gemäß ihrer Definition in einer Richtung verwendet werden, während die symmetrischen Elementarbeziehungstypen in jeder der beiden Richtungen benutzbar sind. Anweisung 1) aus Bild 10 zeigt eine Selektionsanweisung, die keine Versionierung beachtet und sich auf eine einfache Objekthierarchie bezieht. Es werden ausschließlich Objekte und Objektattribute von der Schnittstelle und zugehöriger Implementierung des 1-Bit-Addierers selektiert. Die zweite Anweisung selektiert zu allen Schnittstellenversionen des 1-Bit-Addierers die zugehörige dritte Implementierungsversion (bzw. die davon in der Projektionsklausel angeforderten Daten) und Instanzierungen. Die hierzu notwendige Verzweigung des Objektypgraphen wird mit Hilfe der angegeben Klammerung '(... , ...)' spezifiziert.

3.4 Zusammenfassung und verwandte Arbeiten

OVM bietet adäquate Konzepte zur Modellierung von werkzeugrelevanten Daten (Elementarobjektebene) als auch zur Modellierung entwurfsspezifischer Aspekte (Objektversionsebene) der Entwurfsdaten an. Die Sprache OML erlaubt darüberhinaus eine angepaßte Verarbeitung der so beschriebenen Objektnetze. So können etwa die Anweisungen aus Bild 10 als Spezifikationen für verschiedene Verarbeitungskontexte interpretiert werden. Die dort benutzten Versionsstrukturen werden ausgewertet und liefern als Ergebnis eine vesionsfreie Sicht auf die spezifizierten Werkzeugdaten, die anschließend im werkzeuglokalen Verabeitungsbereich bereitgestellt werden.

---

*) Beziehungstypen müssen dabei in spitze Klammern ("<...>") eingeschlossen werden.

Unser Modell- und Sprachansatz unterscheidet sich von anderen [BB84, BK85, BM88, DL88, Ka90, Sc91, Wi87] im wesentlichen durch die gleichrangige Unterstützung von Elementarobjekt- und Objektebene sowie in der flexiblen und deskriptiven Handhabung der Objektstrukturen. Die explizite Trennung der beiden angesprochenen Ebenen ist in keinem der zitierten Ansätze enthalten. .[Ka90] konzentriert sich auf die Unterstützung der Objektebene. Zusätzlich zum vorgestellten Ansatz werden 'Repräsentationen' von Objekten unterschieden, die durch sog. 'Äquivalenzrelationen' verbunden werden. Auf der anderen Seite wird nur eine starre Realisierung des Abstammungsgraphen (ein Abstammungsbaum) angeboten. [DL88] definieren ein allgemeines Basisversionsmodell, das jedoch ein starres Konzept für die Organisation der Versionen bzw. der versionierten Daten festlegt. Weiterhin werden benutzerspezifische Konfigurationen unterstützt. Eine deskriptive Auswahl von Versionsnetzen bzw. der zugehörigen Daten ist nicht vorgesehen. [BB84, BK85] stellen ein speziell auf den VLSI-Entwurf zugeschnittenes Versionsmodell vor; so ist die Trennung von 'Schnittstelle' und 'Implementierung' systeminhärent. Dies erlaubt natürlich eine starke Anwendungsunterstützung schränkt aber gleichzeitig den Anwendungsraum des Modells ein. Es wird, wie bei den vorigen Modellen, keine spezielle Unterstützung zur Verarbeitung von Elementarobjektnetzen angeboten. [Wi87] beschreibt ein allgemeines Versionsmodell, das mittels 'Objekt-Generalisierung' und 'Objekt-Assoziation' eine relativ flexible Beschreibung von Objekt- und Versionstrukturen erlaubt. Insbesondere die Verwendung von 'Partitionen' und 'Graphen' zur Beschreibung der Assoziation ermöglichen die Nachbildung der meisten vorgestellten Modelle. Die vorgestellte Sprache vHDBL erlaubt darüberhinaus sowohl den Zugriff auf versionierte komplexe Objekte als auch auf nicht versionierte Objekte. Aspekte der Konfigurierung bzw. von Objekt- und Versionsbeziehungen auf der einen Seite sowie eine spezielle Unterstützung der Elementarobjektverabeitung auf der anderen Seite werden nicht adäquat behandelt.

Insegesamt gesehen zeichnet sich der hier vorgestellte Ansatz OVM durch eine adäquate Behandlung von Objektnetzen und einer hohen deskriptiven Auswahlmächtigkeit sowie durch eine explizite Trennung zwischen Elementarobjekt- und Objektversionsebene aus. Letzteres ermöglicht, daß Versions- und Konfigurationsaspekte unabhängig von Datenbereitstellungsaspekten (etwa zur effizienten Navigation entlang von Strukturbeziehungen) betrachtet werden können. Wir werden auf diesen Aspekt nochmals in Kapitel 4 zurück kommen.

4. Realisierung von OVM/OML

Die Realisierung unseres durch OVM und OML gegebenen Objektmodells kann auf recht vielfältige Art und Weise erfolgen. In diesem Kapitel beschreiben wir eine Realisierung, die an der Universität Kaiserslautern durchgeführt wurde und detailliert in [Kä92, KS92] beschrieben ist. Als Basissystem wurde das ebenfalls an der Universität Kaiserslautern entwickelte Non-Standard-DBS (NDBS) PRIMA [HMMS87, GG92] gewählt. Momentan wird die hier vorgestellte Implementierung von OVM/OML in der DB-basierten Entwurfsumgebung PRIMA-Framework [GHM92] getestet.

4.1 Architektur

Die DB-basierte Entwurfsumgebung PRIMA-Framework [GHM92, Su92], skizziert in Bild 11, besteht im wesentlichen aus zwei Komponenten: der Objekt- und Versionsverwaltung (OVV) und der Entwurfsablaufsteuerung (EAS). Beide zusammen ermöglichen ein einfaches ein- und aushängen von CAD-Werkzeugen. Die Komponente OVV implementiert das Objektmodell OVM und stellt an ihrer Schnittstelle die Sprache OML zur Verfügung. Die Komponente EAS regelt den werkzeugbasierten Entwurfsprozeß. Ausgehend von einem festgelegten Ablaufplan steuert sie die Aktivierung einzelner Werkzeuge sowie die, für die Werkzeuge notwendige Ver- und Entsorgung von werkzeugrelevanten Entwurfsdaten. Diese werden mittels OML-Anweisungen bei der OVV angefordert (vgl. Schritt (1) in Bild 11), aus dem Entwurfsdatenbestand extrahiert (Schritt (2)) und in den Objektpuffer eingelagert (Schritt (3)). Der Objektpuffer dient als werkzeuglokaler Arbeitsbereich, auf den das Werkzeug effizient zugreifen und auch Datenänderungen durchführen kann (Schritt (4)).

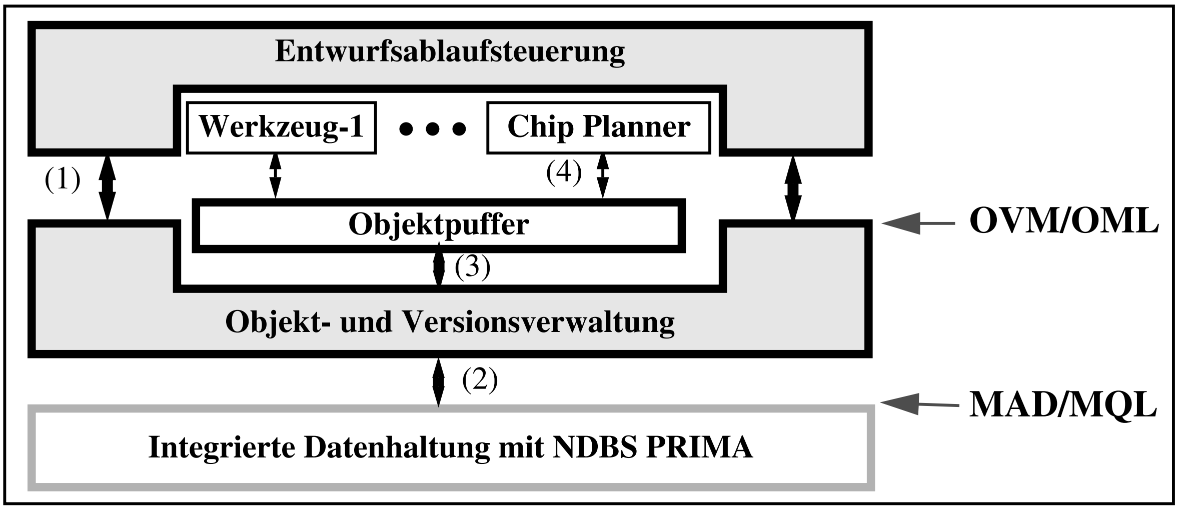

Bild 11: Grobarchitektur von PRIMA-Framework

Die Zuordnung der einzelnen Schritte zu den 6 Phasen eines werkzeugspezifischen Entwurfsschritts (s. Abschnitt 2.1) ist offensichtlich und kann leicht nachvollzogen werden. Es sei hier nur erwähnt, daß die Schnittstelle von der OVV zum Objektpuffer (in Bild 11 mit (3) markiert) eine versionsfreie Werkzeugsicht unterstützt. Im Gegensatz dazu ist die Schnittstelle von der OVV zur EAS (mit (3) markiert) versionsbehaftet und realisiert entwurfsspezifische Aspekte. An der Objektpufferschnittstelle ist die Semantik des Elementarschemas bekannt und es werden daher nur diesbezüglich konsistente Strukturen aufgebaut, die dann an die OVV übertragen und hinsichtlich der Entwurfssemantik geprüft werden, bevor sie dann der Datenhaltung übergeben werden. Dazu hat die OVV-Komponente eine Abbildung von der OVM-Ebene auf die Ebene des Datenmodells der Datenhaltungskomponente zu leisten.

4.2 Abbildung von OVM/OML auf MAD/MQL

Das NDBS PRIMA [GG92] ist eine Erweiterung des Relationenmodells um ein flexibles Komplexobjekt-Konzept. PRIMA implementiert das Molekül-Atom-Datenmodell (kurz MAD) und die mengenorientierte Anfragesprache MQL. MAD/MQL erlaubt die Verarbeitung von Komplexobjekten (Moleküle), die aus Elementarobjekten (Atomen) aufgebaut und über definierte Strukturbeziehungen zusammengesetzt sind. Es wird keine Versionierung dieser Daten unterstützt. Die Strukturbeziehungen werden in MAD/MQL über Referenzattribute gebildet, die die Identifikatoren der zu referenzierenden Atome enthalten. Um symmetrische Beziehungen zu garantieren, werden immer Paare von (zueinander entgegengesetzt gerichteten) Referenzattributen verlangt. Details über die PRIMA-Implementierung und MAD/MQL sind u.a. [GG92, Mi88] zu entnehmen.

Zur Abbildung von OVM/OML auf MAD/MQL muß die OVV-Komponente zum einen ein OVM-Schema als MAD-Schema realisieren und zum anderen die OML-Operationen auf entsprechende MQL-Operationen abbilden. Details können in [Kä92, KS92] nachgelesen werden. Hier beschränken wir uns auf die prinzipielle Vorgehensweise.

Die Konzeption der Elementarobjekttypen ist stark durch die Ideen von MAD beeinflußt, so daß deren Abbildung im wesentlichen syntaktischer Art ist. Die Definition eines Elementarobjekttyps wird durch eine entsprechende Atomtypdefinition und die Definition eines Strukturbeziehungstyps entsprechend durch ein Paar von Referenzattributen realisiert. Diese Vorgehensweise liegt darin begründet, daß die Verarbeitung der Elementarobjekte i. allg. durch Entwurfswerkzeuge erfolgt, die komplexe Algorithmen auf den Daten ausführen. Die Unterstützung solcher Verarbeitungsstrategien (in einem Objektpuffer) ist eine wesentliche Eigenschaft von MAD, so daß hier sinnvollerweise die Konzepte und Methoden von MAD eingesetzt werden. Die Abbildung der Objekt- und Versionstypen erfolgt mit Hilfe weiterer Atomtypdefinitionen sowie der Definition weiterer Referenzattribute. Die Abbildung von Objekt- und Versionsbeziehungen ergibt sich in natürlicher Weise aus der Objekttypabbildung: sie werden jeweils durch Referenzattributpaare in den Objektrepräsentanten und in den Versionsrepräsentanten realisiert.

```
                                                                        MQL
(1)  SELECT Interface (function, no_of_pins), Contents
     FROM   Interface.Implementation - Contents
     WHERE  Interface.name = "1-Bit Adder" ;

(2)  SELECT Interface, Interface_Version, Contents(name),
            Contents_Version(length_of_wires), Wire, Node (position), Instance, Instance_Version
     FROM   Interface_Version - ( .object - Interface,
                                  .Implementaion - Contents_Version - ( .object - Contents,
                                                                        .ref_Wire - Wire,
                                                                        .ref_Node - Node )
                                  .Instantiation - Instance_Version - (  .object - Instance,
                                                                        .ref_Frame - Frame,
                                                                        .ref_Pin - Pin ) )
     WHERE  (Interface.name ="1-Bit Adder") AND (Contents_Version.ct_v_no = 3);
```

Bild 12: Beispiel zur Abbildung der Objekt- und Versionsselektion auf MQL

Zur Beschreibung der Abbildung von OML-Anweisungen auf MQL-Anweisungen sei wiederum auf [Kä92, KS92] verwiesen. An dieser Stelle soll ein einfaches Beispiel genügen. Die zu den beiden in Bild 10 dargestellten OML-Anweisungen zugehörigen äquivalenten MQL-Anweisungen sind in Bild 12 gezeigt. Dabei wird deutlich, daß im wesentlichen jede Klausel einzeln abgebildet wird. Die FROM-Klausel in OML wird modifiziert entsprechend der Schemaabbildung für die Objekt-und Versionsstrukturen (s.o.) und die Projektionsklausel der MQL-Anweisung übernimmt die Angaben aus der OML-Anweisung, wobei allerdings eine Aufteilung der Objekt- und Versionsattribute auf die entsprechenden Repräsentanten erfolgt. Die WHERE-Klausel ist, wie am Beispiel verdeutlicht, nach dem gleichen Prinzip auf die entsprechenden Repräsentanten anzupassen.

4.3 Einsatzerfahrung, Bewertung und Weiterentwicklung

Mit dem praktischen Einsatz unseres PRIMA-Frameworks wird auch gleichzeitig die OVV-Implementierung von OVM/OML getestet. Unser aktueller Einsatzbereich ist der Chip-Entwurf. Dafür haben wir einen Chip-Planner [ASZ92] implementiert und als Werkzeug in den PRIMA-Framework integriert, d.h., die Datenversorgung für eine versionsfreie Werkzeugsicht läuft über die OVV. Weiterhin existiert eine (noch primitive) Entwurfsablaufsteuerung. Sie stellt auch Kooperationskonzepte [HKS92] für den werkzeugbasierten Entwurf bereitstellt, die im wesentlichen auf den vorhandenen Objekt- und Versionskonzepten basieren und daher die Objektversionsebene von OVM/OML benutzen. Die gewonnenen Erfahrungen zeigen, daß die von der OVV bereitgestellte Schnittstelle sich sowohl funktional als auch bzgl. der Abstraktionsebene als geeignet erwiesen hat. MAD/MQL kennt kein Versionenkonzept, bietet dafür aber mit seiner dynamischen Molekülbildung (entlang vordefinierter Strukturbeziehungen) ein mächtiges Werkzeug zur Handhabung von Versionsstrukturen (etwa des Abstammungsgraphen) [KS92]. Werkzeugbasierter Entwurf, Entwurfssteuerung, Kooperation, Verarbeitungskontexte sind nur einige der Konzepte, die wichtig sind für CAD-Frameworks und basierend auf OVM/OML realisiert werden konnten. Trotz dieser erfreulichen ersten Akzeptanz, ist die OVV-Implementierung noch nicht abgeschlossen. Eine wichtige Weiterentwicklung betrifft die Optimierung der generierten MQL-Anfragen. Hierzu wurden schon Konzepte (Algebraische Optimierung, Parallele Anfrageverarbeitung) erarbeitet und z.T. auch schon realisiert [HMS92, Schö90].

4.4 Alternative Realisierungen

Aufgrund des gewählten Architekturkonzeptes werden auch andere Realisieungsmöglichkeiten für OVM/OML ermöglicht. Im folgenden wollen wir einige dieser Alternativen etwas näher beleuchten und bewerten. NF2-artige DBS [DK86, SPSW90] scheinen aufgrund der stark vernetzten Entwurfsdaten (s. Bild 2) weniger gut geeignet zu sein. Objektorientierte DBS (OODBS) hingegen unterstützen netzwerkartige Datenstrukturen, besitzen aber auf der anderen Seite oft nur eingeschränkte Anfragesprachen, da sie (zumindest ursprünglich) nur für eine direkte Objektpuffer-Verarbeitung vorgesehen waren. Diese Verarbeitungsweise zielt darauf ab, Objekte mehr oder weniger direkt (und ohne Umformungen) im Objektpuffer bereitzustellen. Dabei werden Objekte meistens über Fehlerbedingungen (etwa: "Referenziertes Objekt ist nicht im Puffer") angefordert. Diese Arbeitsweise steht in direktem Widerspruch zur werkzeugbasierten Verarbeitung, welche durch das Anfordern eines ganzen Verarbeitungskontextes charakterisisert ist. Hier werden Fehlerbedingungen im vorhinein ausgeschlossen und zudem dem DBS vielfältige Möglichkeiten zur Anfrageoptimierung eröffnet. Auch die mangelnde Sichtenunterstützung in OODBS ist ein wichtiges Bewertungskriterium, da die werkzeugspezifischen Daten (als Verarbeitungskontext via OML-Anweisung angefordert) im wesentlichen Sichten über dem OVM-Schema darstellen. Natürlich sind die Systemerfahrungen, die mit OO-Pufferverarbeitung gemacht wurden, nützlich und ergänzen die Objektpufferkonzepte für Komplexobjekt-DBS. Überraschenderweise finden die Abstraktionskonzepte (Generalisierung, Klassenhierarchie, Vererbung) und auch das Methodenkonzept keine direkte Anwendung. Ein anderer DBS-Ansatz ist XNF [MPP93], eine Erweiterung von SQL. Dort werden Komplexobjekte mittels entsprechenden Anfragekonstrukten über relationalen Datenbanken aufgebaut. Damit können ähnliche Objektstrukturen wie in MAD/MQL aufgebaut und verarbeitet werden. Eine OVV-Realisierung unter Verwendung von XNF kann daher ähnlich bewertet werden wie unsere PRIMA-Realisierung.

5. Resümee

Mit diesem Aufsatz hoffen wir, trotz der Kürze der Darstellung, gezeigt zu haben, daß ein Objektmodell, welches ein Versionskonzept, unterschiedliche Datenabstraktionsebenen (Elementarobjektebene und Objektversionsebene) und eine zugehörige Manipulationssprache (mit Sichtenbildung) unterstützt, gewinnbringend als flexible Entwurfsdatenverwaltung in CAD-Frameworks eingesetzt werden kann. Aufgrund der vorhandenen Abstraktionsebenen, kann Unabhängigkeit von der konkreten Realisierung erreicht werden. Damit besteht die Möglichkeit neue Realisierungskonzepte (etwa aus dem Bereich von OODBS oder Komplexobjekt-DBS) in einfacher Weise zugänglich zu machen. Nach aktuellem Wissensstand läßt sich sagen, daß Komplexobjekt-DBS, die Netzwerkstrukturen aufbauen und verarbeiten können, als Realisierungsbasis prinzipiell in Frage kommen. Über die Verwendbarkeit von OODBS kann momentan noch nicht endgültig entschieden werden. Einige der hier erwähnten Unzulänglichkeiten von OODBS bilden aktuelle Forschungsthemen und lassen für die nahe Zukunft auf deutliche Verbesserungen hoffen.

6. Literatur

ASZ92 Altmeyer, J., Schürmann, B., Zimmerman, G.: Three-Phase Chip Planning - An Improved Top-Down Chip Planning Strategy, to appear in: Proc. of the Int. Conf. on CAD (ICCAD), Santa Clara, Calif., 1992.

BB84 Batory, D.S., Buchman, A.P.: Molecular Objects, Abstract Data Types and Data Models: A Framework, in: Proceedings of the 10th VLDB, Singapore, 1984, S. 172-184.

BK85 Batory, D., Kim, W.: Modeling Concepts for VLSI CAD objects, ACM TODS, Vol. 10, No. 3, S. 322-346.

BM88 Beech, D., Mahbod, B.: Generalized Version Control in an Object-Oriented Database, in: Proc. of the 4th Int. Conf. on Data Engineering, 1988, S. 14-22.

DK86 Dadam, P., Küspert, K. et al.: A DBMS Prototype to Support Extended NF2 Relations: An Integrated View on Flat Tables and Hierarchies, in: Proc. of the ACM SIGMOD Conf., Washington D.C., 1986, S. 356-367.

DL88 Dittrich, K.R., Lorie, R.A.: Version Support for Engineering Database Systems, IEEE TOSE, Vol. 14, No. 4, 1988, S. 429-437.

GG92 Gesmann, M., Grasnickel, A., et al.: Eine Einführung in PRIMA, Forschungsbericht Nr. 20/92, SFB 124, Universität Kaiserslautern, 1992.

GHM92 Gesmann, M., Härder, T., Mitschang, B., et al.: Supporting Cooperative Design in a DBMS-based Design Environment, Forschungsbericht Nr. 29/92, SFB 124, Universität Kaiserslautern, 1992.

HKS92 Hübel, C., Käfer, W., Sutter, B.: Controlling Cooperation Through Design-Object Specification - a Database-oriented Approach, in: Proc. of the European Design Automation Conference, Brussels, Belgium, 1992, S. 30-35.

HMMS87 Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.: PRIMA - A DBMS Prototype Supporting Engineering Applications, in: Proc. of the 13th VLDB, Brighton, 1987, S. 433-442.

HMS92 Härder, T., Mitschang, B., Schöning, H.: Query Processing for Complex Objects, in: Data and Knowledge Engineering 7, 1992, S. 181-200.

HNST90 Harrison, D., Newton, R., Spickelmier, R., Barnes, T.: Electronic CAD Framework, in: Proc. of the IEEE, Vol. 78, No. 2, 1990, S. 393-417.

Ka90 Katz, R.: Toward a Unified Framework for Version Modeling in Engineering Databases, ACM Computing Surveys, Vol. 22, No. 4, 1990, S. 375-408.

Kä91 Käfer, W.: A Framework for Version-based Cooperation Control, Proc. of the 2nd Symposium on Database Systems for Advanced Applications, Tokyo, Japan, 1991, S. 527-536.

Kä92 Käfer, W.: Geschichts- und Versionsmodellierung komplexer Objekte - Anforderungen und Realisierungsmöglichkeiten am Beispiel des NDBS PRIMA, Dissertation am Fachbereich Informatik der Universität Kaiserslautern, 1992.

KS92 Käfer, W., Schöning, H.: Mapping a Version Model to a Complex Object Data Model, Proc. of the 8th Int. Conf. on Data Engineering, Tempe, Arizona, 1992.

Mi88 Mitschang, B.: Ein Molekül-Atom-Datenmodell für Non-Standard-Anwendungen - Anwendungsanalyse, Datenmodellentwurf und Implementierungskonzepte, Dissertation, Universität Kaiserslautern, IFB 185, Springer Verlag, 1988.

MPP93 Mitschang, B., Pirahesh, H., Pistor, P., Lindsay, B., Südkamp, N.: SQL/XNF - Processing Composite Objects as Abstractions over Relational Data, in: Proc. of Ninth Int. Conf. on Data Engineering, Wien, April 1993.

RS92 Rammig, F.J., Steinmüller, B.: Frameworks und Entwurfsumgebungen, in: Informatik Spektrum, Springer-Verlag, Band 15, Heft 1, Februar 1992, S. 33-43.

Sc91 Sciore, E.: Multidimensional Versioning for Object-Oriented Databases, in: Proc. of the 2nd Conf. on Deductive and Object-Oriented Databases, München, 1991, S. 355-370.

Schö90 Schöning, H.: Realisierungskonzepte für die parallele Bearbeitung von Anfragen auf komplexen Objekten, in: Härder, T., Wedekind, H., Zimmermann, G.(Hrsg.): Entwurf und Betrieb verteilter Systeme, IFB 264, Springer Verlag, 1990, S. 204-220.

SPSW90 Schek, H.-J., Paul, H.-B., Scholl, M.H., Weikum, G.: The DASDBS Project: Objectives, Experiences, and Future Prospects, in: IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, 1990, S. 25-43.

Su92 Sutter, B.: Ansätze zur Integration in technischen Entwurfsanwendungen - angepaßte Modellierungswerkzeuge, durchgängige Entwurfsunterstützung, datenorientierte Integration, Dissertation am Fachbereich Informatik der Universität Kaiserslautern, 1992.

Wi87 Wilkes, W.: Der Versionsbegriff und seine Modellierung in CAD/CAM-Datenbanken, Dissertation, Fern Universität / Gesamthochschule Hagen, 1987.

Zi86 Zimmermann, G.: Top-Down Design of Digital Systems, in: Logic Design and Simulation, E. Hörbst (Hrsg.), Elsevier Science Publ., B. V., 1986.