DATA MANAGEMENT FOR VLSI DESIGN IN AN OPEN AND DISTRIBUTED ENVIRONMENT

T.G.R.M. van Leuken

DATA MANAGEMENT FOR VLSI DESIGN
IN
AN OPEN AND DISTRIBUTED
ENVIRONMENT

DATA MANAGEMENT FOR VLSI DESIGN IN AN OPEN AND DISTRIBUTED ENVIRONMENT

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Delft, op gezag van de Rector Magnificus, prof. dr. J.M. Dirken, in het openbaar te verdedigen ten overstaan van een commissie aangewezen door het College van Dekanen, op donderdag 24 maart 1988, te 16.00 uur door

T.G.R.M. van Leuken

elektrotechnisch ingenieur

geboren te 's-Gravenhage

Dit proefschrift is goedgekeurd door de promotor,
Prof.dr.ir. P. Dewilde.

Stellingen behorende bij het proefschrift:

Data Management For VLSI Design In
An Open And Distributed Environment

door
T.G.R.M. van Leuken

1. *Een data schema, waarin de structuur en de organisatie van data wordt beschreven en waarin ook de voorwaarden waaraan deze data moet voldoen tot uitdrukking worden gebracht, is een handig instrument om een omgeving op een abstracte manier te beschrijven.*

2. *Als in een interactieve ontwerpomgeving, de ontwerper wordt bijgestaan door een data management systeem, dan zal deze ontwerper efficienter kunnen werken, omdat het data management systeem een groot aantal administratieve taken van de ontwerper overneemt.*

3. *Als er, net als voor grafische toepassingen, ook een software standaard komt voor ontwerpprogramma's, die de opslag en toegang tot ontwerp data reguleert, dan is het mogelijk geworden om ontwerpprogramma's van verschillende bronnen samen te voegen in één systeem.*

4. *Ontspannen werken in een omgeving welke bestaat uit werkstations met gedistribueerde opslag capaciteit, valt en staat met de betrouwbaarheid en efficientie van het computer netwerk.*

5. *Zolang India problemen heeft met betrouwbare water - en stroom voorzieningen, zal dit land geen grote concurrent worden op het gebied van het ontwerpen van geintegreerde schakelingen.*

6. *Als een bedrijf een software pakket heeft ontwikkeld, dat de potentie heeft een wereldstandaard te worden, dan moet dit bedrijf deze software in het 'public domain' plaatsen. De gevolgen zijn dan dat iedereen deze software zal gaan gebruiken, waarna het bedrijf geld zal*

gaan verdienen aan alle randprodukten van dit pakket.

7. *De beste methode, om de ontwikkeling en evolutie van een data management systeem en de daarin gebruikte programma's te ontkoppelen, is een transactie schema en een aantal basis objecten te definiëren, en deze te gebruiken in een software interface tussen het data management systeem en de programma's.*

8. *Studenten en medewerkers van de TU Delft zouden er mee gebaat zijn, als de gebouwen van de TU Delft 24 uur per dag toegankelijk zouden zijn, opdat zij zo efficient mogelijk gebruik kunnen maken van de beschikbare computer apparatuur.*

9. *Als het aantal bezoekers van een toneel, dans of opera gezelschap daalt, en de kwaliteit van dit gezelschap niet verbeterbaar is, dan is de beste methode om het aantal bezoekers te vergroten, het bouwen van een nieuw onderkomen.*

CONTENTS

Samenvatting

In dit proefschrift wordt een concept gepresenteerd, dat het mogelijk maakt verschillende programma's voor het ontwerpen van geintegreerde schakelingen samen te voegen in één ontwerpomgeving.

Programma's voor het ontwerpen van geintegreerde schakelingen worden op verschillende plaatsen en door verschillende mensen ontwikkeld. Het resultaat is, dat de ontwikkelde ontwerp programma's alleen in de locale omgeving gebruikt kunnen worden, omdat iedereen verschillende data formaten en verschillende dataschema's gebruikt. Als iemand een ontwerp programma uit een andere omgeving wil gebruiken, dan is hij gedwongen of wel de programma code te wijzigen, of wel twee vertaalprogramma's te schrijven.

De beschikbaarheid van werkstations heeft de werkomgeving van VLSI ontwerpers danig veranderd. De meeste ontwerpers beschikken over hun eigen werkstation en hebben toegang tot speciale rekenmachines zoals array processors en super mainframes. Deze verandering heeft zijn invloed op de eisen die gesteld worden aan ontwerp-programma's, namelijk zij moeten in een gedistribueerde omgeving werken.

In het eerste gedeelte van dit proefschrift worden de VLSI ontwerpomgeving en de eisen van een VLSI ontwerpsysteem beschreven. De belangrijkste eisen van een ontwerpsysteem zijn de verplaatsbaarheid van het programma en de uitwisselbaarheid van ontwerp data.

In het volgende gedeelte wordt een dataschema gepresenteerd. Nadat verschillende data modellen zijn vergeleken, is een semantisch data model gekozen, om de structurele semantiek van een VLSI ontwerp database in uit te drukken. Het centrale gedeelte in dit dataschema is de 'basic design unit'. Deze unit is de grens waar ontwerp data management en laag niveau ontwerp data acces elkaar ontmoeten. Ook blijkt deze unit, de unit te zijn waarop een versie mechanisme, locking en recovery toegepast kunnen

worden. In het laatste gedeelte van dit hoofdstuk wordt een uitgebreider dataschema gepresenteerd, dat de gedistribueerde structuur beschrijft.

In het vierde hoofdstuk van dit proefschrift wordt een programma interface gepresenteerd. Deze programma interface, welke is gebaseerd op de invarianten van het dataschema, biedt de software ontwerper een eenvoudig beeld van de organisatie van zijn ontwerp data. Alle ontwerp management functies zijn voor hem verborgen. Data formaten zijn niet gedefineerd, maar als een programma in een andere omgeving moet kunnen werken, dan is een neutraal data formaat nodig. Programma communicatie is een belangrijk onderdeel in een ontwerpomgeving. Een mechanisme om data tussen verschillende programma processen uit te wisselen wordt beschreven.

Het laatste gedeelte van dit proefschrift bestaat uit drie appendices. De eerste appendix beschrijft een dataschema van een technologie database en de programma interface functies. De tweede en de derde appendix zijn copieen van artikelen. Appendix B is een artikel over een dataschema en een versie keten. Appendix C beschrijft een uitgebreider dataschema en een systeem architectuur van een gedistribueerd data management systeem voor het ontwerpen van VLSI schakelingen.

Summary

In this thesis a concept is presented, which will allow for the integration of different VLSI design tools in one design environment.

Many VLSI design tools are developed at different places by different people. As a result, the developed design tools can only be used in the local environment, because different data formats and different data schemas are employed. If someone wants to use design tools from a different origin, he is forced to either change the source code of the program or write two translator programs.

The availability of workstations has changed the work environment of VLSI designers to a great extent. Most designers have their own workstation and have access to special purpose computers, like array processors and super mainframes. This change has its impact on the requirements of the design tools. Design tools, design data, data management and designers all function in a distributed environment.

The first part of the thesis describes the VLSI design environment and the requirements of a VLSI design system. Most important requirements in an open VLSI environment are tool portability and design data exchange.

In the next chapter a data schema is presented. After the comparison of several data models, a semantic data model is chosen to represent the structural semantics of a VLSI design database. In this data schema the central part is the basic design unit. The basic design unit is the border where design data management and low-level design data access meet. Also, the basic design unit is the unit of access on which versioning, locking and recovery apply. In the last part of this chapter the data schema is extended, to structure the distribution of design units.

In the fourth chapter of this thesis a tool interface is presented. This tool interface, which is built on the invariants of the data schema, offers the software designer a clear and simple view of the organization of the design

data. All design data management functions are hidden. Data formats are not defined, but if a tool should be "plugged into" another environment, a neutral data format is necessary. Tool communication is an important aspect of a design environment. A mechanism to exchange data between different tool processes is described.

The last part of this thesis consists of three appendices. The first appendix presents a data schema for a technology database and the tool interface functions. The second and the third appendix are copies of papers. Appendix B is an article on a data schema and a version chain. Appendix C presents an extended data schema and system architecture of a distributed data management system for VLSI design.

1. THE CAD/IC DESIGN ENVIRONMENT

1.1 Design Data Management

Design data management became an important subject in the Computer-Aided Design of Integrated Circuits (CAD/IC) since the beginning of the 1970s. From this point in time the complexity of Integrated Circuits (IC) started to increase dramatically, as did the number of design tools and their data formats. Most of the CAD/IC programs were developed at different locations by different people, thus had their own data formats and their own user interfaces. If someone wanted to use a design tool from another source, he had to write translators to and from each program. This approach implied that for N programs, N^2-N translators had to be written. As a result, designers of integrated circuits had to use many translator programs to design, simulate and test one simple circuit. The number of translators can be reduced to a worst case $2N$ by choosing a common, neutral data format and translating to and from that format [Newton86]. Several data formats emerged during the 1970s and became public domain standards, the best known are CIF [Mead80] and GDS II for mask layout descriptions and SPICE [Nagel75] for circuit descriptions. ISPS [Bell71, Barbacci78] was commonly used as the language to describe the behavior of a circuit in abstract terms.

At the same time people started to use conventional databases for the management of design data. These record oriented database management systems (DBMS) [Held75], were used to store and retrieve all aspects of an IC design, such as rectangles, list of connections and transistors. Soon it became clear that the application of the conventional database management systems was not suited for the CAD/IC environment. A state-of-the-art IC can contain more than 150000 transistors. Over 1 Giga-byte of design data is required to store all aspects of this design. A design data management system (DDMS) has to store this vast amount of design data, controlling the access to the data and maintaining its consistency. The answer for a

suitable design data management system lies in exploiting the inherent hierarchical structure of the design data and the partitioning of the data in design data and in meta design data. The meta design data contains information about the design data. Other requirements of such a data management system include access methods to the design data, multiple versions and design alternatives, support for workstation and network based transactions and optimal performance with low cost hardware.

Textual interchange formats were developed to meet the need for design data transfer between different design systems. Several different formats were used to describe different aspects of an IC. In the 1980s one neutral data format is being developed, EDIF (Electronic Design Interchange Format), [EDIF87, Eurich86], to replace all other textual formats. The definition of this data format can be the basis of a neutral data format in a design data management system.

A design data management system (Figure 1.1) is the kernel of a design environment around which design tools can be integrated [Katz83, Brouwers87]. A proper DDMS operates first as the neutral repository of design data: the design database. The tools create and modify the design data, while the DDMS stores and maintains the design data, thereby guaranteeing consistency.

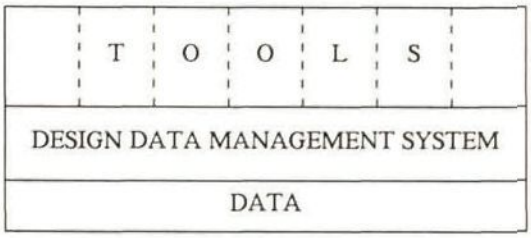


Figure 1.1. Tools integrated on top of a DDMS

Furthermore, a sophisticated DDMS has to supply additional services to provide a basis for the construction of an intelligent design system, which

relieves the designer from the burden of organizing his design data. Clearly, system integration is more than the definition of some common formats: Which copy is the latest version? Has this layout been extracted since it was updated and, if so, which circuit description was derived from it? If I change this layout, which other parts of the design will be affected? It is the ability to answer such questions that differentiates a true DDMS from a simple data repository [Newton86].

Powerful workstations and the availability of computer-aided design tools have improved the productivity of designers to a great extent. Medium size integrated circuits can be developed on a single workstation. However, the design of highly complex circuits still relies heavily on the processing power of mainframes and supercomputers. But also in this case workstations are invaluable to designers. They provide a fast and interactive work environment, where designers can enter a description of a circuit. Computer network capabilities provide the designer access to special hardware or computer power and offer the opportunity of data sharing among teams of designers.

The UNIX operating system has been chosen by many people as the software environment for their workstations, due to its availability, portability, flexibility and functionality. It is quickly becoming the standard operating system for engineering workstations [Hardwick85]. This provides the designers with a common environment for different hardware acquired from different vendors. This also helps software developers by allowing them to easily port their software to different systems without major changes due to the differences in different operating systems. Networking provides resources including hardware simulation accelerators, mass storage devices, plotting servers, printers and some CPU intensive programs available on mainframe computers. A standard networking protocol such as TCP/IP allows hardware from different vendors to communicate electronically.

The open system concept of the UNIX operating system should also be applied to the CAD/IC environment. It will allow the software developer to add new design tools to his system without major effort. This is a very important capability for a system that needs to be dynamically updated as design methods change. Supporting hierarchical designs, where any block may be used at multiple levels and with many repetitions, is essential for the CAD/IC design environment. Design automation tools have as their goal the simplification of the design process by performing many of the synthesis processes automatically and guaranteeing correctness (silicon compilation). This goal can only be reached, if a well designed DDMS is available.

Technical innovations have significantly increased the complexity of integrated circuit design. An ideal VLSI design environment will have the following characteristics:

— it contains a design database with a neutral data format.

— it has a database management system which structures the design data and controls access to the database.

— its design tools are integrated with the design database.

— it is an open system, new tools can be added without a major software effort.

— it supports the hierarchical design methodology.

— it enables the use of shared design data.

— it supports different design styles.

— it supports different design process technologies.

— it has local storage, which in case of a network breakdown enables the user to continue with his design.

— it has a backup mechanism, which ensures the user that after a workstation breakdown design data can be restored.

A system that integrates design tools and a design database, and provides a human interface is clearly essential, if one wants to manage the design process. Also the design system should offer the possibility to extend its capabilities using artificial intelligence techniques. This could be achieved if the data controlled by the data management system is accessible for artificial intelligence programs.

A design environment consists of three basic components: a database containing design and management data, a database interface layer to permit orderly and secure access to the database, and a set of integrated design tools, which feed and are fed by the database via the database interface. This fully developed design system should ease chip development by preserving design data as it is produced. Versions of a design are automatically stored, and designers have only controlled access to change design data.

The availability of some software standards such as the UNIX operation system and the TCP/IP networking protocol provide for an uniform workstation environment. The need for high performance computers is still present.

1.2 Reading on VLSI Design and Data Management

This thesis describes the problem of constructing an effective design data management system. For those readers who are interested in acquiring information on VLSI design itself and its problems, we give a short list of available textbooks, which will serve as a background to the presented work.

One of the best known books about integrated system architecture and design is [Mead80]. An introduction in the integrated circuit design using CMOS, can be found in [Weste85, Glasser85, Mukherjee86]. There are several books about the design of digital systems. [Davio83, Mano82, Breuer77, Mano84] describe this area with considerable depth. The latest developments in the VLSI design area can each year be read in the VLSI

book series [Bryant83, Anceau83]. These describe particular subjects of VLSI design, which are in the focus of interest of the VLSI designers. An insight in the workstation environment, distributed file systems and networking, can be found in [Sun86, Apollo81]. The books deal with implementations of these techniques in a specific workstation environment. [Tsichritzis82] provides the reader with a thorough background on relevant issues in the field of database management.

References

Anceau83. Anceau, F. and Aas, E.J., *VLSI 83, VLSI Design of Digital Systems*, North-Holland (1983).

Apollo81. Apollo, Computer Inc. and Bellerica, N., *Apollo Domain Architecture*. Feb. 1981.

Barbacci78. Barbacci, M.R., "An Introduction to ISPS," *Technical Report, Department of Computer Science, Carnegie-Mellon University*, (1978).

Bell71. Bell, C.G. and Newell, A., "Computer Structures: readings and Examples," *Mc-Graw Hill Book Compagny, New York*, (1971).

Breuer77. Breuer, M.A., "Digital System Design Automation: Languages, Simulation and Data Base," *London Pitman*, (1977).

Brouwers87. Brouwers, J. and Gray, M., "Integrating the Electronic Design Process," *VLSI Systems Design*, pp. 38-47 (June 1987).

Bryant83. Bryant, R., *Third Caltech Conference on VLSI*, Computer Science Press (1983).

Davio83. Davio, M., Deschamps, J.P., and Thayse, A., *Digital Systems with Algorithm Implementation*, John Wiley & Sons (1983).

EDIF87. EDIF,, "Electronic Design Interchange Format, Version 2 0 0, Reference Manual," *EDIF Steering Committee*, Electronic Industries Association, (1987).

Eurich86. Eurich, J.P., "A Tutorial Introduction to the Electronic Design Interchange Format," *Proc. 23rd IEEE Design Automation Conference*, pp. 327-333 (1986).

Glasser85. Glasser, L.A. and Dobberpuhl, D.W., "The Design and Analysis of VLSI Circuits," *Reading MA Addison-Wesley*, (1985).

Hardwick85. Hardwick, M. and Yakoob, N., "Using a Database System and UNIX to Author CAD Applications," *Proc. IEEE ICCAD - 85*, pp. 53-55 (1985).

Held75. Held, G.D., Stonebraker, M., and Wong, E., "INGRES - A Relational Database Management System," *Proc. 1975 Nat. Computer Conference*, AFIPS Press, (1975).

Katz83. Katz, R.H., "Managing the Chip Design Database," *IEEE Computer Magazine* **16**(12) pp. 26-35 (Dec 1983).

Mano82. Mano, M.M., *Computer System Architecture Second Edition*, Prentice-Hall (1982).

Mano84. Mano, M.M., *Digital Design*, Prentice-Hall (1984).

Mead80. Mead, C. and Conway, L., *Introduction to VLSI Systems*, Addison Wesley, Reading MA (1980).

Mukherjee86. Mukherjee, A., "Introduction to Nmos and Cmos VLSI Systems Design," *Prentice Hall*, (1986).

Nagel75. Nagel, L.W., "SPICE2: A Computer Program to Simulate Semiconductor Circuits," *University of California, Berkeley*, (May, 1975).

Newton86. Newton, A.R. and Sangiovanni-Vincentelli, A.L., "Computer-Aided Design for VLSI Circuits," *IEEE Computer Magazine*, pp. 38-60 (April 1986).

Sun86. Sun, Microsystems Inc., *Networking on the Sun Workstation*. Feb. 1986.

Tsichritzis82. Tsichritzis, D.C. and Lochovsky, F.H., *Data Models*, Prentice-Hall, Englewood Cliffs, NJ (1982).

Weste85. Weste, N. and Eshraghian, K., *Principles of CMOS VLSI Design a System Perspective*, Addison-Wesley (1985).

2. FUNCTIONAL REQUIREMENTS OF A DESIGN SYSTEM

2.1 Introduction

Today, one of the main problems in designing a CAD/IC design environment is the management of the complexity of the design data. Further improvements in the process technology promise to increase the already large complexity of a design by another order of magnitude.

At this moment it is practically impossible to design an integrated circuit without the assistance of computers and special computer programs. Many different design tools for computer aided design of integrated circuits have been developed and many more will be developed to meet the demand of the increasing complexity of integrated circuit design.

To our knowledge there does not exist a design system at this moment in which all tools covering functional design to pattern and test generation are integrated around a common database. Such a tool set can only be realized, if design tools from different origins can be integrated. This is not an easy task, because design tools use different data schemas, different data formats, and different graphical interfaces. If software designers would utilize well accepted standards by which their tools can be integrated, then the productivity of VLSI design could be improved substantially.

Integrated circuits are no longer designed by one individual designer. Complex designs are subdivided into smaller units and each unit is designed by one or more designers. The design of an integrated circuit is divided in several subtasks. This decomposition creates a need for concurrent data sharing, because the designers must use or modify each others design data. In this situation, there is a need for a design system that provides database management functions, such as concurrency control, a version mechanism and the storage of design data relationships.

In this work we describe the approaches required of a Design Data Management System (DDMS) to provide a framework for tool integration based on data sharing. In this chapter we will formulate functional requirements for design data management.

2.2 The Goals of a DDMS

The goals of a DDMS are:

1. Enhancing the portability of tools.

   The DDMS can provide functions and specifications that when put into practice, will improve tool portability without greatly restricting the freedom of the tool developer.

2. Facilitate the exchange of design data.

   Design data is exchanged among different sites, among different tools and among different organizations. The DDMS should contain facilities that can be used to translate, store and retrieve a neutral data format.

3. Assuring a uniform design environment.

   The design environment is implied by the operating environment that the tools create for its users. A frequent reason for lack of uniformity is the absence of adequate documentation for user and system interface functions, that are portable among host environments.

4. Provide a framework for supporting design management.

   Large integrated circuits typically are designed by design teams. This requires, for example, controlled sharing of design data, protection of released design data, and monitoring of design methods and progress.

5. Provide a framework for reuse of previous designs.

   With the rapid accumulation of design data, the reuse of past designs

is becoming increasingly dependent upon support from sophisticated browse tools.

Each of these requirements will be discussed to some detail in the remaining of this chapter. Goal one will be discussed in chapter 2.3 and goal two will be discussed in chapter 2.4. The goals three, four and five form the subject of chapter 2.5.

2.3 *Tool Portability*

Tool portability involves a number of capabilities,

— Access functions.

— User interface.

— Tool interface.

2.3.1 Access Functions

A data model will provide the basis for establishing concise semantic descriptions of the design data. If the invariants of the data schema are standardized within the system, a standard tool interface can be developed, which will provide the software designer with standard access functions and will also offer some degree of freedom.

2.3.2 User Interface

The DDMS has to provide a user interface specification that will result in a uniform design environment. This user interface specification consists of two parts. The first part is the specification of a set of functions, which will allow for the development of design tools with an uniform graphical software interface. The second part of the user interface specification is a description of how an user can interact with a design tool. This description specifies among others the construction of user-menu's, pop-up menu's and dialogue windows. In this way the style of interactions between users and design tools is defined. At this moment a well defined graphical interface is the X-window system [Gettys86]. This standard consists of simple graphical functions, but also has standardized menu handling. This will

allow for the specification of a DDMS with an uniform user-interface style.

2.3.3 Tool Interface

If a tool interface is standardized, software developers can create programs in a uniform fashion. The tool interface standard will provide functions, which give the software developer the possibility to access design data and obtain information about design data, without the necessity to have a detailed understanding of the implementation of the DDMS. Tool portability and tool compatibility are greatly enhanced using a standard tool interface, which are the goals of tool integration.

The following requirements can be formulated for a tool interface:

- The tool interface must bring about efficient interaction between the tools and the DDMS.
- The tool interface must be independent of specific tool features or design methodologies. It should be universal, to result in an *open-ended* design system where the DDMS acts as a free-for-all public repository that can communicate with any type of tool and environment.
- The tool interface must be independent of specific features of a DDMS. For example, it must allow interfacing to DDMS's with or without version control, concurrency control, multiple view-types, etc. When this requirement is met, the tools can actually be "plugged in" in the same way in any DDMS, whether it concerns different releases of a DDMS at a certain site or DDMS's at different sites.

In summary, a tool interface should offer some degrees of freedom, but at the same time the necessary discipline to facilitate software evolution and exchanges.

2.4 Data Exchange

If design tools are integrated in a DDMS, this will impose stringent data exchange requirements to ensure communication among data repositories on different hosts. The DDMS must provide a small set of standard formats

for data exchange, preferably a single one [EDIF87, Eurich86, Newton87]. Tools translate into and out of a neutral data format using one pair of translators, no longer requiring the creation and maintenance of a possibly large number of translators. In addition, it will be necessary to describe the data requirements (semantics) of each tool precisely, so that inconsistencies in how data is interpreted and represented can be recognized and dealt with.

2.5 *Design Systems*

If the capabilities of design automation are to be fully exploited, not only design tools have to be integrated in a design system, but also administrative and management tools. If the data representation employed by the tools are standardized, these would give the vendors of design tools the opportunity to fully support these limited data formats. In addition the tools should have common user interface. This will have the effect that designers do not have to master different man-machine interfaces.

2.5.1 Management and Control

The management and control facilities of a DDMS are responsible for supporting two main tasks:

— controlling and monitoring the design process.

— enforcing access controls at the software level.

The DDMS must be able to store dependencies that exist among different design objects (e.g. when a schematic is converted by a translator into a layout description for use by layout tools). If the schematic is changed and a new version is created, the system must be able to relate the new version to the preceding one. Also it must be able to store information about the design object, e.g. a design history, its validation status and which designers have changed the object.

2.5.2 Data Management

It is important that a DDMS permits designers to locate, obtain, and correlate diverse design data quickly (browsing). A key function of a DDMS is that the data management system can deal equally well with

design, management, and control data. The data can be distributed among any number of different workstations. The software designer of a DDMS has to present a clear conceptual model to the user of the DDMS. This model will describe how data is structured, its status and will hide to the designer all implementation details, like physical distribution.

2.5.3 Administration

The administrators of a DDMS must have the ability to store, manage and access the data schema, designer information, tool descriptions, technology rules and descriptions of the workstation environment. To meet this goal the DDMS must be a highly description-driven system. For each type of information a description should be available, which describes the resource requirements, the dependencies, input and output requirements and other. For example, there should be a description of the data schema, describing the data structure and its constraints. The DDMS reads this description and arranges internally its data structure accordingly. If the administrator wants to add some data type, he only has to edit the description file and restart the DDMS.

2.5.4 The Environment

The environment in which the DDMS operates will most likely be a distributed one. Workstations from different vendors will be connected to each other, through one or more computer networks. This operating environment of a DDMS requires the use of software standards, for its implementation. This would allow for the use of a DDMS in several different hardware environments. Already there are several accepted standards for graphical interfaces, operating systems and computer networks. There are no accepted standards for database access functions.

Model of the Design Process

The DDMS must be able to support any model for a design process. It must be independent of any specific design philosophy. For instance it must be able to support, among others a traditional design philosophy as depicted in Figure 2.1.

Figure 2.1 depicts a possible decomposition of the traditional design process; three basic activities are identified. One further level of decomposition for these activities is depicted in Figure 2.2. These activities obviously overlap. Also there are feedback loops.

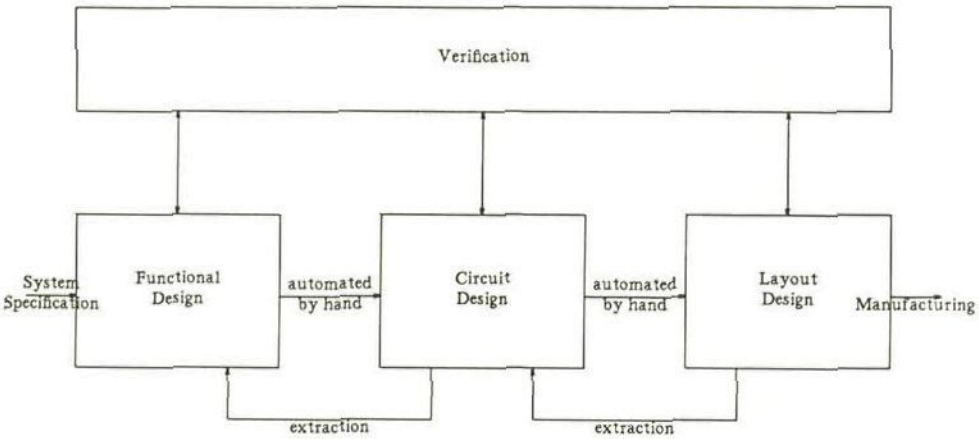


Figure 2.1. An example of the design process

Design Analysis
Design Evaluation
Design Refinement
Design Simulation
Design Test Generation
Design Verification

Figure 2.2. Functions of a design activity in some detail

It must also be able to support, for instance, a Gajski type design model [Gajski85, Gajski86] (Figure 2.3), or even a silicon compiler with its various tasks.

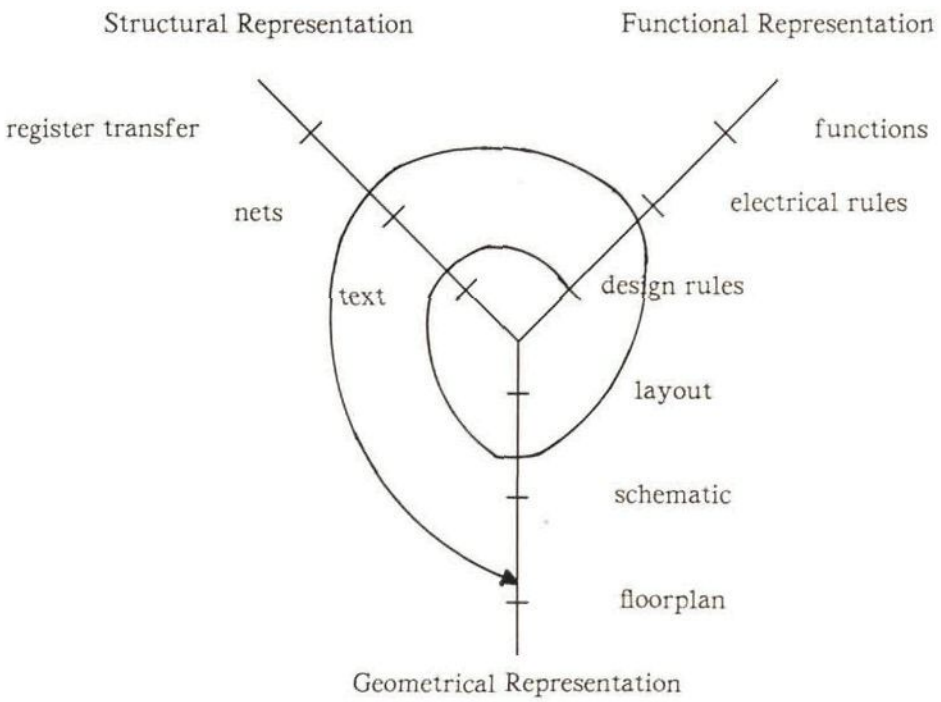


Figure 2.3. Gajski Ychart

Several different tasks must be performed before the design is ready for fabrication.

1. the functional description is translated into a structural description.
2. the layout of each structural component is instantiated.
3. all structural components are placed on silicon and routed.

Relationships

There are many objects in a design database that have relationships to other objects. One of the most important functions of a DDMS is to store these relationships. This would allow for the use of special management tools

which can maintain the consistency of a design.

Design Teams
In a multi user environment, many designers will try to access design data at the same time. The DDMS must provide services which control the access to the design data. Other issues include data sharing, version control and configuration control.

Managing the Design Process
The following is a list of general requirements for data management and control functions to be supported by a DDMS.

1. Designers should be protected from unvalidated design changes.
2. Access to design objects must be limited.
3. There must be a mechanism which enforces consistency constraints over design objects.
4. In a database supporting multiple versions of objects, policies must be provided as to which version is automatically selected at object checkout time.
5. Hierarchical and multilevel representations of arbitrary complexity must be supported.

More specifically:

Version Management,
there are two important problems that a DDMS must address. The first is that it should be impossible for a designer to use a design object of another designer, without the consent of that designer or the design manager. This would e.g. protect a designer from using a design object, which is currently being changed. Secondly, the DDMS should have some automatic selection of the right design objects to be used. This would mean that a designer uses design objects with known properties.

Control of Modifications,
a designer has the task of determining whether a design object is consistent, that is that it meets its specifications and that it satisfies implementation constraints. The DDMS can only provide services to help the designer with this task, it can record which design tool has been used on a particular object. The designer however, is the one who has to interpret this information and take the right decisions.

Reuse of Designs,
if information about previous designs is made available to designers, significant savings can be achieved. The DDMS data management tools are responsible for this task. There are two types of tools which can provide designers with information about previous designs and designs in progress. They are a browser and a prober.

Basically a browser is a tool that allows the designer to browse through designs. The designer has to decide which design object in the database is relevant to him. A probing tool has the capability to search through the database, after the designer has partially specified his requirements. The probing tool will list all design objects that match these requirements. Probing is more difficult than browsing, because probing requires pattern matching capabilities and possibly knowledge of what is stored in the database.

2.6 *Design Methodology*

A methodology is a combination of rules that describe default, preferred, and mandatory steps in a design process. A design system has the requirement to support hierarchical designs, to support the division of design steps into smaller steps. Also, the design system should support a mechanism to control the sequence of steps taken, design check point validation and approval for the release of a design.

2.7 Requirements Overview

The main functions of the DDMS are to provide an integrated design environment for using tools and managing design data and to support the management of the design process and data exchange between organizations. It must support these functions in a way that ensures that the DDMS is:

— adaptable.

— distributed.

— portable.

— extensible.

— evolutionary.

The requirements are presented in two groups. The first group defines the functional requirements:

— Tool integration.

— Data Exchange.

— Management and Control.

— Data Management.

— System Interface.

The second group defines the required approaches that address the functional requirements:

— Data model and data schema.

— Tool interfaces.

— Distributed data management facilities.

— User interface.

The main topic of this thesis is to construct a systematic methodology for a DDMS, which will satisfy the functional requirements.

In the next chapter we shall discuss several data models. One of these data models, the semantic data model, seems to be very suitable to describe the CAD/IC environment. Two data schemas are presented. The first one expresses the structure and properties of a particular design environment, the second data schema discloses the extra structure and properties to represent a distributed design environment. In chapter 4 a tool interface and a user interface are discussed.

References

EDIF87. EDIF,, "Electronic Design Interchange Format, Version 2 0 0, Reference Manual," *EDIF Steering Committee*, Electronic Industries Association, (1987).

Eurich86. Eurich, J.P., "A Tutorial Introduction to the Electronic Design Interchange Format," *Proc. 23rd IEEE Design Automation Conference*, pp. 327-333 (1986).

Gajski85. Gajski, D.D., "ARSENIC Silicon Compiler," *Proc. ISCAS '85*, pp. 399-402 (1985).

Gajski86. Gajski, D.D., Dutt, N.D., and Pangrle, B.M., "Silicon Compilation (tutorial)," *IEEE 1986 Custom Integrated Circuits Conference*, pp. 102-110 (1986).

Gettys86. Gettys, J., Newman, R., and Fera, T. Della, "Xlib - C Language X Interface, Protocol Version 10," *MIT, Cambridge, Mass.*, (1986).

Linn86. Linn, J.L. and Winner, R.I., *Engineering Information Systems*, The Institute for Defense Analyses, Alexandria, Virginia (1986).

Navathe86. Navathe, S., Elmasri, R., and Larson, J., "Integrating User Views in Database Design," *IEEE Computer Magazine* **19**(1) pp. 50-62 (Jan 1986).

Newton87. Newton, A.R., "Electronic Design Interchange Format, Introduction to (EDIFVersion 2 0 0)," *Proc. IEEE CICC '87*, pp. 531-535 (1987).

Roussopoulos84. Roussopoulos, N. and Yeh, R.T., "An Adaptable Methodology for Database Design," *IEEE Computer Magazine*, pp. 64-80 (May 1984).

3. CONCEPTUAL MODEL OF THE DDMS

3.1 Data Models

3.1.1 Introduction

A *data model* is a collection of concepts and constructs for expressing the static properties, dynamic properties, and integrity constraints of an application environment [Lyngbaek84, Afsarmanesh84, Bekke83]. It is characterized by:

— A collection of constructs: the data definition language.

— A collection of fundamental operations: the data manipulation language.

— A set of integrity constraints defined on its constructs.

Given a data model, a *data schema* is defined to describe the structure and properties of a specific application environment. A data model can be seen as a generic mechanism from which data schemas can be instantiated.

Finally, a *database* is a data repository containing a possibly large amount of interrelated data, structured according to a corresponding data schema. Hence, a data schema can be seen as a generic description out of which the contents of a database can be instantiated.

Over the years several data models have been developed. Historically, the following four classes of data models can be recognized:

— hierarchical

— network

— relational

— semantic

The hierarchical, network, and relational data models are frequently referred to as the classical data models [Lyngbaek84, Bic86]. In the next paragraph an overview is presented of these classical data models [Hardwick87].

3.1.2 Classical Data Models

3.1.2.1 Hierarchical Data Model

The hierarchical data model consists of nodes organized in a tree [Tsichritzis76, Bekke83, Bic86]. The nodes in the hierarchical data structure correspond to records in tables of data. Between the nodes of a tree a one-to-many relationship exists. Because the data structure must form a tree, the direction of the arcs is always towards the leaves of the tree. The existence of a root-node is obligatory. A hierarchical database is a set of ordered trees; the placement of nodes in a tree is significant. Thus a node can only be seen in the context of its hierarchy. The advantage of the hierarchical data model is that it allows for fast retrieval operations and easy contextual naming. The main drawback are its limited structuring capabilities, e.g. it does not allow to represent many-to-many relations directly, and it provides only primitive operations.

3.1.2.2 Network Data Model

The network data model is based on nodes and arcs (graphs) [Bachman69]. It is an extension of the hierarchical model, a node can have several superior as well as several subordinate nodes. An owner record type can have one-to-many relationships with other member record types, called a set type. The presence of the owner record in a set is essential. As is shown in [Bekke83] not all record types in a network data schema correspond to the complete definition of a particular concept. Thus, to retrieve the information of a certain node, several records of different types might have to be attended by a one record at a time process called *navigation*. This makes that the algorithms are often complex, while the user must be aware of the internal organization.

3.1.2.3 Relational Data Model

The relational data model is a more user oriented data model [Codd70]. It is based on the mathematical concept of a relation, i.e. a subset of a cartesian product. A relation is a set of *n-tuples* (records), and is typically represented by a table. A column is called an *attribute*, and the set of values from which the attribute values can be drawn is called the *domain*. Each relation has a *primary key:* one attribute or a combination, the values of which distinguish the (unique) tuples from each other. The relations do not contain implicit references (pointers). Associations between tuples are exclusively represented by attribute values drawn from a common domain.

The main attraction of the relational model is its mathematical clarity [Bic86], which facilitates the formulation of nonprocedural, high-level queries and thus separates the user from the internal organization of the data. The separation of the logical organization and the internal organization, results in data independence for the user. The relational model has some serious drawbacks. First it is a flat model; the relations are not positioned with respect to each other. The use of composed keys does not provide the user with sufficient means to represent all abstractions in a precise way. The integrity constraints have to be defined explicitly; this is not an integral part of the modeling process. In [Bekke83, Bekke85] several examples are given that clearly show the various defects of the relational model.

3.1.3 Semantic Data Models

The classical data models are all *record based.* When modeling an application environment, not all record types in the resulting schema correspond to the complete definition of a particular concept from that environment. That is, they lack semantic expressiveness [Bic86, Afsarmanesh84, Hardwick87].

The *semantic data models* enable the user to better formalize the semantics of his data, and are therefore considered more *user oriented.* Instead of being based on the record model, the semantic data models are *object based;*

the application environment is modeled as a collection of interrelated objects, each one corresponding to a concept from this environment.

Attempts to categorize the semantic data models are described in [Lyngbaek84] and [Afsarmanesh84]. Several semantic data models have been investigated.

3.1.3.1 The OTO-D Data Model*

In the semantic approach the notion of abstraction, i.e. representing the relevant details while suppressing the irrelevant ones, plays a dominant role. When representing the invariants of a dynamic environment, not the elements themselves but their properties are important. The semantic approach is based on objects and the notion of *type*. Each object has a type, which is defined by a certain number of different properties.

In this chapter we shall use examples, which have no relations with the CAD/IC environment to prepare for the data schema that will be defined in chapter 3.4. It demonstrates the general application of the discussed data models. For example the abstraction:

TYPE student = name, address, department

defines an object, type student, characterized by the properties name, address and department. These properties are called *attributes*. An object having the properties of a certain type is called an *instance* of that type. A data schema consists of a number of these type definitions.

3.1.3.1.1 Convertibility and Relatability

In the semantic data model we can distinguish two types of invariant properties [Bekke83]. *Convertibility* is the property that each type has only

* OTO-D has been developed at the Mathematics & Computer Science department of the Delft University of Technology by ir. J.H. ter Bekke and his group.
"OTO-D" stands for Object Type Oriented Data model.

one predicate and that each predicate belongs to one type. Because the type is completely characterized by the predicate (attributes), while on the other hand each predicate describes one type, there is a one-to-one correspondence between the type and the predicate of an assertion. Convertibility also has consequences at the instance level. Each object is uniquely characterized by its attribute values; the instance identification is of no importance. Based on the notion of convertibility the type definitions can be checked for completeness during the construction of the conceptual model.

The second invariant property, *relatability*, is the property that, if the name of an attribute in a predicate equals the name of a type definition, the attribute relates to that type definition. For example in:

TYPE student = name, address, department
TYPE department = name, head

the attribute department in the definition of student is related to the definition of department. Relatability also has consequences at the instance level. It implies that an attribute value is related to an instance of the type of which the attribute is a property. As a consequence the set {student ITS department} is at any time a subset of {department}: *subset invariance*.

When modeling an application environment it is often simple to recognize a number of types and give some preliminary type definitions. However, to remove imperfections they have to be checked individually (for completeness) and in connection (for consistency). In this process the semantic concepts of convertibility and relatability can be applied.

3.1.3.1.2 Aggregation and Generalization

OTO-D offers two abstraction primitives to construct a data schema: aggregation and generalization. *Aggregation* is a form of abstraction in which a certain number of different properties is combined to create a new named object. Examples of aggregations were student and department. OTO-D offers a clear diagrammatic notation to visualize the relationships among the types of a data schema. The example looks like:

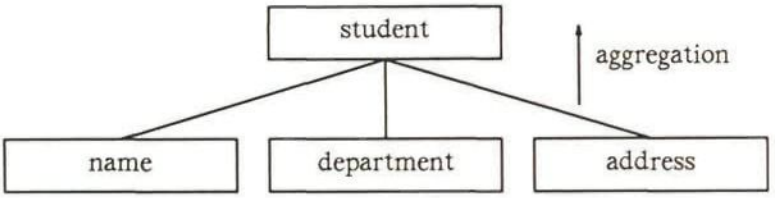


Figure 3.1. Example of an aggregation

Generalization (Figure 3.2) is a form of abstraction that relates a type to a more generic one. In knowledge representation research this is known as the IS-A relationship.

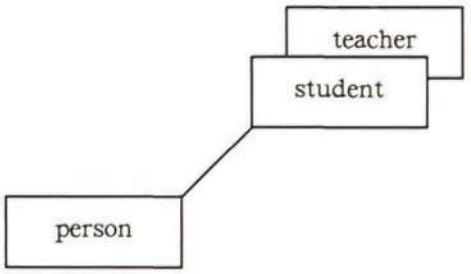


Figure 3.2. Example of an generalization

3.1.3.1.3 Data Manipulation Language

The data manipulation language of OTO-D offers *selection*, *extension* and *modification* commands. The most important expression is the selection, the general form of which is:

GET <type name>
ITS <attributes> *property list*
WHERE <condition> *qualifying predicate*

As a consequence we can only "look downward" along the schema, starting from a composed type to its attributes, its attributes its attributes, etc. Given an arbitrary schema, the semantic concepts of OTO-D guarantee that all data that can be addressed this way is present (referential integrity) and

related in a meaningful way according to the schema. An example of a selection command on the data schema presented above:

GET student
ITS name, address
WHERE department ITS name == 'EE'.

3.1.3.2 DODM

DODM is a simple object-oriented model for multiple databases [Lyngbaek84]. It provides a small set of primitive operations that allow users to define, manipulate and retrieve objects. DODM also supports object sharing, access control and inter database relationships. In DODM databases are modeled as a collection of objects and relationships, no distinction is made between meta design data and design data (see chapter 3.4). All kinds of data are stored as objects in the database and the objects are interrelated by user defined relationships. The model provides no mechanism for object classification.

Suppose, for example that you want to express in DODM, that the primitive objects 'name' and 'head' characterize a 'department', and the objects 'name', 'address' and 'department' characterize a 'student'.

In DODM a database will consist of a collection of tuples of the type (x, y, z). Let x, y and z be objects. Then the tuple (x, y, z) represents the assertion that a relation y exists between x and z. Let FIND ("?", y, z) be the basic query which returns all objects, such that (x, y, z) belong to the database.

Now consider the FIND operation:

FIND (FIND(?, "has dept", FIND(?, "has name", "EE")), "has name", ?)

The type information has no significance in this operation. The object 'department' and the object 'student' both have the attribute "has name". You can not specify that in FIND (?, "has name", "EE"), only the objects 'department' are to be considered and not the objects 'student'. There is no limited scope for attribute names. An equivalent OTO-D retrieve operation

looks like:

```
GET student
    ITS name
    WHERE department ITS name = "EE"
```

In this case the object classification TYPE has significance in the search operation, i.e. the string "EE" is only used in the 'department' object context.

The main purpose of DODM is not to introduce a high level semantic data model, but rather to provide a basic frame work for object oriented modeling. DODM supports broadcast communication and point-to-point communication. Three functions allow the user to send/receive messages to/from a DODM database in a network. For these reasons, DODM could be used as an implementation layer, on which the OTO-D model is built. ODM provides the basic object functions, while the 'D' (Distributed) in DODM will allow for the distribution of objects in a network.

3.1.3.3 DAPLEX

DAPLEX [Shipman81] is a data definition and manipulation language for database systems, grounded in a concept of data representation called the functional data model. The basic constructs of DAPLEX are the object and the function. These are intended to model conceptual objects and their properties. In general a DAPLEX function maps a given object into a set of target objects.

Three striking types of functions in DAPLEX are:

— Derived functions.

— Multi argument functions.

— Multi valued functions.

Derived functions allow users to represent arbitrary object relationships directly by defining them in terms of existing relationships. There is a connection between the direction of the functional dependencies of objects and the queries to be executed. In the data schema the retrieval path

possibilities have to be recorded. Each function definition adds such a retrieval path. The derived functions give the user the capability to add extra retrieval paths. This mechanism allows for faster retrieval of data at the cost of redundancy. It is common practice to use pattern matching, in resolving a query in a semantic data model. Besides the pattern matching facilities, DAPLEX has the derived function mechanism in order to execute particular queries.

In the semantic data model, one of the basic principles is that a relationship should be accessible from both directions. The derived function mechanism poses a question concerning this basic semantic principle, because of the one to many relationships of a function and the one to one relationship of a derived function.

DAPLEX offers the designer of a data schema the use of multi argument functions. This construct has the advantage that the introduction of new objects can be reduced (Figure 3.3).

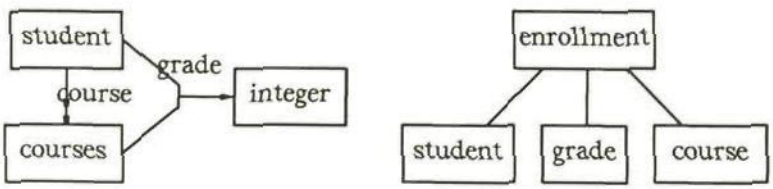


Figure 3.3. Reduction of objects using the multi argument functions

In this example an extra object, the enrollment, was introduced in the right data schema. The problem of the functional data model, with its multi argument functions and multi valued functions is the conflict that arises with the requirements of semantic convertibility. Convertibility is considered as an invariant of all definitions within a conceptual model, i.e. a time invariant attribute which is never changed by any database operation.

The DAPLEX data model does not clearly define the structure of an object, because it makes no distinction between connections that link objects to attributes and connections that link objects to objects, i.e. it has a lack of expressing structural semantics. The DAPLEX data model contains some ambiguities, which makes the use of this data model disputable.

3.1.3.4 Conclusion

We have described several data models. A database for the CAD/IC environment can be modeled using any of these data models [McLeod80, Hardwick87]. In our application the design database is a cluster of networks. The network data model is suited to represent this cluster of networks. However, we can make the following remarks about this data model:

1. The algorithms to store and retrieve information are complex and not general (chapter 3.1.2.2).
2. The network data model can be placed in the category of semantic data models. The semantic data model is more general and will give a clear overall picture.

The relational data model and the hierarchical data model are both not suited in the CAD/IC environment. Structuring design data is one of the important requirements of a database management system. The hierarchical data model has limited structuring capabilities, and the relational data model is flat.

In a semantic data model the definition of the integrity constraints [Smith77] is an integral part of the modeling process. A semantic data model offers a clear diagrammatic notation to visualize the relationships among the composed types of a conceptual model. The data manipulation language is tuned to the concepts that are of importance during data definition.

Starting from the semantic concepts of convertibility and relatability OTO-D provides us with the means to formally judge a conceptual model of an

application environment, constructed using the abstraction primitives aggregation and generalization. Furthermore, it can be shown that the conceptual model of the data dictionary, i.e. a database containing data about the data (meta design data), can be expressed in terms of OTO-D itself. Therefore, we believe that OTO-D comprises a methodology for data modeling, suited to formalize the semantics of the design data. It is the data model that shall be used further is this work.

3.2 *The Basic Design Object*

At first sight, classical database management systems (DBMS) offer some attractive facilities for the reliable storage of design data, including recovery mechanisms and concurrency control. However, most of these DBMSs have been targeted for business applications and do not specifically address the problems encountered in a design environment [Sidle80]. Transactions on a business DBMS typically are short in duration and affect only a small amount of data. In the design environment on the other hand, the designer requests all the information pertaining to a piece of design to modify it extensively over a long period of time before returning it to the database [Buchmann84].

The important issue is that VLSI design applications invariably deal with conceptually *localized collections of related data* which are manipulated as a single entity. This localization needs to be conserved by the design database. In line with several other researchers [Katz83, Batory85] we call these basic objects *design objects*. The design object should play a dominant role in the organization of the design data within the design database. The arguments for this approach are listed below:

— The design object is the *unit of access*. Design objects are extracted and replaced as a unit. Hence, such issues as *concurrency, recovery* and *versioning* should be handled at the level of the design object.

— To support *concurrent* access a mechanism is needed that locks design objects as atomic units.

- *Recovery* issues are also handled at the level of the design object. The design database will undo the effects of an incomplete design transaction by returning to the last saved copy of the design object (additional recovery facilities provided by the tools left out of consideration).
- The design object is the *unit of version propagation.* Because the design object is the unit of retrieval and storage, versioning of design data should be handled at this level.
- Design objects can be seen as the nodes of a hierarchical multi-view 'matrix' [Dewilde86].
- By taking the design object as the basic entity for further modeling, we hope to construct a coherent DDMS framework, without getting involved with representation details of some predetermined types.
- To the designer the design object has a well defined meaning: the behavioral description of his ALU, the circuit description of a flip-flop or his new routing result.

3.3 *The Initial Data Schema*

To define the object type *design-object*, we have to examine by which other object types design-object is characterized. First, a design object has a *name* by which it can be identified. Further, in a logically distributed environment each design object has been constructed in connection with a certain *project*. Other attributes of design-object might be its *designer* or the *date* of construction:

TYPE design-object = name, project, designer, date

In a project oriented environment there is no need for all names to be globally unique. Therefore, the scope of a design-object ITS name is limited to the design-object ITS project. The resulting diagram is given below.

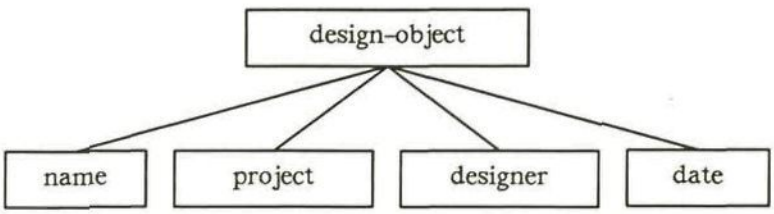


Figure 3.4. Diagram of the definition of the type design-object

3.4 *Design Management Data Schema*

At a first glance the DDMS is much like a librarian, providing controlled access to its basic objects, the cells, while administering relevant information about these objects: the meta design data. The meta design data describes how the basic objects are related to each other. It contains all hierarchical information of a design, as well as equivalence information relating basic objects of possibly different view types. The administration of the evolutionary development of a design, which distinguishes a DDMS from a classical library, is supported by dedicated version and state control mechanisms.

Our approach employs a semantic data modeling technique (OTO-D) as a formalized tool for the analysis of the semantics of the meta design data [Leuken85, Wolf86, Wolf88]. In this way a conceptual data schema, reflecting the different object types and their relationships, is derived. When the dependencies encountered in the design data are made explicit, they can be maintained and made available to both management tools and sophisticated design tools. The semantic data model employs a declarative data manipulation language, which provides a simple and clear access mechanism to the meta design data.

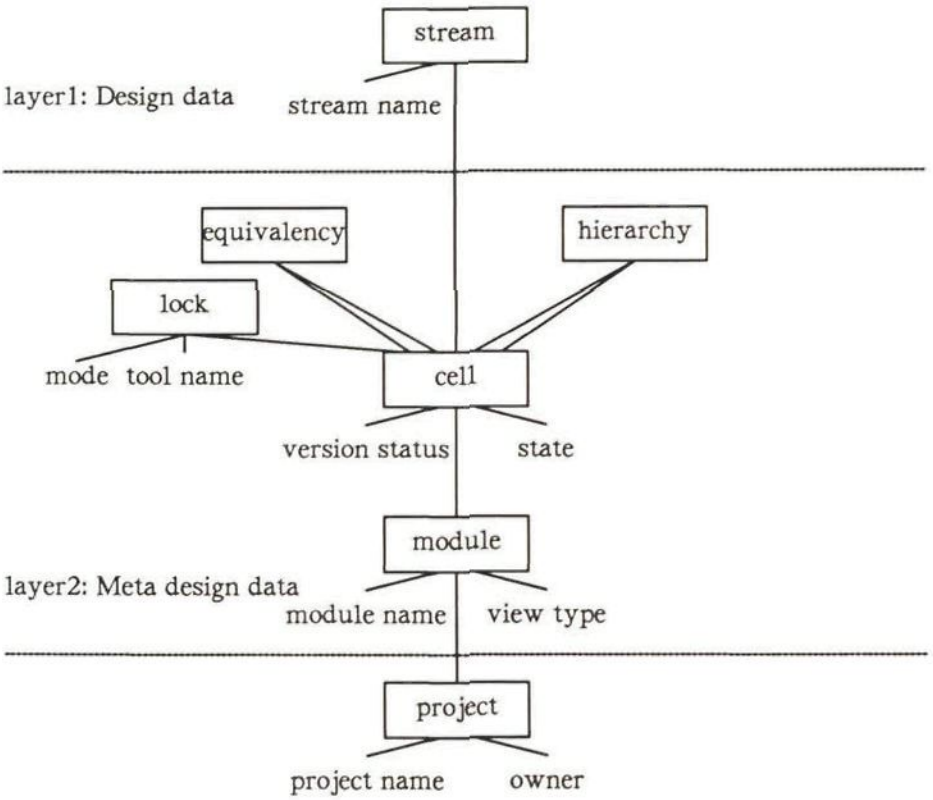


Figure 3.5. Conceptual data schema

Figure 3.5 depicts the semantic data schema [Leuken85, Wolf86, Wolf88], which we take as representative for a design environment. The data schema contains hierarchies, equivalencies, versions, locks and views. The basic objects in the data schema are the design objects or cells. The data management system maintains information about the design objects: the meta design data. The meta design data describes how the design objects are related to each other. It contains all hierarchical information of a design

description in a particular view, as well as equivalence information relating design objects in possibly different views. The version information of an object, locking information, ownership etc. are also considered meta design data, being maintained by the data management system.

The functional capabilities of the DDMS are partitioned into three layers. The first layer contains low level I/O functions. Its objective is to provide efficient access to the design data. It also offers data independence, which allows for the usage of data compression techniques. The second layer provides controlled access to the design data, while maintaining information about it. The object types in this layer will be discussed in more detail in the next chapters. The logical distribution of design data across different local databases requires a dedicated communication mechanism, while consistency among these databases should be preserved. These issues are resolved by the third functional layer.

3.4.1 The Cells

The basic object in the data schema is thus the cell. A cell represents by definition a logically related set of design data. It describes a functional part of an integrated circuit in terms of primitives of a certain view type, as well as references to other cells of the same or different view type. The cell is the appropriate unit of exclusive access for manipulation of the design data by the user. The DDMS does not interpret the representation details of a cell; these are handled by the design tools.

3.4.2 Relationships and View-Types

It is generally accepted that the complexity of VLSI design can effectively be managed by partitioning the design data into hierarchically related objects while introducing multiple levels of abstraction (view types) at which a design can be described. The relationships between cells of possibly different view types are administered explicitly by an equivalency mechanism. By introducing the object types *hierarchy*, *equivalence*, and the attribute *view-type* in the data schema we allow the designers and their tools to exploit the hierarchical and equivalence relationships, e.g. for top down

design or layout to circuit extraction. In our decomposition, equivalent cells are not required to have isomorphic hierarchical trees of subcells.

A view of an object can have a more general meaning. In the smalltalk system [Goldberg83] a view is an abstraction of an object that automatically can be generated by object methods. We use view-type in a different way, as the classification of an object, e.g. it is an attribute of an object with a static value. This necessary in our environment because of the evolutionary nature of the design data.

3.4.3 The Version Mechanism

The version mechanism (Figure 3.6) permits several cells to exist as the different versions of a module, i.e., bearing the same name, while a version status is attached to each of them [Dewilde86a, Leuken85, Wolf86, Wolf88].

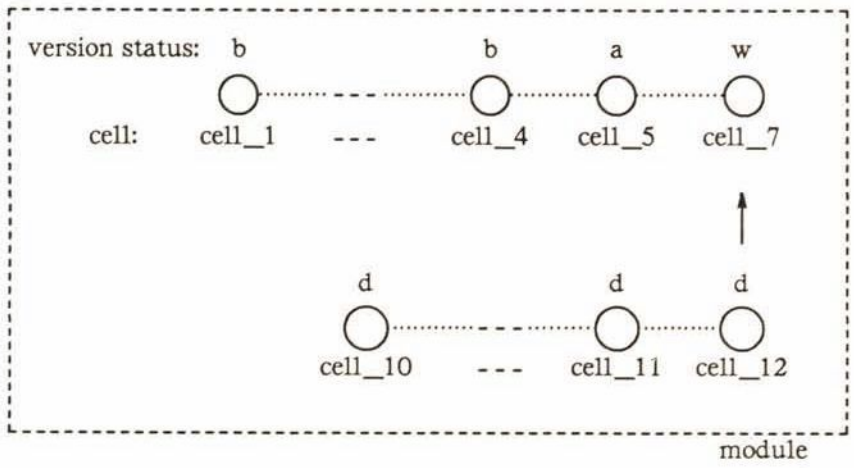


Figure 3.6. Version mechanism

The version statuses available are *backup*, *working*, *actual* and *derived*. The *actual* version status is unique in a particular module. The *actual* cells of the different modules form a (hierarchically) consistent set. An update

transaction roughly proceeds as follows. A cell that has to be updated is checked out. After updating, the data will be checked in as a *working* cell, which can be verified and/or reedited independently of other design activities [Bayer80]. When a cell has reached some definite state, it may be added to the hierarchically consistent set by invoking the *install* command. After successful completion the installing procedure will give the *working* cell the *actual* version status. The former *actual* cell will obtain the *backup* version status. The hierarchical links from *actual* and *working* cells calling this cell will be redirected to the new *actual* cell. Only one *working* cell can exist per module; it is the only cell of a module that can be modified by a designer at a particular time.

It is possible to have several *derived* cells per module. These are temporary cells derived from other representations, allowed to co-exist for verification purposes or automatically generated. A selected *derived* cell may be added to the version chain. Thus, our module consists of a linear chain of cells, each with an explicit version status.

3.4.4 Locking and Concurrency Control

A concurrency control mechanism synchronizes the execution of the design transactions in a multi designer and multi process environment. A commonly used mechanism is locking. In a system with a lock mechanism access to a database object is allowed when the transaction owns a lock on it. Design transactions differ from classical database transactions in that the former are of long duration. Consequently, the whole design transaction is not known in advance and common locking techniques are not suitable here, see also [Bancilhon85].

In our data schema the lock mechanism of the design data is represented by the object type *lock*. It is an aggregation of *cell* (i.e. the object to lock), *tool* (i.e. the design tool who submits the transaction), and *lockmode* (i.e. the type of the lock). Types of locks are: *readonly*, *write*, and *attach*. During a read transaction on a hierarchically consistent cell (i.e. with version *actual*), it should be locked with *readonly*. Editing a cell is only allowed on a

working version, on which occasion it is locked exclusively for write. Thus, the version mechanism contributes to the concurrency control mechanism by omiting conflicting combinations of transactions, i.e. the read-write conflicts [Bernstein83]. This solution is suitable for transactions of long duration. Design tools frequently put some additional information into a cell. The transactions involved should lock the cell with *attach*. This type of lock is not necessarily exclusive. The ranges, i.e. the sets of streams accessed, of certain design tools might be disjoint, allowing for concurrent execution even within the cell environment. A table containing the compatible tools will be consulted before granting a lock request of *attach* type.

3.5 *The Design System Architecture*

To describe the distributed situation we need a new data schema on top of the data management schema presented so far. A verbose characterization of the design management environment is as follows (Figure 3.7 and Figure 3.8). Design objects that constitute an integrated circuit are grouped together in a *project*. A *project* is a set of related design objects that describe an integrated circuit. It provides a logical context for the designers. There can be several designers working simultaneously on the same project. The group of designers that can work in a project is administrated in the *configuration database*. Passwords and log-on dates are recorded here too. To distinguish projects on the same machine, a project has a *project owner*. The *project owner* is usually the name of the *logical machine* where the design data of the project is completely stored, in combination with a chosen name, for example "hostname:project1". This provides an unique identification of a project within a community of workstations. A *logical machine* constitutes a central processing unit and a, possibly distributed, file system. Each integrated circuit is designed using a particular design process. Properties of the technology used are stored in the *technology database*. Each designer has its own private workspace, represented in the data schema as *user database*. Here, a designer can store private data, which allows him for example to override some system configuration defaults. The *universe* is

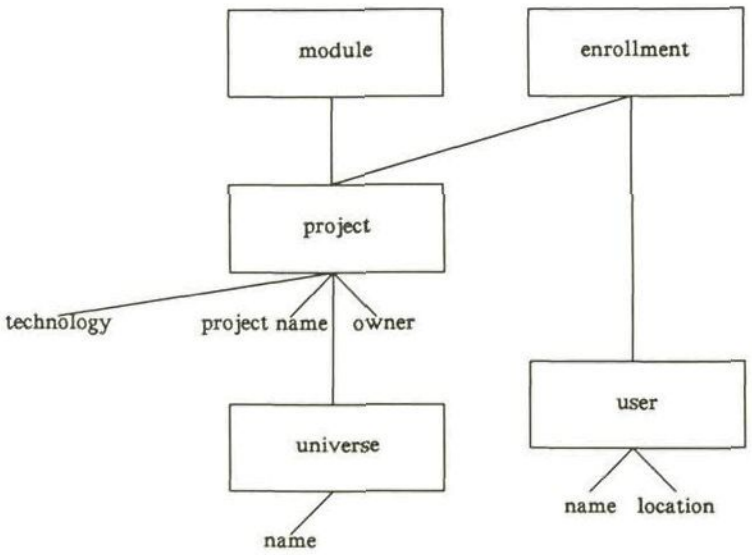


Figure 3.7. Project data schema

the collection of projects, the technology database, the configuration database and the user databases.

The design data of a team of designers working on the same integrated circuit, is stored on a logical machine in a project. Each project is managed by a *design management process*. The *design management process* maintains the meta design data of the project. All design transactions taken by a designer working on a design object in a project are by consent of the design management process. The design management process controls the concurrent sharing of design data, the version evolution, stores hierarchical relations, equivalence relations, grants and administrates access to the project.

In the data schema and design management architecture presented so far no provisions are made for distributed design management. In the next chapter an extension of the data schema will be presented that includes provisions for distributed design.

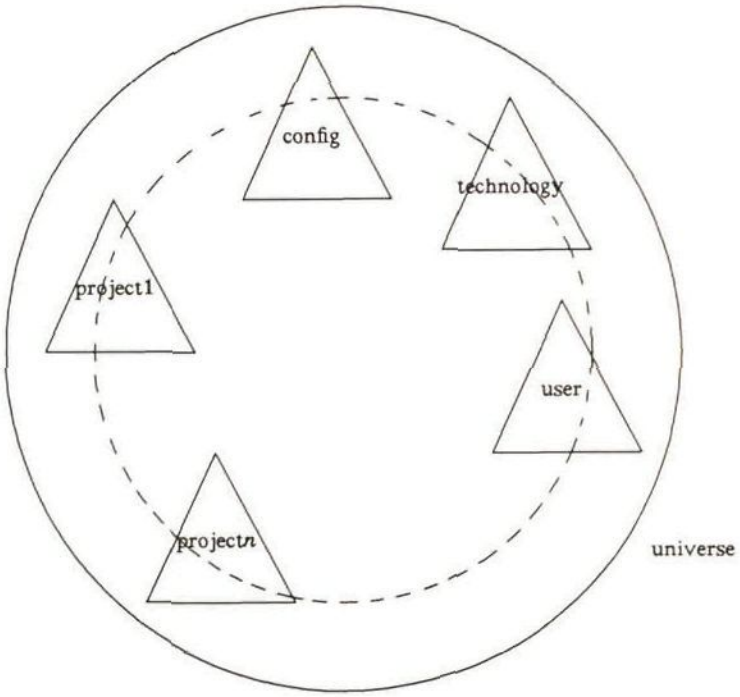


Figure 3.8. Project environment

3.6 *A Logically Distributed Environment*

The logical distribution of the design data across different databases is expressed in the data schema by the object type *project*. A project represents a collection of cells, thereby offering a local context in which the activities of a designer take place. The logical distribution of the design data is visible to the designer, providing a clear and simple concept of his environment. The project acts as the unit of authorization, the software verifies access permissions when designers or tools try to enter the project. Furthermore, the concept of project can be used to administer certain properties of the cells that belong to a certain project at a global level. For instance, we administer the technology of the cells at the project level,

restricting the cells within that project to being designed using this technology.

3.6.1 The extended data schema

Formalizing the previous description of the design data manager environment and consequently incorporating the distribution features are the next tasks. This will be achieved through the implementation of a new data schema. In the previous chapter we introduced a design system environment. The most important aspects of the design system environment are:

— The data schema, describing relations between design data. These relations are stored in the meta design data.

— A design management process (DMP), that manages the meta design data within a *project*.

In a modern design environment a group of designers work on their individual workstations, to design an integrated circuit in a reasonable amount of time. This introduces the problem of distributed design management, which we should envision as residing on top of the design data schema. Distribution means that

1. the data schema has to be extended to include information about where design data is stored and its status.
2. if the network between the workstations breaks down, designers can continue to work without major interruptions.
3. the performance of the system should be acceptable.
4. data transfers on the network should be as minimal as possible.
5. The physical distribution of the data should be invisible for the designer.

A distributed design management system creates the problem of localization of design data at different places within one project and the problem to

maintain the consistency of the meta design data and the design data.

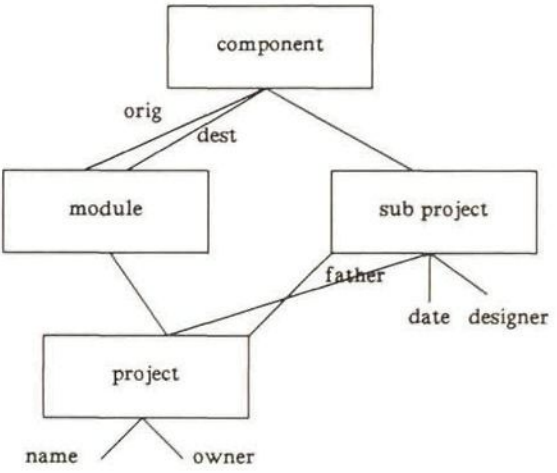


Figure 3.9. Extended project data schema

We propose a solution, where the *project* is divided in *subprojects* (Figure 3.9). A *subproject* consists of a set of design objects, and is a project. If a designer wants to modify some cells of a project, he asks the design management process of his local project to provide him with design data from a remote project. The local design management process contacts the remote design management process. The remote design management process verifies the status of the design data and makes the design data available if the status is in order. The local design management process creates a sub-project with the attribute subproject-owner equal to the remote design management process project owner. In the subproject-configuration database (Figure 3.10) data is stored about who created the sub-project and when it was created. The design data in question can consist of one cell or an hierarchy of cells, depending on the designers request. At the same time the remote design management process creates also a sub-project with the appropriate attributes set. Both design management processes, now have information about where the design data is present and who owns it.

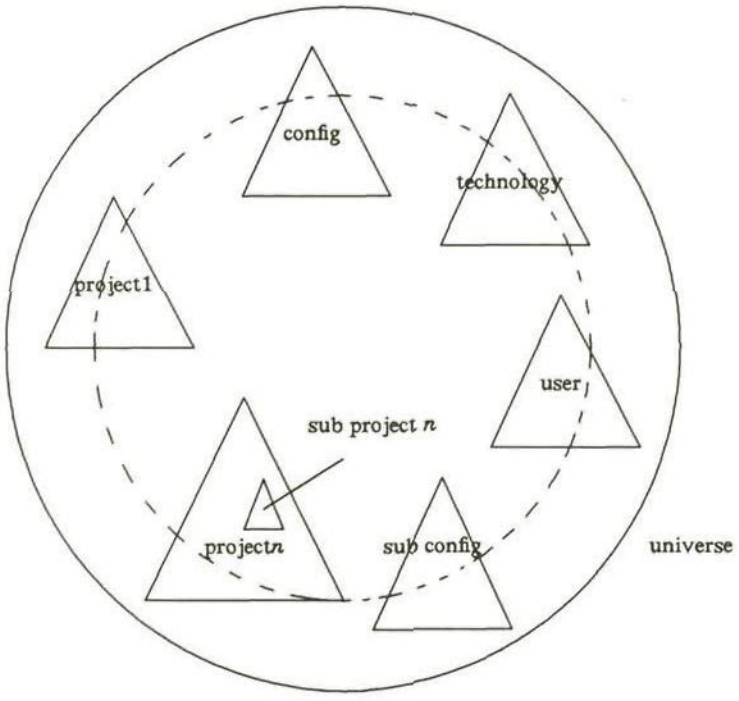


Figure 3.10. Extended project environment

Depending on the mode wherein the designer made his request to provide him with design data, the version attributes and lock attributes are altered on the remote project. If the mode was read-only, the requested version is made available and a read lock is set. If the mode was edit the version attributes are possibly changed and a write lock is set, and the version is made available to the local design management process. The design data is not available if a write lock has already been set on the requested design data. In addition to the design data that has been copied from the remote project to the local project, the meta design data belonging to the requested design data is copied also. The designer working on the local project can now work completely independently from the remote project. If the design data was copied from the remote project with mode edit, after the editing is

done, it has to be installed on the remote project. This includes updating the design data on the remote project as well as updating the meta design data on the remote projects.

The subproject-owner attribute is used to verify the access rights of a designer. If the subproject-owner name of a subproject is different than the *project owner* name, permission to modify or access design data has to be obtained from the design management process managing the project of the *project owner*.

3.6.2 The Distributed Design Data Manager

The data schema of the distributed design manager has some additional data types. The added type subproject allows for the storage in the meta design data of information consisting of: the name of a subproject, the subproject owner, the list of all cell names in a subproject and the physical location of the design data of each cell. Since the locking strategy is identical in the distributed environment, no change is necessary. The lock modes are *write* and *read* [Widya88]. In all situations a cell can be edited by only one designer.

This distributed design manager requires some extra functionality. The extra functionality is transparent for the software designer. He sees no difference between a tool running in a non distributed environment and a distributed one. Some additions are made internally in the design manager. First, the functions of the design manager have to check the attribute projectowner. This will decide if a tool can access design data locally, or if the design data should be copied from another machine. Secondly, since the data schema is extended, additional hard coded queries are needed. In principle, queries could be handled by a general purpose query interpreter. However, for efficiency reasons special purpose functions are to be added.

Two new design management tools are necessary. The first one enables the designer to create a subproject. The second one installs the subproject in the project.

The program that creates a subproject has as arguments the name of a project, the name of a root cell and a number representing the required hierarchical depth of the tree. It copies the design data of each cell in the tree to the appropriate machine and copies the relevant meta design data. The relevant meta design data encompasses the first order relationships of each cell. The program also updates the meta design data of the project owner and the meta design data of the subproject owner. Design tools can now work normally on the subproject machine, as long as the design data is present. If a design tool wants access to design data not present on the machine, the design manager will first copy all relevant data from the project owner machine after which the design tool can continue to operate. It is possible that several copies of a working version exist. It is the task of the design manager to assure that a change of a working version is broadcasted to all subprojects were this version is present.

After the designer has finished his work, the subproject should be installed on the project owner machine. The install design management tool takes care of this task. Using the transaction history mechanism it installs all modified cells.

3.6.3 Examples

We will discus the procedure to edit a cell on a workstation in different cases. The situation is (Figure 3.11):

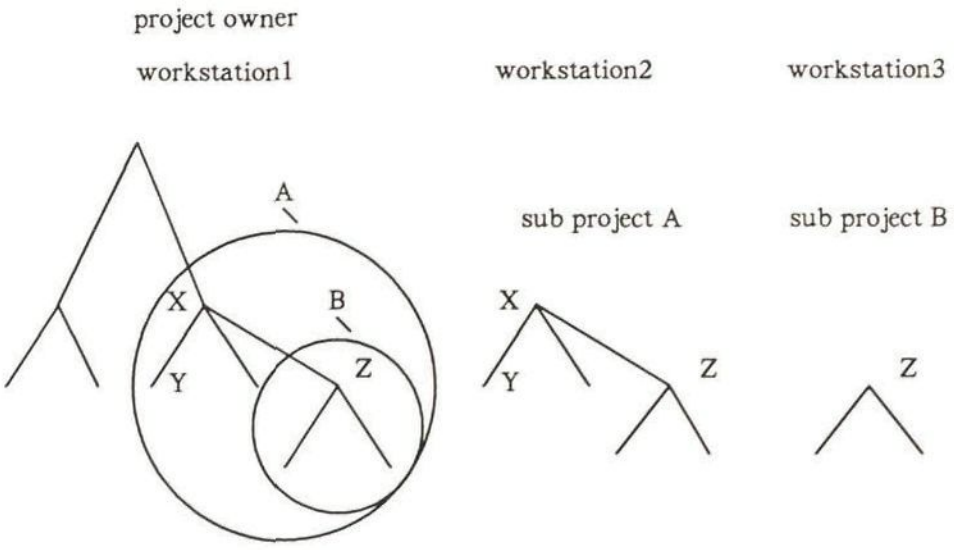


Figure 3.11. Example of data distribution on workstations

— Workstation 1 is the *project owner*.

— Workstation 2 has checked out subproject A. The cells in subproject A are checked out with lock mode *read*, and all the cells have the version status *actual*.

— Workstation 3 has checked out subproject B. The cells in subproject B are checked out with lock mode *write*, and all the cells have the version status *working*.

The *project owner* has stored in its sub-configuration database the following information:

— that cell X is in subproject A, lock mode *read*.

— that cell Y is in subproject A, lock mode *read*.

— that cell Z is in subproject A and in subproject B.

— that cell Z is edited (lock mode *write*) in subproject B.

The cases which we will now present, will demonstrate that only one copy of a cell can be edited at any time on any machine.

Case 1. Suppose workstation 2 wants to edit cell X. Actions:

1. Workstation 2 does a checkout of cell X for editing. The intended result will be that the version status of cell X will be changed from *actual* to *working*.
2. The checkout function sees that workstation 2 is not the owner of cell X. Workstation 2 inquires the lock mode of cell X with the *project owner*.
3. The *project owner* checks its tables for cell X, it has lock mode *read*, it changes lock mode to *write* and answers workstation 2 OK.
4. The checkout function changes the version status *actual* to the version status *working* on workstation 1 and workstation 2.

Case 2. Suppose workstation 2 wants to edit cell Z. Actions:

1. Workstation 2 does a checkout of cell Z for editing. The intended result is that the version status of cell Z will be changed from *actual* to *working*.
2. The checkout function sees that workstation 2 is not the owner of cell Z. Workstation 2 inquires the lock mode of cell Z with the *project owner*.
3. The *project owner* checks its tables for cell Z, has lock mode *write*, answers workstation 2 NOTOK, since the cell is edited by workstation 3.

4. The checkout function returns with an error message. Cell Z continues to have the version status *actual*.

Case 3. Suppose workstation 2 wants to checkin cell X after editing. Actions:

1. Workstation 2 does a checkin of cell X. The intended result is that cell X with version status *working* will be copied from workstation2 to workstation1.
2. The checkin function sees that workstation 2 is not the owner of cell X. Workstation 2 inquires the lock mode of cell X with the *project owner*.
3. The *project owner* checks its tables for cell X, has lock mode *write*, edited by workstation 2, answers OK.
4. the checkin function sends a checkin request to the *project owner*, and then the design data of cell X. Depending on the mode of the checkin, the *project owner* changes the lock mode of cell X in its sub-configuration tables.

Case 4. Suppose workstation 3 wants to checkin cell Z after editing. Actions:

1. Workstation 3 does a checkin of cell Z. The intended result is that cell Z with version status *working* will be copied from workstation3 to workstation1.
2. The checkin function sees that workstation 3 is not the owner of cell Z. Workstation 3 inquires the lock mode of cell Z with the *project owner*.

3. The *project owner* checks its tables for cell Z, has lock mode *write*, edited by workstation 3, answers OK.

4. The checkin function sends a checkin request and the design data to the *project owner*.

The install procedure, i.e. the procedure that installs a *working* version of a cell in the hierarchical consistent *actual* tree of a design, can be started on any workstation. If the procedure is started to install a cell in a subproject, the install procedure will copy the working version of the cell to the *project owner* machine, where after it continues to run on the *project owner* machine. If for some reason the network connection breaks down, a designer can continue to work on his workstation. The install procedure still works correctly in this situation, however it can not copy the working version to the *project owner* machine. In this case the install procedure will install the cell in the version chain of the subproject. This method of working has an advantage and a disadvantage. The advantage is that a designer can continue to work without interruption. The disadvantage is that the versions of a cell created locally have to be merged in the version chain of the *project owner*, after the network connection is available again. This could be done by a special tool, which can only operate in close contact with the designer.

3.6.4 Discussion

There are three important issues, when implementing a distributed data manager. The first question is: how important is autonomy. If you want to be able to continue to work after a network crash, copies of design data and meta design data have to be made. If you find autonomy not important and you have a network file system, no copies of the design data have to be made. The meta design data is also copied in this case.

The next issue is performance. It takes computer resources to make copies of data. Thus, it is important to copy only that part of the design data that actually will be used by a designer. If a subproject is created, the question is: which versions of a cell will be copied. There are several possibilities.

First, only the actual version is copied. This can be done without consequence for the database consistency. In this case it is impossible to edit a cell in a subproject. Secondly, the actual and, if it exists, the working version are copied. This means that multiple copies of an editable version exist. The distributed data manager has to maintain the consistency of the database. This could be achieved by one of in the following ways. 1). Only one copy of a working version can actually be edited. This can be implemented by checking designer names, or by allowing only one subproject to edit a working version, until the version is installed. 2). If a working version is changed, the updated version is broadcasted to all other users. As third possibility it is possible to copy the actual version and copy the working version only when desired. After editing, the working version remains in the subproject, until it is installed. In this situation only one copy exists.

The third issue is the consistency of the database. As we saw before, it is important that only one editable working version exists in the universe. The locking procedure of the distributed data manager guarantees that only one designer can edit a version at a particular time. However, if multiple copies of a working version exist, several different version can come into existence. This is a undesirable situation. The distributed data manager should prevent the existence of more than one editable working version in a universe.

3.7 *Conclusions*

We argued that the design object, being manipulated as a single entity by VLSI design applications, should play a dominant role in the definition of the organization of the design data. An initial data schema was constructed, reflecting the object type *design-object* as the aggregation of a number of attributes that characterize a design object.

Taking the design object as the basic entity for further modeling will result in a conceptually uniform framework for a design data management system. In our approach the detailed design data, i.e. structural/behavioral

descriptions of pieces of hardware, is concentrated within atomic objects. Although access to these design objects is provided by the design data management system, it does not interpret their representation details; these are handled by the tools. Instead, the design data management system maintains information about the design objects, i.e. the *meta design data.*

The meta design data describes how the design objects, at this level viewed as atoms, are related to each other. The version information of an object, locking information, ownership etc. are also considered meta design data, being maintained by the design data management system. The concept of the design object provides a basis for a consistent system philosophy, not being blurred by view dependent issues.

We have presented a distributed design management system for the VLSI design environment. A semantic data model is used to model and structure the design data. To increase design speed a distributed environment is necessary. The data schema is extended with *subprojects*, representing the temporary distribution of the design data in a community of workstations.

Dividing our distributed data schema in projects, subprojects and cells with their attributes, provides us with the next database properties.

1. The individual databases are autonomous. They are physically independent.
2. The individual databases are homogeneous.
3. A project, or a design database has a horizontal fragmentation. The distribution of the design data is based on subsets of cells (subprojects). Counter part of a horizontal fragmentation is a vertical fragmentation where the distribution is based on subsets of attributes.
4. The design data is temporarily partly redundant. Some cells may be present on one or more workstations at a particular time.
5. Most importantly the access of the design data will be primarily local. After a copy has been made from the project-owner to the local

workstation no further data transfers over the net are necessary. In most cases the access times of data over a net is at least 20% slower than accessing the data on a local disk.

6. The project-owner knows where copies of cells are present and their status, because this information is stored in the meta design data.

7. Communication will primarily exist between the project-owner and a local workstation.

In the next chapter we will describe a design data management system architecture, which incorporates the presented data schema. The two most important aspects are the tool interface and the user interface.

References

Afsarmanesh84. Afsarmanesh, H. and McLeod, D., "A Framework for Semantic Database Models," *Proc. NTU Symposium on New Directions for Database Systems*, (May 1984).

Bachman69. Bachman, C.W., "Data Structure Diagrams," *Data Base* 1(2) pp. 4-10 (1969).

Bancilhon85. Bancilhon, F., Kim, W., and Korth, H.F., "Transactions and Concurrency Control in CAD Databases," *Proc. IEEE ICCD*, pp. 86-89 (1985).

Batory85. Batory, D.S. and Kim, Won, "Modeling Concepts for VLSI CAD Objects," *ACM Trans. on Database Systems* 10(3) pp. 322-346. (Sept 1985).

Bayer80. Bayer, R., Heller, H., and Reiser, A., "Parallelism and Recovery in Database Systems," *ACM Trans. on Database Systems* 5(2) pp. 139-156 (June 1980).

Bekke83. Bekke, J.H. ter, "Database Design (in Dutch)," *Stenfert Kroese*, (1983).

Bekke85. Bekke, J.H. ter, "De Effectiviteit van Relationele Systemen," *Proc. Conf. Data: Beheer en Controle*, pp. 19-28 NGI, Sectie EDP-Auditing, (May 1985).

Bekke86. Bekke, J.H. ter, "OTO-D: Object type oriented data modeling," Report 86-02, Delft University of Technology, Delft (1986).

Bernstein83. Bernstein, P.A. and Goodman, N., "Analyzing Concurrency Control Algorithms When User and System Operations Differ," *IEEE trans. on Software Engineering* SE-9(3) pp. 233-239 (May 1983).

Bic86. Bic, L. and Gilbert, J.P., "Learning from AI: New Trends in Database Technology," *IEEE Computer Magazine* 19(3) pp. 44-54

(March 1986).

Buchmann84. Buchmann, A.P., "Current trends in CAD databases," *Computer-Aided Design* **16**(3) pp. 123-126 (May 1984).

Codd70. Codd, E.F., "A Relational Model of Data for Large Shared Data Banks," *Comm. of the ACM.* **13** pp. 377-387 (1970).

Dewilde86. Dewilde, P., Annevelink, J., Leuken, T.G.R. v., and Wolf, P. v.d., *Intelligent VLSI Datamanagement*, Delft University of Technology (1986).

Dewilde86a. Dewilde, P., Leuken, T.G.R. van, and Wolf, P. van der, "Datamanagement for Hierarchical and Multiview VLSI Design," pp. 1.1-1.29 in *The Integrated Circuit Design Book: Papers on VLSI Design Methodology from the ICD-NELSIS Project*, ed. P. Dewilde, Delft University Press, Delft (1986).

Goldberg83. Goldberg, A. and Robson, D., "Smalltalk80 The Language and its Implementation," *Addison-Wesley Pub. Co*, (1983).

Hardwick87. Hardwick, M. and Spooner, D.L., "Comparison of Some Data Models for Engineering Objects," *IEEE CG&A*, pp. 56-66 (1987).

Katz83. Katz, R.H., "Managing the Chip Design Database," *IEEE Computer Magazine* **16**(12) pp. 26-35 (Dec 1983).

Leuken85. Leuken, T.G.R. van and Wolf, P. van der, "The ICD Design Management System," *Proc. IEEE ICCAD - 85*, pp. 18-20 (1985).

Lyngbaek84. Lyngbaek, P., *Information Modeling and Sharing in Highly Autonomous Database Systems*, Ph.D. Thesis, Univ. of So. California, Los Angeles (August 1984).

McLeod80. McLeod, D., "Abstraction in Databases," *ACM special issue, Proc. of the workshop on Data Abstraction, Databases and Conceptual Modelling*, pp. 19-25 (1980).

Shipman81. Shipman, D.W., "The Functional Data Model and the Data Language DAPLEX," *ACM Trans. on Database Systems* 6(1) pp. 140-173 (March 1981).

Sidle80. Sidle, T.W., "Weaknesses of Commercial Data Base Management Systems in Engineering Applications," *Proc. 17th IEEE Design Automation Conference*, pp. 57-61 (June 1980).

Smith77. Smith, J.M. Smith and D.C.P., "Database abstractions: Aggregation and Generalization," *ACM Trans. Database Systems* 2(2) pp. 105-133 (June 1977).

Tsichritzis76. Tsichritzis, D.C. and Lochovsky, F.H., "Hierarchical Database Management: A Survey," *ACM Comput. Surv.* 8 pp. 67-103 (1976).

Widya88. Widya, I., Leuken, T.G.R. v., and Wolf, P. v.d., "Concurrency Control in a VLSI Design Database," *Proc. 25th Design Automation Conference*, (1988).

Wolf86. Wolf, P. van der, "Conceptual Design of a Design Data Management System for VLSI Design," MS-Thesis, Delft University of Technology, Delft (July 1986).

Wolf88. Wolf, P. van der and Leuken, T.G.R. van, "A Distributed Data Management System for VLSI Design," *Proc. 25th Design Automation Conference*, (1988).

4. SYSTEM ARCHITECTURE

4.1 Introduction

It is generally acknowledged that a crucial part of an integrated VLSI design environment is a *Design Data Management System* (DDMS), to form the kernel around which all design tools are integrated [Katz83, Newton86]. The DDMS provides essential facilities to the tools and the designer, based on the knowledge it has of relationships that are present within the design data. To mention are such aspects as concurrency control, crash recovery, support for design evolution in a hierarchical multi-view context (versioning, maintaining verification statuses and equivalence relationships) and physical distribution across multiple workstations. These facilities offer the tool developer a way to manage the complexity by exploiting their properties and relieve the designer from the burden of organizing his design data. These issues were discussed in the previous chapter.

In this chapter, we will focus on the interface between VLSI design tools and such a DDMS in an environment where both the tools and the DDMS are constantly evolving. In this environment, the tools should depend as little as possible on the DDMS to avoid extensive tool modifications with each new release of the DDMS. Furthermore, the DDMS must be open-ended: It should be easy to add new tools to the system in such a way that they become a consistent part of the design environment. Thus, what is needed is a decoupling of the software development and evolution of the DDMS on the one hand and the tools on the other hand.

The only way to effectuate this decoupling is by a standardization of the *Data Management Tool Interface* (DMTI) [Meijs87] between the tools and the DDMS. Via the DMTI the tools obtain access to the design data, while taking advantage of the facilities that are provided by the DDMS. In practice, a DMTI is a set of library functions that can be used by the tool developer, in such a way that he does not need to have a detailed

understanding of the implementation of the DDMS. In this chapter, we will introduce such a DMTI, based on a transaction schema that formalizes the procedural aspects of the communication between the tools and the DDMS.

4.2 Tool Interface Requirements

The following requirements can be formulated for the Tool Interface (TI) (see chapter 2):

- The TI must bring about efficient interaction between the tools and the DDMS.
- The TI must not be tailored to specific tool features or design methodologies. It should be universal, to result in an *open-ended* design system where the DDMS acts as a free-for-all public repository that can communicate with any type of tool and environment.
- The TI must not be tailored to specific features of a DDMS. For example, it must allow interfacing to DDMS's with or without version control, concurrency control, multiple view-types, etc. When this requirement is met, the tools can actually be "plugged in" in the same way in any DDMS, whether it concerns different releases of a DDMS at a certain site or DDMS's at different sites.

In summary, a TI should offer some degrees of freedom, but at the same time the necessary discipline to facilitate software evolution and exchanges. Our opinion on how to introduce this discipline is expressed most concisely by the following thesis:

Thesis

> The optimal way to decouple the development and evolution of the DDMS and the tools is to agree on a common *transaction schema*, and reflect this in the definition of the TI.

A transaction schema consists of a set of procedures which are to be executed in a particular sequence, according to which the tools obtain access

to the design data. Our transaction schema is based on a number of assumptions that we believe to be general within the context of chip design.

As argued in chapter 3, the design data is organized on a *per project* basis. A *project* offers the designer a local context in which a collection of *cells* is present. A cell represents a logically related set of design data, describing a functional part of an integrated circuit in terms of certain primitives, as well as references to other cells. It is the appropriate unit of exclusive access for manipulation of the design data by the user. Within a cell the actual design data is organized as a set of *streams*, but no assumptions are made on the contents of these streams.

The agreement on these assumptions permits the definition of a transaction schema, and hence a TI, that localizes the interaction between the tools and the DDMS. Any tool modifications that are required to adapt the tool to other implementations of a DDMS will then be *strictly local* and will not alter the *structure* of the program. Consequently, they can be done with much less effort.

We can in this respect draw an analogy with a public library, where the books (cells) are organized in racks based on certain criteria as author, title or language, without making any assumptions on the actual text that is contained in the books. If the internal organization is hidden from the public, we can have the following procedure employed. To borrow a book, a form describing it should be filled in. This form is accepted at the desk and the book is returned (if it is available). If this is the only direct interaction between the public and the library personnel, it permits the library (DDMS) to be reorganized invisibly. For example, new racks can be placed or internal procedures automated, without having this local interaction with the public (tools) changed.

4.3 The Transaction Schema

In this section, the TI transaction schema will be defined. As a consequence of the recognition of project, cell and stream as units of access, the

transaction schema will be a layered one. The effect of a tool on a design environment is called a *tool-execution*. It is a (possibly interleaved) sequence of *project transactions* bracketed by an *initialization* and a *termination*. Similarly, a project transaction is a (possibly interleaved) sequence of *cell transactions* bracketed by an *open project* and a *close project*. A cell transaction is a (possibly interleaved) sequence of *design data transactions* bracketed by a *checkout* and a *checkin*, while a design data transaction is a sequence of *design data IO operations* bracketed by an *open stream* and a *close stream*. A design data IO operation is either a *read operation* or a *write operation*.

We present these definitions graphically in Figure 4.1. The boxes on one level represent a sequence of actions, executed from the left to the right. Child boxes specify a refinement of the father action. A starred box represents an iteration and boxes with a small circle imply alternatives. This diagram is a variation on a *Entity-Action diagram* as defined in [Jackson83].

4.4 Related Work

There are just a few tool interfaces for design data management purposes [Brouwers87]. Most likely, a reason for this is that design data management is a relatively new field in CAD/IC. Recently one tool interface was introduced: OCT [Harrison86].

The OCT data manager provides simple procedures to store and retrieve design data. The basic design unit in OCT is a facet, the attributes of a facet are cell name and view type. A facet can be a set of nets, transistors, boxes, edges, etc. The OCT interface contains functions to open and close a facet, and read and write objects in a facet. Also it provides functions to attach and detach different objects. It is possible to attach a transistor to a net, or a box to a layer, or a facet to another facet. The OCT data manager has no concurrency control, has no version mechanism and has no support for high level design transactions. In other words, OCT incorporates a simple storage mechanism, and provides none of the services of a more advanced DDMS.

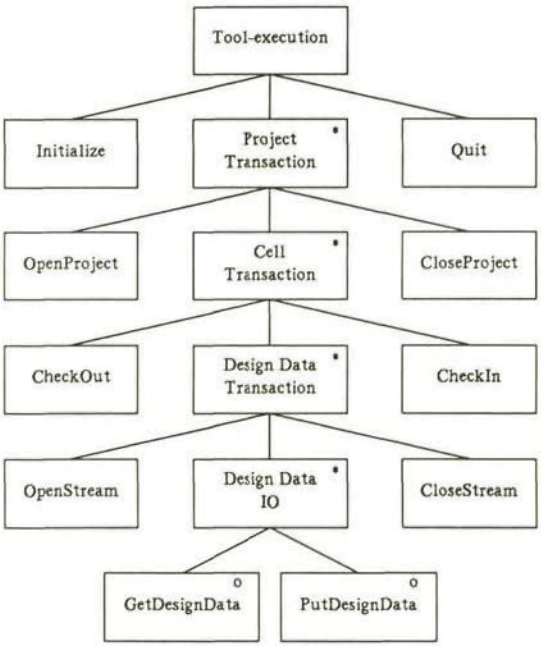


Figure 4.1. Transaction Schema

4.5 The Tool Interface

4.5.1 Concepts

The TI defines a set of functions that should be used by the tool developer to obtain access to the design data. Basically, there is one function in the TI for each leaf of the tree in Figure 4.1. These functions co-operate with each other in such a way that this access will proceed in accordance to the transaction schema presented in the chapter 4.3. That is, they implement the procedure according to which access to the design data can be obtained.

Access to either the design environment, a project, a cell or a stream can be obtained by executing the corresponding opening-bracket function, as represented by the leafs at the left-hand side of the tree in Figure 4.1. A

transaction is terminated by executing the corresponding closing-bracket function at the right-hand side. In between, lower-level transactions can be performed.

The functions in the TI communicate with each other by means of *abstract data types*, called *keys*. The contents of these keys is not fully specified in this TI definition, but can depend on the particular DDMS at hand.

There are four types of keys, one for each layer:

DM_UNIVERSE	universe transaction key,
DM_PROJECT	project transaction key,
DM_CELL	cell transaction key,
DM_STREAM	design data transaction key.

The key returned by an opening-bracket function at some layer is part of the argument list of the functions at the next lower level and of the closing bracket function. This allows the interleaving of more than one sequence of calls of lower-level functions. The closing bracket function invalidates the key.

Typically, a key contains all necessary information about the object for which access was obtained, for use by the lower-level functions. Depending on the particular DDMS at hand, this can for example be physical location, access permissions and state.

Each key contains a pointer to the next higher level key that was passed as an argument to the function returning the lower level key, so that the complete context is known at the lowest levels. Also, all keys with the same "parent key" are linked together in a list that is attached to this parent key. This facilitates error recovery and automatic clean-up actions. For instance, the closing bracket functions could terminate all their lower level transactions still in progress. When a key is invalidated by the corresponding closing bracket function, it is removed from the list.

The opening bracket functions take as arguments, apart from a parent key, an identification of the object for which access is to be obtained and possibly an access mode. In an actual implementation these arguments will reflect certain features of the DDMS: When it provides version control at the cell level, the parameters of the *dmCheckOut* function must somehow identify the version to be checked-out [Meijs86].

After verifying and establishing the access, appropriate information for further use by the lower level functions is stored in the key that is returned. As a direct advantage, the visibility of particular features of the DDMS can be confined to a small number of places in the TI. The fact that there is a version mechanism at the cell level is visible in the argument list of dmCheckOut, but only there. It is not visible in the argument lists of the lower level functions, but completely hidden in the implementation of the *DM_CELL* key.

In an actual implementation of a DDMS, appropriate actions will be associated with each function of the TI. By introducing the right levels of intervention, the TI as presented here provides a natural and universal framework to localize these actions. In the next section, the functions at the different layers are presented, together with some examples that illustrate how particular DDMS features can be embedded in the TI.

4.5.2 The TI Functions

4.5.2.1 Global Initialization and Termination

Two functions are needed for global initialization and termination. They establish and release contact between the tool and the design environment.

- dmInit (toolname): unikey
- dmQuit (unikey)

DmInit is the opening bracket function of a tool-execution and returns a *DM_UNIVERSE* key. This key contains information about the design environment (for example hostname, user-id, process-id, working directory etc.) in which the tool is executed. The main purpose of *dmInit* is to initialize the TI interface. The tool identifies itself by means of the

argument toolname. An action that might be performed by the DDMS is to consult a tool database to obtain more detailed information about the tool.

DmQuit is the closing bracket of a tool-execution. It takes care of the necessary clean up operations.

Between dmInit and dmQuit the project transactions are executed.

4.5.2.2 Project Transaction Layer

At this level such aspects as projects, libraries and distributed databases can be handled.

- dmOpenProject (unikey, projid, openprojmode): projectkey
- dmCloseProject (projectkey, closeprojmode)

DmOpenProject initiates a project transaction and returns a *DM_PROJECT* key. This key contains information about the particular local database or project, represented by projid, and the access mode, represented by openprojmode. Actions that might be performed are verification of the access rights, retrieval of technology information, setting up LAN connections or contacting a local manager process. The project key will be passed as an argument to the functions at the cell transaction layer.

DmCloseProject terminates the project transaction. The details of this operation are specified by closeprojmode. In a physically distributed environment, actions to be performed might include returning local copies, closing LAN connections, etc.

4.5.2.3 Cell Transaction Layer

The functions at this layer take care of aspects of cell transactions. To mention are concurrency control, versioning, view-types, maintenance of verification statuses and equivalence relationships, etc.

- dmCheckOut (projectkey, cellid, checkoutmode): cellkey
- dmCheckIn (cellkey, checkinmode)

DmCheckOut is the opening bracket function of a cell transaction. Its arguments are a *DM_PROJECT* key, identifying the particular project for which access rights have been obtained by *dmOpenProject*, and an

identification of a particular cell, denoted by cellid. The checkoutmode parameter specifies what type of interaction is to take place, so that the TI can anticipate on it. For example, in a multi-user environment any number of simultaneous readonly or only one single update transaction can be allowed at the same time.

DmCheckIn is called to terminate a cell transaction initiated by *dmCheckOut*. Cellkey has been obtained from *dmCheckOut*. Checkinmode specifies how the transaction has to be terminated, e.g. whether the transaction should commit or rewind. Actions that might be performed include removal of locks, updating of verification statuses, deletion of scratch data that was created for recovery purposes, etc.

In order to hide the versioning capabilities of a DDMS and possible different view-type values, a browse function is provided.

- dmBrowse (projectkey): cellid

This function has one argument projectkey and returns a cellid. *dmBrowse* is a machine-man interface routine, which displays on a screen a representation of the data schema of the project and allows the user to select a cell.

In a DDMS that recognizes *cell* as a unit of access, an administration will be present on top of these cells. Compare this to the card-trays of a public library. This administration is used by the DDMS itself to maintain some information about the cells. It should also be accessible to the tools, for instance, to allow them to obtain a cell-list or to ask for and insert equivalences. For this purpose the functions *dmGetMetaDesignData* and *dmPutMetaDesignData* are provided.

- dmGetMetaDesignData (projectkey, request, arguments)
- dmPutMetaDesignData (projectkey, request, arguments)

DmGetMetaDesignData (DmPutMetaDesignData) can be used to obtain (store) information from (into) the local administration of the project identified by projectkey.

By means of the request argument and a variable argument list, the specific retrieval or update operations (queries) can be passed to the DDMS. Which queries can actually be formulated depends on the conceptual model that is employed by the DDMS for the organization of its administration. For instance, a DDMS supporting equivalence relationships between cells of (possibly) different view-types, will accept queries on this information.

4.5.2.4 Design Data Transaction Layer

These functions are at the lowest level, i.e. closest to the physical IO. They map the design data to and from the physical storage structure being exploited.

- dmOpenStream (cellkey, strname, iomode): streamkey
- dmCloseStream (streamkey, closestreammode)
- dmGetDesignData (streamkey, format, arguments)
- dmPutDesignData (streamkey, format, arguments)

DmOpenStream returns a *DM_STREAM* key which will be used by *dmGetDesignData, dmPutDesignData* and *dmCloseStream.* The cellkey argument is obtained from *dmCheckOut.* Strname identifies a stream of data belonging to the cell identified by cellkey. Iomode specifies the mode of access to the data, e.g. read or write. *DmOpenStream* can check this mode against the checkout mode. For example, it should be forbidden to open a stream for writing if the checkout mode was readonly.

In a Unix implementation, a stream will probably be implemented as a file, but this need not be the case. For example, experiments have shown that it is possible to transparently map IO operations onto shared memory as an efficient channel for direct inter-tool communication via the TI. Anyway, *dmOpenStream* knows where and how to find the data, given the information that is present in cellkey and its parent *DM_PROJECT* key.

DmGetDesignData and *dmPutDesignData* perform the actual in- and output of design data. They know the mapping to and from the storage structure employed. Streamkey is obtained from *dmOpenStream. DmGetDesignData* and *dmPutDesignData* do not restrict the formats of the detailed design data that can be transferred. The mechanism used is very similar to that of the

printf() and scanf() functions in C.

DmCloseStream must be called to terminate a design data transaction. Files can be closed or allocated memory can be freed. Closestreammode specifies the details.

4.5.2.5 Calling Pattern

As an illustration of how these functions co-operate, we present in Figure 4.2 a calling pattern of the functions of the TI. The layered structure of the transaction schema is reflected in the indentation of the code.

```
DM_UNIVERSE unikey;
DM_PROJECT projectkey;
DM_CELL cellkey;
DM_STREAM streamkey;

unikey := dmInit (toolname);

  projectkey := dmOpenProject (unikey, projid, openprojmode);

    cellkey := dmCheckOut (projectkey, cellid, checkoutmode);

      streamkey := dmOpenStream (cellkey, strname, iomode);

        dmGetDesignData (streamkey, format, arguments);
        dmPutDesignData (streamkey, format, arguments);

      dmCloseStream (streamkey, closestreammode);

    dmCheckIn (cellkey, checkinmode);

  dmCloseProject (projectkey, closeprojmode);

dmQuit (unikey);
```

Figure 4.2. TI calling pattern

4.5.3 Discussion

The TI introduced here formalizes the procedural aspects of the interaction between VLSI design tools and a DDMS that provides a number of facilities to these tools. It does not prescribe the semantics of the design data, which would be unacceptable. It also does not completely prescribe the argument lists of the TI functions or the interpretation of these arguments. For example, the version of a cell to be checked-out will be an argument of the dmCheckOut function in a DDMS with a version mechanism, and is unimportant otherwise.

Openness is retained by avoiding tool-specific aspects in the TI definition. Furthermore, we illustrated the ability of the TI to absorb DDMS-specific features that not necessarily have to be visible to the tools. By offering a proper set of "anchor points", the standard TI greatly facilitates software exchangeability: Modifications that are required to adapt a tool to another DDMS will be strictly local and will not alter the structure of the program. In fact, most of the work can usually be done mechanically, for example with a stream editor.

The TI introduced in this paper has been implemented in the Delft Release 2 and 3 of the ICD-NELSIS system [Dewilde86], which currently contains over 40 DDMS-intensive CAD/IC tools such as layout editors and generators, design rule checkers, extractors, a placement and routing system, simulators, etc. The decoupling introduced by the TI turns out to be extremely useful for software development, as it permits DDMS and tools to evolve separately to a large extent. As an illustration of the openness, we remark that integration of the KIC layout editor [Billingsley83] via the TI was completed in about two days.

From the experience we have gained, it appears that the principles introduced in this paper indeed allow to do proper data management efficiently.

As an additional benefit, the TI completely hides operating system dependencies. There are no path-names, file access modes, system calls etc. visible in the tools. Consequently, it should be easy to port the design system to other operating systems by changing the TI library. For example, a VMS implementation can exist side by side to a UNIX implementation. It is also possible to port some (small) parts of the system to PC-like systems.

Our approach should be distinguished from such general object management systems as offered by the Portable Common Tool Environment (PCTE) [Bull85]. We recognize four levels of intervention, at which access to the environment, projects, cells and streams is arranged. This layered transaction schema has been adopted specifically for application in the area

of VLSI design. From our point of view, general object management systems are a possible implementation vehicle for a DDMS for VLSI design.

The standardization that we propose emphasizes the procedural aspects of the interaction between the tools and the DDMS. Standardization of 1) the features of the DDMS that can not be absorbed by the TI but must be visible to the tools and 2) the semantics of the design data that is transferred, should also be considered.

The particular features of the DDMS that must be visible to the tools manifest themselves most profoundly in the arguments of the functions at the project- and cell transaction layers. The more one can agree on the effect of such generally accepted aspects as versions and view-types on the arguments of the standard functions is reached, the easier it becomes to obtain *consistent integration* of a foreign tool in a specific DDMS environment.

Also for the sake of consistent integration, agreement should be reached on the semantics of the design data. If tools are exchanged and should operate in close co-operation with each other, this kind of standardization across different sites will be very useful. For several tools to communicate they should understand each other. This is the point where the ideas presented in this paper meet the EDIF standardization activities [EDIF87], which also require a common interpretation of design data at different sites.

4.6 Tool Communication

Implementing a design data management system involves two important decisions. First, one has to decide which data schema will be used to model the design data. Then a design management program has to be written that incorporates the chosen data schema. Important elements of the program are the data handling and the communication with other programs. In our design environment several types of design tools are used. The first group are the application processes. Layout editors, simulators, design tools in general belong here. The second group are the designer interface processes.

Typically these are the command interpreter and window manager processes. The third group are the data management processes. Process group 1 and 2 are connected with designer activities, while process group 3 is connected with project activity.

In the distributed environment two communication mechanisms are important. The first type of communication will typically take place at one site, while the second type of communication will take place between different sites.

4.6.1 Communication Between Group 1 and Group 3 Processes

Application processes (group 1) require access to the meta design data managed by the data management process (group 3). Communication between group 1 processes and group 3 processes will take place on one site.

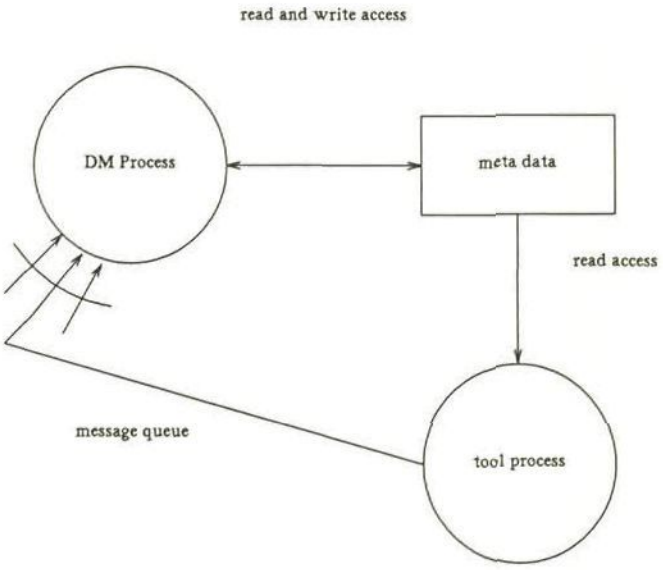
There are several possibilities to implement an efficient communication mechanism between processes, given the Unix environment.

1. pipes, communication mechanism between father and son processes.
2. named pipes (System V only), communication mechanism between unrelated processes.
3. shared memory and message queues.
4. sockets, communication mechanism between unrelated processes on different sites using IPC primitives.

The problem is to define a communication mechanism with the following properties:

1. Fast access to the meta design data.
2. Local as well as remote data communication.
3. Controls the update of the meta design data.

Figure 4.3 depicts our solution.



The tool process and the design management process have direct read access to the meta design data, because the meta design data is stored in shared memory. The tool process has write access to the meta design data via the design management process. This is a queue based mechanism.

Figure 4.3. Meta data access

Our solution for the communication problem is a combination of the use of shared memory and the use of message queues. We divide the queries in read-only requests and in update requests, and use different communication mechanisms for them. Meta data is stored in shared memory. This is done because application tools frequently execute requests to obtain information about the design data. Queries that require read operations on the meta design data, are executed by functions that have direct access to the information stored in the meta design data. These functions also have the possibility to deny access to the meta design data, for example when the meta design data is being updated.

Queries that can alter the contents of the meta design data, are implemented with functions that use message queues to send the request to the design management process. If an application process wants to modify the contents of the meta design data, it sends a message to the design management process, requesting some write operation. When the design management process receives a request to modify the meta design data, it disables all read accesses to the meta design data. Then it executes the request, updates the meta design data and enables the read access to the meta design data again. Disabling and enabling the access to the meta design data is done using signals and semaphores. It is also possible to defer a request as long as the design management process 'thinks' it is best to do so. Since only one process has write access to the meta design data, no conflicts arise about multiple write operations.

4.6.2 Communication Between Group 3 Processes

This type of communication takes place between design management processes. Since projects can reside on different sites, the implementation of a communication mechanism uses the facilities of the network between the sites. In a Unix environment an ethernet connection and TCP/IP protocols are generally present. In this case all the requests are send to the processes using the TCP/IP networking interface. The sent requests are mostly inquire queries. The same communication protocols and communication procedures can be used for both message queue mechanism as for the TCP/IP network mechanism. A design management process not only polls its message queues but also its ethernet connection to see if any request has arrived. All received requests have the same format and require the same execution.

4.6.3 Communication Between Group 1 Processes

Sometimes a design step requires several design tools to execute. Here frequently the need exists to pass design data from one design tool process to the next design tool process. The Unix environment provides no mechanism to create a multi data communication mechanism between processes. However, this type of data communication can be simulated

using named pipes or by multiplexing the design data on one io-channel. An other possibility to implement multiple pipes is the use of shared memory segments. Each communication mechanism has its own shared memory segment assigned to it. One process writes data in the memory segment while the next process reads the data from the segment. The advantage of this type of data communication is that disk access is considerably reduced, which leads to faster execution of the design tools. The disadvantage is that the size of the freely available memory of a system is reduced.

4.6.4 Group 2 Processes

The designer interface processes communicate with the designer and with the design management process. The function of these type of processes is to provide information and to activate design tools. All relevant information is stored in the meta design data. The processes in group 2 have the same requirements as the processes in group 1, i.e. fast access to the meta design data and local as well as remote data communication. However, they don't need mechanisms to share design data. They do need fast access to the meta design data, and a communication mechanism to communicate with group 3 processes.

4.6.5 Query Language

To obtain information from the meta design data or to store information in the meta design data a query language is needed. The query language should support design transactions functions as well as administration functions and browse functions.

We believe that just a few language constructs can provide us with the functionality we need. The language constructs can for example be taken from a set operation language (a recent example is the the enhanced C language) (Figure 4.4) [Katzenelson83, Katzenelson85], or a dedicated data manipulation language [Bekke83] (Figure 4.5).

exists expression1 in expression 2 suchthat expression 3

If there exists an element in expression2 such that expression 3 is true then the lvalue of expression1 is set to the value of such an element.

add expression1 in expression2 position expression3

Add adds the element expression1 to the sequence expression2 in position expression3.

Figure 4.4. Query language using the EC syntax

get subject its property list where qualifying predicate

Given a type (subject) and the qualifying predicate the system will be able to determine the set of instances of the subject.

insert subject its list of assignments

The subject is inserted with the list assignments.

Figure 4.5. Query language using the OTO-D data manipulation language

We propose an inline expansion of the query language constructs using a simple preprocessor. This type of integration is also used in Ingres [Held75]. The preprocessor program expands the query language constructs into C language code and into remote procedure calls. The advantage of this approach are: readable source code, fast program execution and easy to use. Code optimization also can be done by the preprocessor. Since there are only a few query language constructs the preprocessor is simple to develop.

4.6.6 Communication Procedure

When a request of a design process can not be executed locally, the request has to be send to the appropriate remote machine (see chapter 3.6). The transmission of a request can be implemented using Remote Procedure Calls (RPC) [Birrell84].

The idea of remote procedure calls is based on the mechanism for transfer of control and data, which normally takes place within a computer program composed of several procedures. The same mechanism can be used both for

a program running on a single computer as for a program running on several computers, connected through a computer network. When a remote procedure is invoked, the calling environment is suspended, the parameters are passed across the network to the environment where the procedure is to execute, and the desired procedure is executed there. When the procedure has finished and has produced its results, the results are passed back to the calling environment, where execution resumes.

When making a remote call, five pieces of program are involved: the local design process, the RPC client interface library, the communication package, the RPC server interface library, the remote design process. Their relationship is shown in Figure 4.6.

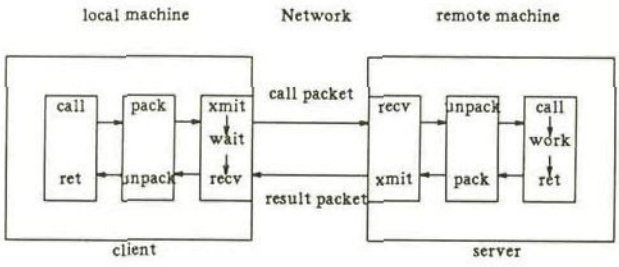


Figure 4.6. Interactions for a remote procedure call

At the moment that the design process makes a remote call, it makes a call to a procedure which will invoke the corresponding procedure in the RPC client interface library on the local machine. This procedure is responsible for placing a specification of the target procedure and the arguments into one or more packets and asking the TCP/IP network interface to transmit these in a reliable way to the remote machine. On receipt of these packets, the TCP/IP network interface on the remote machine passes them to the RPC server interface library. The RPC server interface unpacks them and again makes a normal local call, which invokes the appropriate procedure on the remote machine. Meanwhile, the local process in the local machine is suspended awaiting a result packet. When the call in the remote machine

completes, it returns to the RPC remote interface library and the results are passed back to the suspended process in the local machine. There they are unpacked and the RPC client interface library returns them to the local design process. The TCP/IP network interface is responsible for retransmissions, acknowledgments, packet routing, and encryption. The user sees no difference between a procedure executing on the local machine and a procedure executing on a remote machine.

Besides using the TCP/IP network interface for the communication mechanism, message queues can be used if both client and server process execute on the same machine. The message queue mechanism is a faster mechanism for passing packets than the TCP/IP network mechanism. This is particularly useful in a distributed environment where programs need to communicate with local programs as well as remotely executed programs. The interface library decides which communication mechanism to use. If the request can be executed locally, the message queue mechanism is used, otherwise the TCP/IP network interface is used. For a program developer the use of the RPC library is transparent.

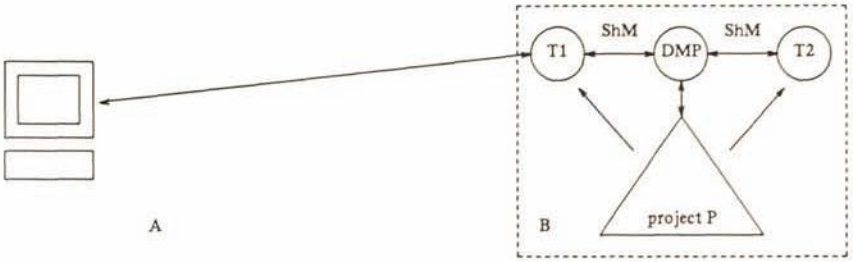
4.6.7 Implementation

A Data Management Process (DMP) controls the access to the meta design data of a project and communicates with design processes and other DMP's. Efficient access to the meta design data is achieved by using shared memory techniques. Requests from processes to update the meta design data are placed in a queue. The DMP executes these requests sequentially [Bernstein83]. Since the DMP also controls the read access to the meta design data, it is capable of prohibiting access while a critical update is in progress.

Requests that concern cells in other projects are sent to the appropriate DMP, using RPC message passing techniques. In this way the physical distribution of the design data in a distributed workstation environment is made transparent to both the tools and the designer.

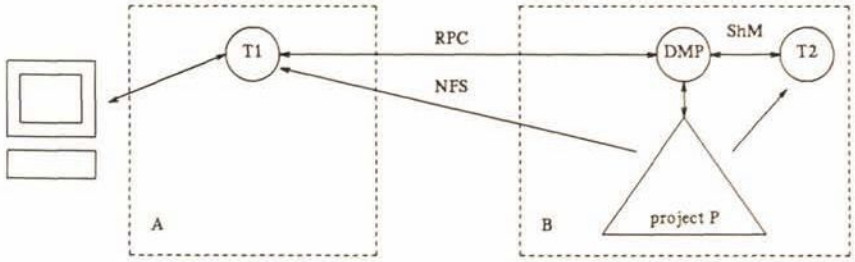
In a design environment several workstation configurations are possible. In a simple environment there is no physical distribution of design data and no distribution of tool processes (Figure 4.7). If the file server concept is used together with some diskless workstations, it is possible to distribute the tool processes (Figure 4.8). In this case interactive programs will have a faster response time. A Network File System (NFS) provides access to the design data on the file server. A major drawback of this environment is the limited reliability of the network. If the network is not accessible, designers will be unproductive. In Figure 4.9 a workstation environment is depicted, where the workstation has local storage. In this case there is physical distribution of the design data and distribution of tool processes. If the network is not accessible, the designer can continue without interruption. However, special tools are necessary to merge the design data when the network is again in working order.

The DDMS presented in chapter 3.6 supports three workstation environments. In the first case (Figure 4.7), the DMP provides tool communication mechanisms, which don't need networking capabilities. In the second case (Figure 4.8), the DMP provides tool communication, using a RPC mechanism. This makes it possible to run design tools on physically different workstations. In the last case (Figure 4.9), the DMP provides the mechanisms to distribute design data on physically different machines, using the sub-project concept. In all three cases the DMP controls the access of the meta design data and thus the access of the design data.



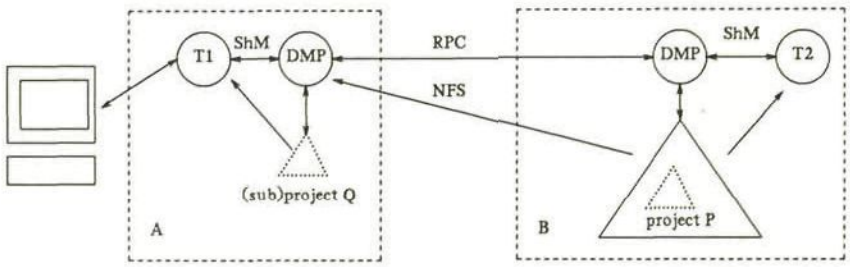
ShM = Shared Memory, DMP = Data Management Process, T = Tool.
A = dumb terminal, B = workstation.

Figure 4.7. Project Environment for one Workstation



ShM = Shared Memory, DMP = Data Management Process, T = Tool.
A = diskless workstation, B = fileserver.

Figure 4.8. Project Environment for one file server and several diskless Workstations



ShM = Shared Memory, DMP = Data Management Process, T = Tool.
A = workstation, B = fileserver.

Figure 4.9. Project Environment for one file server and several Workstations

4.7 User Interface

A User Interface (UI) provides the designer with an interface, which enables him to browse through the meta design data of his universe. If his universe is partitioned into a technology database, a project database, a user database and a configuration database, then the UI can provide him information about which technologies are present in the technology database, which projects there are in the project database and so on. After the designer has obtained the name of a project, he could open this project and ask the UI to list all modules in this particular project. This type of information is typically meta design data. At this level the designer is not interested in design data. The user interface program should be written based on a distributed windowing system. The X-window system [Gettys86] is the only windowing system that has this functionality and is a well accepted standard.

Besides the browse function, the UI should have the functionality of a command interpreter; there is a command syntax which enables the designer to start design tools with the proper arguments. The command syntax, the query language, can also provide the designer with a more sophisticated

type of browsing, "probing", if it supports wildcards. For example:

```
GET module
        ITS name
        WHERE name == "D*"
```

which will return all module names starting with a "D".

Figure 4.10 presents the process structure.

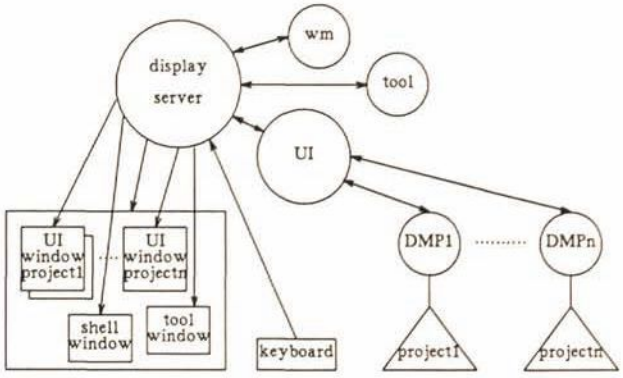


Figure 4.10. The process structure of an UI and several DMP's

Assumed is an environment with a windowing system like X-windows. After startup, the UI opens a window on the screen. The designer can now browse through information available to him from the configuration database. When the designer has obtained the name of a project, the designer can open this project. The UI opens another window on the screen and tries to establish communication with the DMP which controls the particular project. After a successful connection, the designer has access to the meta design data of that project. It is now possible for him to start design tools in this project.

The difference between the UI and an ordinary design tool is that the UI only accesses meta design data and will not access design data. Both type of tools use the same TI and use the same communication mechanisms.

4.8 Conclusion

At this point of our development, we are able to verify that, the functional requirements as stated in chapter 2 have been achieved. The approaches used to address the functional requirements of a DDMS are:

— A data model and a data schema.

— Tool interfaces.

— Distributed data management facilities.

— An user interface.

In chapter 3 a data model and a data schema were presented. The data schema describes the structural semantics of the data in a VLSI design system. Also a data schema was presented that structures the distribution of data across a network of design systems. In chapter 4 a tool interface and an user interface were presented. These form the interface between the design tools and the system software and between the design tools and the designers of VLSI circuits.

The functional requirements of a DDMS are (chapter 2):

— Tool integration.

— Data Exchange.

— Management and Control.

— Data Management.

— System Interface.

Data exchange is achieved by using a standard language which allows for the transport of a description of a VLSI circuit from one design system to

another [EDIF87]. The design system has tools available which will translate/extract this language to/from its database.

Management and control of data and design data management are achieved by the introduction of a version chain, concurrency control and a transaction schema. The linear version chain and concurrency control allow for the existence of several versions of a cell, while only one of these versions can be edited by only one designer at a time. The versions have a status and there is a default selection of a version, which provides the designer with information about a version and which relieves him of the task of identifying each version, when he wants to access it. Formulated where the following requirements for a tool interface:

- The tool interface must bring about efficient interaction between the tools and the DDMS.
- The tool interface must be independent of specific tool features or design methodologies. It should be universal, to result in an *open-ended* design system where the DDMS acts as a free-for-all public repository that can communicate with any type of tool and environment.
- The tool interface must be independent of specific features of a DDMS.

The first requirement is met in the following way. The data in the DDMS is split in two types: design data and meta design data. The meta design data is controlled by the DMP, and the design tools can access the meta design data via the DMP. Design tools have direct access to the design data, after they have obtained the mandatory permission. The design tools can now built their own optimal data structure. They communicate with the database with two simple functions (Figure 4.1).

The second requirement is achieved by introducing a small set of complex objects: the universe, the project, the cell and the stream. No assumptions are made on the exact attributes of these types. Besides these complex object types, the DDMS supports hierarchy and equivalence relations. This allows for the use of any design methodology and any type of design tool.

The implementation of the DDMS makes use of well known standards as the C language and the X-windows system. There are strong indications that the DMTI proposed by us will become a standard [SECT87].

The third requirement is achieved by the introduction of the dmbrowse function in the DMTI. This function hides DDMS specific features like versioning. Concurrency control and transaction recovery is also hidden by the DMTI. The DDMS has its own user interface. This means that no design tool program has to be changed to incorporate any new DDMS browse facility.

4.9 Results

A DDMS for VLSI design that provides an open framework for the integration of design tools and relieves the designer from the burden of organizing his design data has been presented. After identifying the basic entity that is involved in a design transaction, the cell, a data schema representing the logical organization of the VLSI design data was developed. The OTO-D semantic data model, offering a set of well defined modeling constructs, permitted us to formalize the semantics of the meta design data. Openness of the DDMS was secured by avoiding incorporation of features of a particular tool set or design representation.

The resulting data schema is simple, yet powerful enough to reflect the basic requirements of the DDMS outlined in chapter 2. It incorporates such features as hierarchical decomposition, multilevel design, design transactions / concurrency control and versioning. Thus, it provides the basis for a DDMS that implements a set of global system facilities to provide a framework for the construction of a powerful design environment. The DDMS serves both the tools and the designer by making the information about the structure and status of the design available to them.

This DDMS-research has been carried out in the context of the ICD-NELSIS system [Dewilde86], which currently contains over 40 DDMS-intensive

CAD/IC tools such as layout editors and generators, design rule checkers, extractors, a placement and routing system, simulators, etc. These tools have all been equipped with the DMTI defined in this thesis. The decoupling introduced by this DMTI turns out to be extremely useful for our software developments, as it permits DDMS and tools to evolve separately to a large extent.

In the ICD-NELSIS system, several DDMS-experiments have been carried out and a prototype has been produced. Currently, work is under way towards a DDMS-release that comprises all facilities described in this thesis. The meta design data storage module exploiting shared memory is operational, including the query facilities. An attractive version mechanism and interactive browsing facilities are currently being implemented.

Performance comparison between a software release without design data management functions and the software release with the DDMS functions shows, depending on the design tool in question, a difference in execution speed between 0 and 20 percent. The decrease in performance has no relation with the function of a design tool. It depends on the number of design transactions requested by the tool.

Future work will, among other things, focus on powerful design management facilities that can guide the designer through the design process towards a correct design. The first ideas for this are presented in appendix C. Another area of interest is the management of information about such environmental aspects as technologies, tools, and designers, as a logical extension of the information modeling approach presented in this thesis.

References

Bekke83. Bekke, J.H. ter, "Database Design (in Dutch)," *Stenfert Kroese*, (1983).

Bernstein83. Bernstein, P.A. and Goodman, N., "Analyzing Concurrency Control Algorithms When User and System Operations Differ," *IEEE trans. on Software Engineering* SE-9(3) pp. 233-239 (May 1983).

Billingsley83. Billingsley, G. and Keller, K., "KIC: A Graphics Editor for Integrated Circuits," User's Manual, University of California at Berkeley (1983).

Birrell84. Birrell, A.D. and Nelson, B.J., "Implementing Remote Procedure Calls," *ACM Trans. on Comm. Sys.* 2(1) pp. 39-59 (Feb 1984).

Brouwers87. Brouwers, J. and Gray, M., "Integrating the Electronic Design Process," *VLSI Systems Design*, pp. 38-47 (June 1987).

Bull85. Bull,, GEC,, ICL,, Nixdorf,, Olivetti,, and Siemens,, *PCTE: A basis for a portable common tool environment*, Third edition, volume I 1985.

Dewilde86. Dewilde, P., Leuken, T.G.R. van, and Wolf, P. van der, "Datamanagement for Hierarchical and Multiview VLSI Design," pp. 1.1-1.29 in *The Integrated Circuit Design Book: Papers on VLSI Design Methodology from the ICD-NELSIS Project*, ed. P. Dewilde, Delft University Press, Delft (1986).

EDIF87. EDIF,, "Electronic Design Interchange Format, Version 2 0 0, Reference Manual," *EDIF Steering Committee*, Electronic Industries Association, (1987).

Gettys86. Gettys, J., Newman, R., and Fera, T. Della, "Xlib - C Language X Interface, Protocol Version 10," *MIT, Cambridge, Mass.*, (1986).

Harrison86. Harrison, D.S., Moore, P., Spickelmier, R.L., and Newton, A.R., "Data Management and Graphics Editing in the Berkeley Design

Environment," *Proc. IEEE ICCAD-86*, pp. 24-27 (1986).

Held75. Held, G.D., Stonebraker, M., and Wong, E., "INGRES - A Relational Database Management System," *Proc. 1975 Nat. Computer Conference*, AFIPS Press, (1975).

Jackson83. Jackson, M., *System Development*, Prentice Hall, Englewood Cliffs, N.J. (1983).

Katz83. Katz, R.H., "Managing the Chip Design Database," *IEEE Computer Magazine* **16**(12) pp. 26-35 (Dec 1983).

Katzenelson83. Katzenelson, J., "Higher Level Programming and Data Abstractions - A Case Study Using Enhanced C," *Software Practice and Experience* **13** pp. 577 - 595 (1983).

Katzenelson85. Katzenelson, J., *The Enhanced C Programming Language Reference Manual*, Technion - Israel Institute of Technology, Haifa 32000, Israel (March 21, 1985).

Meijs86. Meijs, N. v.d., Leuken, T.G.R. v., Wolf, P. v.d., Widya, I., and Dewilde, P., "Data Management Interface Definition," *ESPRIT project, code 991, WP1, task 1.*, (Dec 31, 1986).

Meijs87. Meijs, N. v.d., Leuken, T.G.R. v., Wolf, P. v.d., Widya, I., and Dewilde, P., "A Data Management Interface to Facilitate CAD/IC Software Exchanges," *Proc. IEEE ICCD '87*, (1987).

Newton86. Newton, A.R. and Sangiovanni-Vincentelli, A.L., "Computer-Aided Design for VLSI Circuits," *IEEE Computer Magazine*, pp. 38-60 (April 1986).

SECT87. SECT,, "Software Environment for CAD Tools," *The Sect Data Handling Committee*, CadLab, CNET, NMP-CAD, ES2, TU Delft, (1987).

5. APPENDIX A

5.1 Technology Database

The data schema of the technology database (Figure 5.1) is based on the EDIF [EDIF87] dataschema.

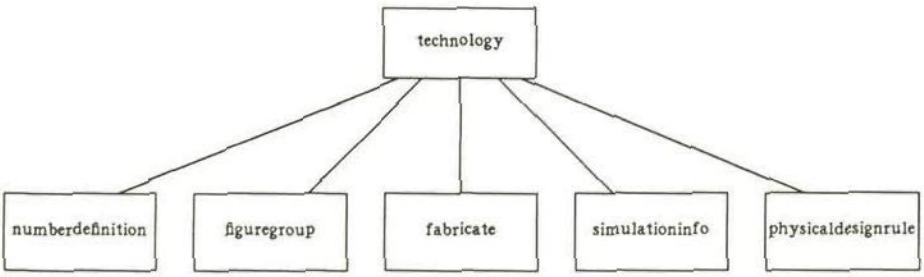


Figure 5.1. The Data schema of the technology database

The technology database contains all information related to the intended implementation of a design. Figure 5.2 gives an example of a technology description in the EDIF syntax.

The most important data types in the data schema are technology, figuregroup, fabricate, simulationinfo and physicaldesignrule.

5.1.1 Numberdefinition

The numberdefinition type contains scaling information for the designs in a project. Scale defines the relationship between numbers used in the project database and numbers outside of that project. For example, a cell can be designed on lambda grid, and lambda is four micron. Numberdefinition relates symbolic units to physical units.

```
(technology
  (numberDefinition
    (scale edifUnits externalUnits (unit unitType))
    ...
  )
  (figureGroup figureGroupNameDef
    (cornerType cornerType)
    (endType endType)
    (width width)
    (color percentRed percentGreen percentBlue)
    ...
    (property propertyNameDef ...)
    ...
  )
  ...
  (fabricate layerNameDef figureGroupNameRef)
  ...
  (simulationinfo
    (logicValue logicNameDef
    ...)
  ...
  )
  (physicalDesignRule
    (figureWidth ruleNameDef ...)
    ...
    (overlapDistance ruleNameDef ...)
    ...
    (interFigureGroupSpacing ruleNameDef ...)
    ...
  )
  (comment ...)
  (userdata ...)
)
```

Figure 5.2. A technology description in EDIF

5.1.2 Figuregroup

The figuregroup type defines graphical default characteristics. Attributes which may be given default values here are: cornertype, endtype, pathwidth, borderwidth, color, fillpattern, borderpattern, textheight and visible. The figuregroup type is used also within the physicaldesignrule type.

5.1.3 *Fabricate*

The fabricate type defines which figuregroups are intended to be used for physical fabrication and introduces the layer names.

5.1.4 *Simulationinfo*

The simulationinfo type collects all information about the logic values used for modeling. Values are defined by names and by specifying their characteristics, which may be: electrical, Boolean and relational between other logic values. Attributes in the logicvalue type are: voltagemap, currentmap, Booleanmap, compound, weak, string, dominates, logicmapoutput, logicmapinput, isolated, resolves, property, comment and userdata.

5.1.5 *Physicaldesignrule*

The physicaldesignrule type specifies a set of geometrical design rules applying to the actual design process. It is possible to represent rules for single figures using the attributes: figurewidth, figurearea, rectanglesize, figureperimeter, intrafiguregroupspacing, notchspacing and notallowed. Rules for pairs of figures are represented by the attributes: overlapdistance, overhangdistance, enclosuredistance and interfiguregroupspacing. The physicaldesignrule type also may include definitions of new figure groups which are created with the figuregroup type. The data schema should be extended if conditional design rules are to be stored in the technology database.

5.1.5.1 *The Tool Interface for the Technology Layer*

- dmOpenTechnology (envkey, techid, opentechmode): techkey
- dmCloseTechnology (techkey, closetechmode)

DmOpenTechnology opens the technology database and returns a *DM_TECHNOLOGY* key. This key contains information about the particular technology, represented by techid, and the access mode, represented by opentechmode. Actions that might be performed are verification of the access rights. The technology key will be passed as an argument to the functions in the technology access layer.

DmCloseTechnology terminates the technology transaction. The details of this operation are specified by closetechmode.

- dmGetTechnologyData (techkey, request, arguments)
- dmPutTechnologyData (techkey, request, arguments)

DmGetTechnologyData (DmPutTechnologyData) can be used to obtain (store) information from (into) the technology database with the technology identified by techkey.

By means of the request argument and a variable argument list, the specific retrieval or update operations (queries) can be passed to the interface function. Which queries can actually be formulated depends on the conceptual model that is employed by the DDMS for the organization of its technology database.

References

EDIF87. EDIF,, "Electronic Design Interchange Format, Version 2 0 0, Reference Manual," *EDIF Steering Committee*, Electronic Industries Association, (1987).

6. APPENDIX B

THE ICD DESIGN MANAGEMENT SYSTEM*

T. G. R. van Leuken
P. van der Wolf
Department of Electrical Engineering
Delft University of Technology
The Netherlands

ABSTRACT

To organize the design process of VLSI circuits and the design evolution involved, a design data management system is needed that supports reliable storage, concurrency control, hierarchical decomposition, multilevel design and version control. A design system data scheme can be based on a semantic (object oriented) data model, where objects are used for modeling the structures imposed on the design data. Using the basic constructs of the data model a data scheme can be defined that reflects the object types and their relationships encountered in VLSI designs.

A dominant part of the design data management system is the transaction manager, supporting versions and concurrency control. In our version mechanism some of the versions of each cell have a special status, thereby allowing automated selection of the right version on a certain request. This version mechanism offers the user a clear conceptual picture and supports the evolutionary development of a design in a highly automated fashion.

* This research was supported in part by the commission of the EEC under the 3744/81 program (ICD-contract).

6.1 Introduction

In the ever increasing complexity of integrated circuit design a new way has to be found to control and organize the design data and the design evolution. A team of integrated circuit designers usually generates huge amounts of interrelated data. They often work in an iterative way, making small changes in existing cell descriptions, in an attempt to decrease the dimension of the design or to improve the performance. This design process involves several levels of abstraction, called views. The system view, network view and layout view are the most used among them.

To organize the design process, making the designers job more cost effective, a design data management system is needed that supports reliable storage, concurrency control, hierarchical decomposition, multilevel design and version control. This will not only diminish the administrative task of the designer, but also allow the construction of more sophisticated design tools.

In the next section we will discuss the concepts of our design data management system. In section 3 we describe the version management and concurrency control. Conclusions and implementation results are given in section 4.

6.2 Basic Concepts

The development of a design data management system includes the definition of a conceptual model, reflecting the different types of data and their dependencies. When the dependencies encountered in design data are stored explicitly, they can be maintained and made available to both management tools and sophisticated design tools.

The basic entities in our design system are the design objects. A design object contains a logically related set of design data, describing a part of an integrated circuit in terms of primitives of a certain view, as well as references to other objects in the same view. The structuring of design data as a set of interrelated design objects provides us with a flexible design

system, because no assumptions are made about the views that have to be supported. Furthermore, the design object is the appropriate unit of exclusive access for manipulation by the user.

Although access to the design objects is provided by the data management system, it does not interpret their representation details. These are handled by the design tools. The data management system maintains information about the design objects: the meta data. The meta data describes how the design objects, at this level viewed as atoms, are related to each other. It contains all hierarchical information of a design description in a particular view, as well as equivalence information relating design objects in possibly different views. The version information of an object, locking information, ownership etc. are also considered meta data, being maintained by the data management system.

The logical organization of the meta design data is represented in a data scheme using a data model. The classical data models: the hierarchical, network and relational models, are all record oriented. Recently, new object oriented data models have been developed; the semantic data models [Lyngbaek84, Shipman81]. Using a semantic data model, data schemes can be defined as object types and relationships between object types. Each object type corresponds to a collection of objects (or entities) that share common properties. As opposed to the classical data models a well defined semantic data model may act as a formalized tool to support strict modeling.

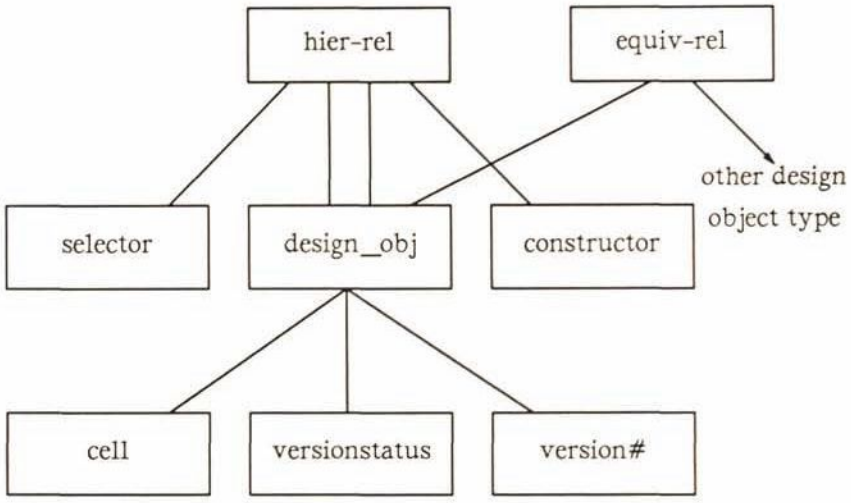


Figure 6.1. Design data management subscheme

The data scheme is divided in several similar subschemes, one per view. Each subscheme (Figure 6.1) contains a design-object type, whose instances agree with the design objects of a particular view. If the design data is structured in a hierarchical way, it seems natural that the data scheme reflects this structure. Each subscheme contains one type (hier-rel), whose instances correspond to a hierarchical relationship between two design objects, with an attached selector and constructor. Subschemes may be connected by one or more equiv-rel, whose instances are the equivalence objects linking two design objects from possibly different views. Based on this data scheme, reflecting the structure of the meta design data, dependencies are made available to both design tools and designers by the design data management system. The latter is supported by a user interface consisting of a set of browse commands.

6.3 *Versions and Concurrency Control*

Design data is a too precious resource to let it simply be overwritten on every design transaction. This would also consequently override any verification effort that was made before. A better approach is to create new versions at certain design transactions [Katz84], thereby not only offering a backup facility, but also simplifying recovery. This way the evolution of a design is remained to some extent and older versions can be used to explore possible design alternatives.

In principle it is possible to support whole trees of versions per cell, where users can select versions for updates without any conceptual constraints. This would result in a highly complicated version mechanism because of the hierarchical relationships between the versions of different cells. A clear concept has to be offered to the user for a version mechanism to become effective. A satisfying solution can be obtained by restricting the evolution to a linear chain of versions, while additional commands allow explicit manipulation of the version chain. One version has a special status, and will be used in an update transaction.

At any time the design data has to form a hierarchically consistent set. Unfortunately, consistency checks on every design transaction would mean that flexibility and performance of the design system degrade. Our version mechanism allows the coexistence of a hierarchically consistent set of design data and a set of newly created design objects for which only the downward consistency is maintained. Design data that has to be updated is checked out. After updating, the data will be checked in as a 'working' version, that can be verified and reedited independently of other design activities. When it has reached some definite state, it may be added to the hierarchically consistent set by invoking the 'install' command. This installing procedure performs some checks and will give the 'working' version the 'actual' status. The hierarchical links from 'actual' and 'working' versions of other cells calling this cell will be redirected to the new 'actual'. The former 'actual' version will obtain the 'backup' status. As a result, the version chain has

the following appearance (Figure 6.2).

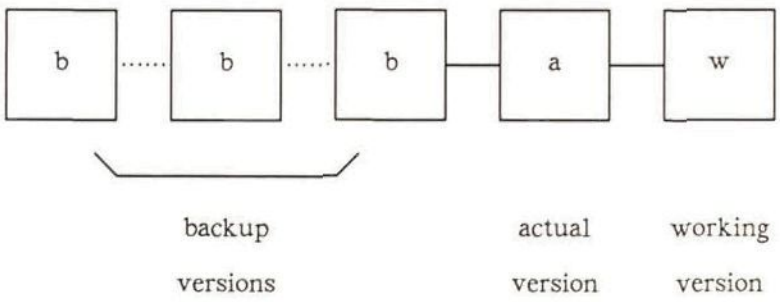


Figure 6.2. Design data management version chain

The 'actual' version is always present and is part of the hierarchically consistent set. One 'working' version and one or more 'backup' versions may be present. Because some of the versions of each cell have a special status, automated selection of such a version is possible. This relieves the user from explicit specification on every transaction. The 'actual' versions have global scope, meaning that when a designer refers to a cell for inclusion as a subcell in a composite object, the 'actual' version will be handed to him. As a result we have an hierarchically consistent set of 'actual' versions that is maintained implicitly, and a set of 'working' versions attached to it.

Besides the install command, other commands are present to manipulate the version chain. One of them is a restore command, that allows the user to make a 'backup' configuration 'actual' again. Other commands allow the removal of versions and the reusal of 'backup' versions by transporting them to a new cell.

VLSI designs typically are created by teams of designers working simultaneously on different pieces of the circuit. The design system should control concurrent sharing of design data. A locking mechanism is provided, the transaction lock, which ensures that only one designer can

generate a new 'working' version of a particular cell at a time. The version mechanism allows a flexible locking strategy, where all other cells remain available for update. Furthermore, any other designer can browse the 'actual' versions of the locked design data. In our implementation all designer processes in the same process group inherit the assigned access rights. This mechanism makes it possible to stop a layout editor session, start a verification program and resume the layout editor session after the verification program has terminated.

6.4 Conclusions and Status

We have implemented a design data management system that, based on an extendible data scheme, not only allows conventional bottom-up cell assembly, but also offers a framework for top-down design and silicon compilation. It supports multiple levels of abstraction, called views. Each view contains hierarchically related design objects, that may be mapped on equivalent design objects in other views.

The version mechanism presented proved to be successful. To the user it supports the evolutionary development of a part of a design in a highly automated fashion, based on a clear conceptual picture. Additional commands allow explicit manipulation of the version chain. To ensure exclusive access rights of a design object a flexible locking procedure is supported.

A data access layer provides us with an unified set of interface functions that execute the access operations on the design data. Because our data access layer is based on several generalities recognized in design transactions, forward compatibility is achieved. At the same time it offers data independence. At this moment we are investigating the implementation of a distributed object oriented design data management system.

References

Katz84. Katz, R.H. and Weiss, S., "Design Transaction Management," *Proc. 21st IEEE Design Automation Conference*, pp. 692-693 (1984).

Lyngbaek84. Lyngbaek, P., *Information Modeling and Sharing in Highly Autonomous Database Systems*, Ph.D. Thesis, Univ. of So. California, Los Angeles (August 1984).

Shipman81. Shipman, D.W., "The Functional Data Model and the Data Language DAPLEX," *ACM Trans. on Database Systems* 6(1) pp. 140-173 (March 1981).

Vogel84. Vogel, T., Wolf, P. van der, and Dewilde, P., "Conceptual Database Model ICD," Internal Report, Delft University of Technology, Delft (Oct 1984).

Wiederhold82. Wiederhold, G., Beetem, A.F., and Short, G.E., "A Database Approach to Communication in VLSI Design," *IEEE Trans. on CAD of Integrated Circuits and Systems* **CAD-1**(2) pp. 57-63 (April 1982).

7. APPENDIX C

DATA MANAGEMENT FOR VLSI DESIGN: CONCEPTUAL MODELING, TOOL INTEGRATION & USER INTERFACE*

P. van der Wolf, N. van der Meijs, T.G.R. van Leuken, I. Widya and P. Dewilde

Delft University of Technology
Department of Electrical Engineering
Mekelweg 4, 2628 CD Delft
The Netherlands

A Data Management System (DMS) for VLSI design is presented that supports hierarchical decomposition, multiple levels of abstraction, concurrency control and design evolution in a distributed workstation environment. Semantic data modeling techniques are employed to derive a simple, yet powerful, data schema that represents the logical organization of VLSI design data. The resulting DMS provides an open framework for the integration of design tools and relieves the designer from the burden of organizing his design data. The Data Management Browser (DMB) enables the designer to browse through the information that is maintained by the DMS about the structure and status of his design. The Data Management

* This research was supported in part by the commission of the EC under ESPRIT contract 991.

Interface (DMI) decouples the development of the tools and the DMS. The DMS-principles have been tested in practice.

7.1 INTRODUCTION

It is generally acknowledged that a crucial part of an integrated VLSI design environment is a *Data Management System* (DMS), to form the kernel around which all tools are integrated [Katz83, Brouwers87]. A proper DMS operates first of all as the common repository of design data: the design database. The tools create and modify the design data, while the DMS stores and maintains the design data, thereby guaranteeing consistency.

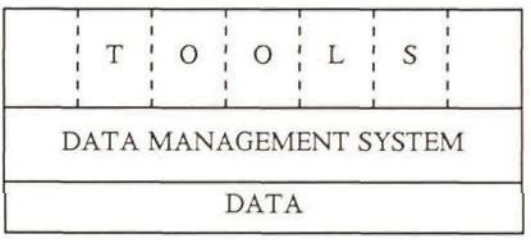


Figure 7.1. Tools integrated on top of DMS

Furthermore, a DMS has to supply additional services to provide a basis for the construction of an intelligent design system. Clearly, system integration is more than the definition of some common formats: Which copy is the latest version? Has this layout been extracted since it was updated and, if so, which circuit description was derived from it? If I change this layout, which other parts of the design will be affected? It is the ability to answer such questions that differentiates a true DMS from a simple data repository [Newton86].

In this paper we focus on the construction of such a DMS that serves both the *tools* and the *designer*. It must provide an intelligent framework for tool integration that permits the designer to easily retrieve information about the structure and status of his design. Going a bit more into detail,

we can identify a number of aspects that a modern DMS for VLSI design should cover:

- *A uniform interface to the tools.*
  A transaction mechanism is needed to control the interaction between the tools and the DMS. Via this interface the tools obtain access to the design data to perform updates or verifications, while taking advantage of the facilities provided by the DMS.

- *An interface to the designer.*
  Basically, the DMS is the bookkeeper of the design system. It must therefore offer an attractive user interface that allows the designer to browse through his design data and make inquiries about the status of specific parts of his design.

- *Views and hierarchy.*
  Generally accepted aspects of chip design are hierarchy and multiple view-types, i.e. levels of abstraction at which a design can be described. These aspects have to be supported by the DMS such that the designer and his tools can exploit them.

- *Concurrent design activities.*
  VLSI designs are created by teams of designers, working together in the same project environment. Moreover, each designer can have multiple tasks running concurrently. The DMS has to provide facilities to guarantee consistency under concurrent operations.

- *Version mechanism.*
  A version mechanism has to be incorporated to support the evolutionary development of a design. Also, a proper version mechanism potentially permits a more flexible concurrency control mechanism for long design transactions.

- *Verification statuses.*
  The evolutionary nature of the design process also requires the DMS to maintain the verification statuses of the design under construction. That

is, administer the statuses and warrant their consistency. Previously derived verification results can then be reused, which is particularly effective in combination with hierarchical design strategies. Secondly, the designer can get well-informed when deciding which operation to apply where. Important related aspects are the generation and maintenance of consistent documentation and the support for auditing when the design reaches its final stage.

- *Logically distributed design data management.*
  The DMS has to offer the designer a conceptual environment where the design data is logically distributed across different *projects*. Some formalism must allow him to refer from his own project to design descriptions created by other designers. In this way libraries also become an integral part of the design system.

- *Physically distributed design data management.*
  The DMS must operate in an environment of physically dispersed workstations and file servers, connected by some network.

- *Efficiency.*
  It is one thing to invent elegant concepts for the above mentioned aspects of data management, but it is yet another challenge to arrive at an implementation of a DMS that handles them efficiently. This is not only a matter of smart software engineering at the implementation stage, but also implies that particular choices have to be made at the conceptual design stage.

The facilities mentioned above offer the tool developer as well as the designer a way to manage the complexity by exploiting their properties. Yet, these properties have to be sufficiently general so that the DMS will not restrict the functionality of the design system.

Obviously, the construction of a DMS is not just a matter of straightforward implementation of the different facilities. A more fundamental approach is required to build a DMS that provides a coherent

open framework on top of which design tools can be integrated. Our approach in this is best characterized as follows:

- First of all, we clearly distinguish between (local) *tool aspects* and (global) *system aspects*, especially when taking the design data into account. The DMS must be based on the *invariants* that can be recognized here, rather than the features of a particular tool set or design representation.

- A formalized approach to the construction of a conceptual model, reflecting the different types of data and their relationships encountered in VLSI design (a *data schema*), is adopted. Our approach is original in that we import semantic data modeling techniques from the database area, to be used as a formalized tool for data analysis. Based on these techniques a data schema is derived for the *management data*.

- Given this data schema, a DMS is implemented that provides the above mentioned facilities, based on the knowledge it has of relationships that are present within the design data. As we will show, this approach provides a basis for intelligent interaction with the designer, being based on global system concepts rather than detailed tool aspects.

In the next section we briefly introduce a number of data models and motivate our choice of one semantic variant. In section 3 we look at the design data itself and recognize the design object as the basic entity. The data schema is extended in section 4 to comprise the notions of view and hierarchy. In the following section the transaction model for the design objects is described and versioning is incorporated. In section 6 the system architecture is described in more detail and the implementation of the different components is discussed. The interaction with the designer is discussed in section 7. In section 8 we focus on the Data Management Interface (DMI) which takes care of the communication between VLSI design tools and the DMS. Our conclusions are given in section 9.

7.2 *DATA MODELS*

7.2.1 Introduction

A *data model* is a collection of concepts and constructs for expressing the static properties, dynamic properties, and integrity constraints of an application environment [Lyngbaek84].

Given a data model, a *data schema* can be defined, describing the structure and properties of a specific application environment.

A *database* is a data repository containing a possibly large amount of interrelated data, structured according to a corresponding data schema.

Historically, the following four classes of data models can be recognized [Afsarmanesh84]:

- hierarchical data models
- network data models
- relational data models
- semantic data models

The hierarchical, network, and relational data models are frequently referred to as the classical data models [Lyngbaek84, Bic86]. The hierarchical and network models only provide primitive operations, and the user must deal with aspects of the internal organization. The relational data model is a more user oriented data model. However, it is a flat model: the relations are not positioned with respect to each other. As opposed to the hierarchical and network approach, relations do not contain implicit references (pointers). Associations between tuples are exclusively represented by attribute values drawn from a common domain. The use of composed keys does not provide the user with sufficient means to represent all abstractions in a precise way. Furthermore, the definition of various integrity constraints is not an integral part of the modeling process [Bekke83].

7.2.2 *Semantic Data Models*

The classical data models are all *record based.* When modeling an application environment, not all record types in the resulting schema correspond to the complete definition of a particular concept from that environment. That is, they lack semantic expressiveness [Afsarmanesh84, Bic86, Hardwick87].

The *semantic data models* enable the user to better formalize the semantics of his data, and are therefore considered more *user oriented.* Instead of being based on the record model, the semantic data models are *object based;* the application environment is modeled as a collection of interrelated objects, each one corresponding to a concept from this environment.

Attempts to categorize the semantic data models are described in [Lyngbaek84] and [Afsarmanesh84]. For our purpose the data model preferably comprises a methodology for data modeling, permitting us to formalize the semantics of the design data. We concluded that the OTO-D semantic data model [Bekke83] is best suited, as it offers a number of well-defined modeling constructs accompanied by an attractive query language and has a clear way of visualizing the data schemas one defines. OTO-D stands for Object Type Oriented Data model. It can be seen as a follow-up on the semantic hierarchy model, as originally pioneered by Smith and Smith [Smith77]. For a good understanding of the data schemas that will be derived in the following sections, we first give a short introduction to the OTO-D data model.

7.2.3 *The OTO-D Data Model*

7.2.3.1 *Data Definition: the Notion of Type*

The semantic approach is based on the notion of *type.* A type is defined by a certain number of different properties. For example the abstraction:

TYPE student = name, address, department

defines a student as being completely characterized by the properties name, address and department. These properties are called *attributes.* An object

having the properties of a certain type is called an *instance* of that type.

A type definition is a positive statement or *assertion* about the application environment, consisting of a *subject* and a *predicate*. A data schema consists of a number of these type definitions. The subject denotes the new concept and the predicate denotes the collection of known properties by means of which the concept is described. Thus, as opposed to the relational approach, object types are defined in terms of previously defined object types.

OTO-D offers two forms of semantics to formally judge a data schema. The first one is *convertibility*: each object is uniquely characterized by its attribute values. Based on the notion of convertibility the type definitions can be checked for completeness. The second one is *relatability*: an attribute is related to a type with the same name. For example "department" in:

TYPE department = name, head
TYPE student = name, address, department

Relatability automatically fixes referential integrity constraints at the instance level: an attribute value is related to an instance of the type to which the attribute is related.

7.2.3.2 Aggregation and Generalization

OTO-D offers two abstraction primitives to construct a data schema: aggregation and generalization. *Aggregation* is a form of abstraction in which a certain number of different properties is combined to create a new named object. Examples of aggregations were student and department. Together they constitute an *aggregation hierarchy*. *Generalization* is a form of abstraction that relates a type to a more generic one. In knowledge representation research this is known as the IS-A relationship. It is possible that a type appears more than once as an attribute in a type definition, to fulfil different roles. For this purpose OTO-D offers *role attributes*, denoted by a prefix. Examples of the use of role attributes will follow in section 4.

OTO-D offers a clear diagrammatic notation to visualize the relationships among the types of a data schema. The student-department example looks

like:

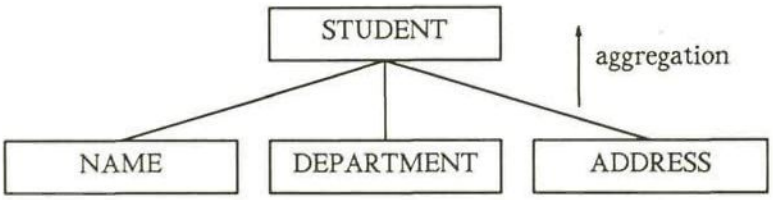


Figure 7.2. Example of an aggregation

Along with these pictures goes a quite strict interpretation. Each attribute relationship is represented by a line that goes from the bottom of the composed type to the top of the attribute type. The referential integrity constraints along these lines always have to be satisfied: an instance of a composed type is existence-dependent on the instances of its attribute types.

7.2.3.3 Data Manipulation Language

The Data Manipulation Language (DML) of OTO-D allows the formulation of nonprocedural, high-level queries and thus separates the user from the internal organization of the data. It offers *selection, extension* and *modification* commands. The most important expression is the selection, the general form of which is:

<type name>	
ITS <attributes>	*property list*
WHERE <condition>	*qualifying predicate*

The ITS construct permits downward traversal along the attribute relationships of the data schema; starting from a composed type we can "look downward" to ITS attributes, ITS attributes ITS attributes, etc. Given an arbitrary schema, the semantic concepts of OTO-D guarantee that all data that can be addressed this way is present (referential integrity) and related in a meaningful way according to the schema. The ITS construct can be used both in the property list to retrieve lower attribute values, and in the qualifying predicate to impose constraints on these lower attribute

values for object selection.

Using the keyword GET, a selection command can be formulated. A typical query on the data schema presented above:

```
GET student
    ITS name, address
    WHERE department ITS name = 'EE'.
```

To modify the contents of the database, three types of modification commands are available: INSERT, DELETE and UPDATE.

7.2.3.4 Conclusion

It has been our conclusion that OTO-D supports the definition of a data schema that correctly reflects the (structural) semantics of an application environment. It offers a number of well-defined modeling constructs, incorporates integrity constraints in the modeling process, offers a clear diagrammatic notation and has a simple, but powerful, data manipulation language. In the following sections we will use OTO-D to derive a data schema for the logical organization of VLSI design data.

7.3 THE DESIGN OBJECT AS THE BASIC ENTITY

7.3.1 The Design Object as the Basic Entity

At first sight, conventional database management systems (DBMS) offer some attractive facilities for the reliable storage of design data, including recovery mechanisms, concurrency control and integrity maintenance. However, most of these DBMSs have been targeted for business applications and do not specifically address the problems encountered in a design environment [Sidle80]. Transactions on a business DBMS typically are short in duration and affect only a small amount of data. Concurrency and recovery strategies have been tuned towards these characteristics. In the design environment on the other hand, the designer requests all the information pertaining to a piece of design to modify it extensively over a long period of time before returning it to the database [Katz83, Buchmann84].

The important issue is that VLSI design applications invariably deal with conceptually *localized collections of related data* which are manipulated as a single entity. This localization needs to be conserved by the design database. In line with several other researchers [Katz83, Batory85] we call these basic entities *design objects*. Examples of design objects are: a piece of layout, a netlist description, etc. The design object must play a dominant role in the organization of the design data within the design database. The arguments for this approach are listed below:

- The design object is the *unit of access*. Design objects are extracted and replaced as a unit. Hence, such issues as *concurrency, recovery* and *versioning* should be handled at the level of the design object.
- As will be shown in section 4, design objects are the nodes of the hierarchical multi-view 'matrix'.
- By taking the design object as the basic entity for further modeling, we hope to construct a coherent DMS framework, without getting involved with representation details of some predetermined view-types.
- To the designer the design object has a well defined meaning: the behavioral description of his ALU, the circuit description of a flip-flop or his new routing result.

7.3.2 The Initial Data Schema

To define the object type *design-object* with OTO-D, we have to examine by which other object types design-object is characterized. First, a design object has a *name* by which it can be identified. Further, in a logically distributed environment each design object has been constructed in connection with a certain *project*. Other attributes of design-object might be its *designer* or the *date* of construction:

TYPE design-object = name, project, designer, date

Thus, a design object is completely characterized by its name, project, designer and date. Name and date can be base types. Designer might be defined separately, using such properties as name, address, department,

salary, etc. The same holds for project, although it might as well be a base type. These details are not important when defining the object type design-object.

In a project oriented environment there is no need for all names of design objects to be globally unique. Therefore, the scope of a design-object ITS name is limited to the design-object ITS project. In fact, a project can be seen as a clustering object containing a number of design objects. The schema models correctly that a design object can only exist in connection with *one* existing project: design-object ITS project. Of course we need mechanisms to *refer* to a design object from other projects, e.g. for hierarchical inclusion. However, the actual description resides in its "home" project, which may be a library. The resulting diagram is given in Figure 7.3.

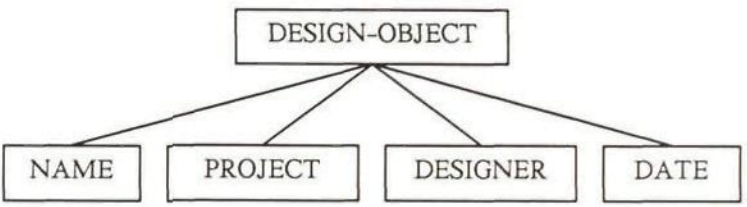


Figure 7.3. Diagram of the definition of the type design-object

The concept of design object provides a profound basis for a consistent system philosophy, not being blurred by view-type dependent issues. The 'real' design data, i.e. the polygons, wires, devices, nets, ports, etc., is concentrated within the atomic design objects. Although access to the design objects is provided by the DMS, it does not interpret their representation details; these are handled by the tools. Instead, the DMS maintains *information about the design objects* to provide the facilities described in section one. This data is called the *meta design data.*

The meta design data describes how the design objects, viewed as atoms, are related to each other. As we will see in the following sections the

hierarchical and equivalence information can be found at this level. The version information of an object, verification statuses, locking information, ownership etc. are also considered meta design data, being maintained by the DMS.

We can compare the way we look at a design object with the way a librarian looks at a book: an object characterized by e.g. a title, author and date as the abstraction of its contents about which no further knowledge is required. An important distinction from a public library, however, is that in the design environment many relationships may exist between the individual "books" (design objects), while the "books" are also subject to evolution.

7.4 HIERARCHY AND MULTIPLE VIEW-TYPES

7.4.1 Hierarchy

The most obvious way to handle the inherent complexity of VLSI design is by structuring the design in a hierarchical way: decompose a circuit into several smaller subcircuits, preferably with well defined interfaces and limited mutual interaction, which in turn can be decomposed into even smaller sub-subcircuits, etc [Niessen83]. The advantages are clear: the subcircuits can be constructed and verified independently, and once verified they need not be verified again when changes occur in either higher circuits or neighboring subcircuits; only the interaction needs to be verified.

The structure of an hierarchical VLSI design can be typified as a *directed acyclic graph*, with the design objects being the vertices and the hierarchical relationships being the edges connecting these vertices. These hierarchical relationships are the only composition information about the design that is known to the DMS. A design object is constructed using zero or more component- or *son*-design objects, and can be used as a component by zero or more composite- or *father*-design objects. It is even allowed that a father-design object refers more than one time to the same son-design object. The leaf nodes are design objects that do not refer to component design objects; they are just convenient collections of primitive description

elements.

If a design object is used more than once as a component, the corresponding hierarchical relationships are just references to a common definition. Attached to the hierarchical relationship is the information unique to the son-object's *instance*, i.e. how the son-object is actually used within the father-object. This composition information is called the *constructor*. It can be used by design tools to compute the actual instances, thereby instantiating the directed acyclic graph into a tree of instances. The details of the constructor differ according to the view-type. For instance, in the layout view the compositions are described by geometric transformations. The constructor generally contains *repetition* data to describe *regular structures* in an efficient way.

An *instance name* can be used as an additional selector to identify the various instances within the same father-object.

The connection between the object types *design-object*, *hierarchical-relationship* (having a one-to-one correspondence with *instance)*, *constructor* and *instance-name* is defined as follows:

TYPE hierarchical-rel (or instance) = instance-name,
father_design-object, son_design-object, constructor

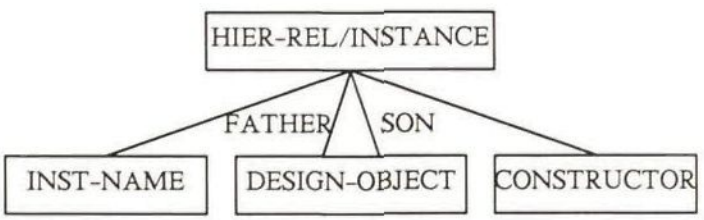


Figure 7.4. Modeling the hierarchical relationships between design objects

All information concerning an instance is aggregated into a private entity; an *instance* is an object that can be derived from a *son-design object* using a *constructor*, to be used in a *father-design object* with an additional selector

instance-name. The two design-object attributes each have their own role (role attributes), denoted by the prefixes *father* and *son*. The schema correctly reflects that a design object can be involved in zero or more hierarchical relationships, both as a father- and as a son-design object. Multiple hierarchical relationships may exist between a father- and a son-design object, distinguished by their instance names and constructors. Relatability guarantees that a hierarchical relationship can not exist without both a father- and a son-design object.

The schema allows the DMS to administer which design objects are used where, and to make this information available to the tools and the designer. Given a father-object, the son-objects can be retrieved, but in a similar way the father-objects can be retrieved given a son-object. For example, the son-objects of a design object named 'ALU' can be retrieved by the simple query:

```
GET hier-rel / instance
    ITS inst-name, son_design-object ITS name
    WHERE father_design-object ITS name = 'ALU'
```

With a similar query we can also retrieve the father-objects that use this 'ALU':

```
GET hier-rel / instance
    ITS father_design-object ITS name,
    father_design-object ITS designer
    WHERE son_design-object ITS name = 'ALU'
```

7.4.2 *View-types*

The second important way to handle the inherent complexity of VLSI design is to support several *levels of abstraction* at which a design can be represented. The different abstractions of a design are called the *views* of that particular design. Each view describes the design to a certain extent, omitting details that are irrelevant to that specific level of abstraction. Well known *view-types* are layout, circuit, logic and functional.

It is not up to the DMS to determine which view-types are supported. New design methodologies, and their associated representations, are still evolving. Hence, the DMS must provide support for organizing multiple views of a design, without it understanding their detailed structure. In this line of thought, a design object is a representation of a design at *some* level of abstraction. Consequently, we have to extend the type *design-object* with the classifying attribute *view-type*.

TYPE design-object = name, view-type, project, designer, date

According to this type definition each design object is of *one* view-type. This classification logically partitions the complete set of design objects into a number of subsets, one for each view-type. For instance: the layouts. Retrieval of such a subset can be done by a simple query:

GET design-object
ITS name, designer, date
WHERE view-type = 'layout'

The scope of the name of the design object is limited from the project to the view-type. A system-wide identification of a design object is given by the triplet (project, view-type, name).

An interesting aspect of the last modeling step is that we did *not* introduce some grouping object, e.g. *design*, to which all views of a piece of design are related. This could have been done by introducing the type *design* as an intermediate level in our data schema:

TYPE design = name, project
TYPE design-object = design, view-type, designer, date

However, in the multiview context we want each design object to carry ITS own name, instead of attaching the name to a more generic grouping object. This name can then be assigned or changed independent of the names of design objects of other view-types. Retrieval of all design objects that carry the same name is in our approach just one of many simple queries that can be issued:

```
GET design-object
    ITS view-type, designer, date
    WHERE name = 'flipflop'
    AND project = 'our_project'
```

The reason for this approach is, that we do not want to suggest implicit relationships between objects of different view-types that carry the same name. Such implicit relationships are not desirable in an environment where the design objects are constantly evolving and consistency maintenance across relationships is of primary importance. Secondly, we want a more flexible mechanism for administering these relationships. For instance, it must allow more than one layout to be related to the same functional description.

7.4.3 *Equivalence Relationships*

We say that two design objects are *equivalent* if *certain aspects* of both descriptions have shown to be *identical,* for instance, by deriving one representation from the other in the synthesis or verification process. The DMS must provide the mechanisms to store and maintain the *equivalence relationships* between the original and the derived design objects. For this purpose we define the object type *equivalence-rel* as follows:

```
TYPE equivalence-rel = original_design-object, derived_design-object,
                                                    tool, constructor
```

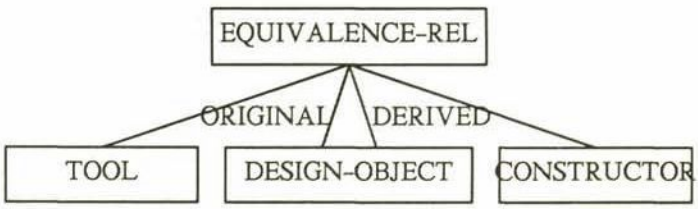


Figure 7.5. Modeling the equivalence relationships between design objects

Tool is the tool that derived the relationship and *constructor* is some information used for this operation (e.g. parameter values). According to

this schema the DMS supports equivalence relationships that have been established between pairs of design objects, without any premature assumption about their nature. A design object can be involved in zero or more equivalence relationships; many-to-many relationships are covered by this schema. It also models correctly that an equivalence relationship can only exist if both design objects exist. When one of them is removed the semantic integrities require the removal of the equivalence.

The following example shows how we can retrieve the "extracted circuit(s) of a layout named flipflop":

```
GET equivalence-rel
    ITS derived_design-object ITS . . . .
    WHERE tool = 'extractor'
    AND derived_design-object ITS view-type = 'circuit'
    AND original_design-object ITS view-type = 'layout'
    AND original_design-object ITS name = 'flipflop'
```

7.4.4 The Interplay between Hierarchy and Equivalence

It is generally acknowledged that the constraint of identical hierarchical decompositions across all view-types yields an unacceptable inflexibility [Katz83a]. In our approach the orthogonal concepts of hierarchy and equivalence have not been intertwined; there are no mutual dependencies between the hierarchical and equivalence relationships in the schema.

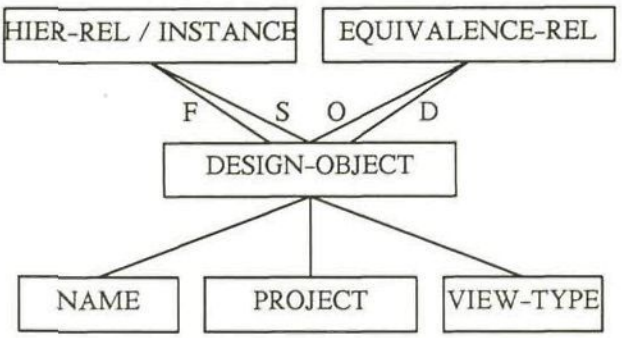


Figure 7.6. The hierarchical multiview framework

According to our data schema, Figure 7.7 gives a typical example of the structure of a design:

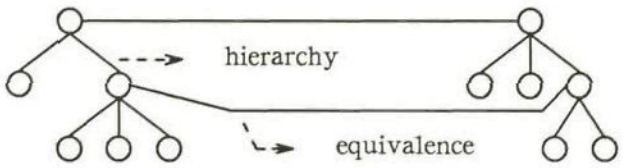


Figure 7.7. Typical structure of a design

Design objects are related as atomic objects by either 'vertical' hierarchical relationships or 'horizontal' equivalence relationships. This is called the *hierarchy multiview matrix* [Dewilde86]. Structure, where it exists, can be exploited without constraining the designer. Browse tools should interactively visualize this hierarchical multiview structure to the designer, thereby encouraging him to follow a structured design methodology, exploiting the principles of view and hierarchy.

7.5 DESIGN EVOLUTION

7.5.1 Design Transactions and Concurrency Control

7.5.1.1 The Transaction Model

In conventional database technology, the notion of *transaction* plays an important role. A transaction is a collection of database operations that are either executed completely or not at all. It is the atomic unit of consistency and recovery. We define a *design transaction* as a transaction in which a (single) design object is involved. Examples are an edit session or some verification on a design object. In section 3 we already mentioned that design transactions exhibit different characteristics than transactions on business DBMSs. They typically are of long duration and involve large amounts of data.

To support design transactions we adopt the transaction model described by Lorie [Lorie83]. A design object is CheckedOut from a shared database to a so called private database, while flagging the object by a *lock*. A CheckIn can be issued by the private system to return the design object to the shared database, thereby removing the lock. A design transaction is the period of time from a CheckOut to the corresponding CheckIn.

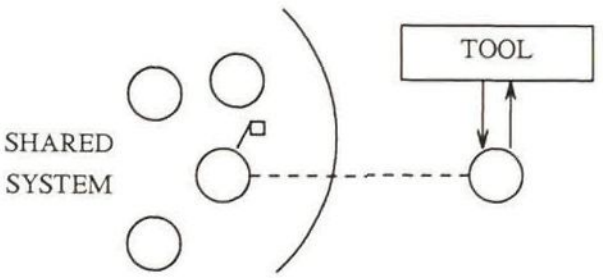


Figure 7.8. A CheckedOut design object

Adopting this model, *recovery* issues can be handled quite easily. Updates are not done in place, so when a tool is being aborted, or worse, in case of a system crash, the design transaction can easily be rewinded by returning to the last saved copy.

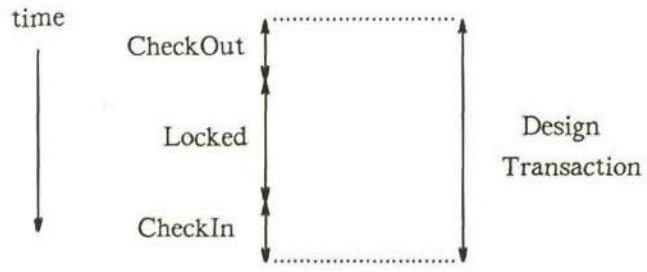


Figure 7.9. Locking between CheckOut and CheckIn

The concurrency control strategy is very simple: a complete design object is locked by creating a *design lock* entity during the CheckOut transaction on the shared system. Once the design object has been CheckedOut successfully, unconstrained access to all the requested representation details of the object is allowed to the tool that initiated the transaction.

Tools that issue a CheckOut request on an object that has already been CheckedOut will hit a lock. On this occasion we do not simply refuse the new request, as even at the level of design object some concurrency has to be offered, e.g. to allow multiple read-only transactions. A tool can CheckOut a design object with one of several modes. If the design object has already been locked by another tool, this mode is compared with the lock mode to see if they conflict. If so, the designer is notified and allowed to proceed with other, non-conflicting design activities. Thus, the design lock has a conceptual meaning to the designer, as opposed to locks in business DBMSs which are pure implementation matters.

General rules can be applied to decide on the modes. For instance, "update is exclusive, concurrent read-onlys are allowed". To increase concurrency, new modes accompanied by refined rules can be introduced. Ultimately, the CheckOut operation might consult some rule base containing tool specific compatibility information. For instance, we may define under which circumstances a DRC and LtoC-extractor are permitted to run concurrently on the same layout object, even though they both might attach derived

information to it.

7.5.1.2 Incorporating the Design Lock in the Data Schema
We represent the semantics of the design lock as follows:

TYPE design-lock = design-object, lock-mode, date, tool, designer

A *design-lock* is characterized by the *design-object* that is locked with *lock-mode* by *tool* and *designer* at *date*. More than one design lock may be related to one design object, each with its own lock-mode, date, tool etc.

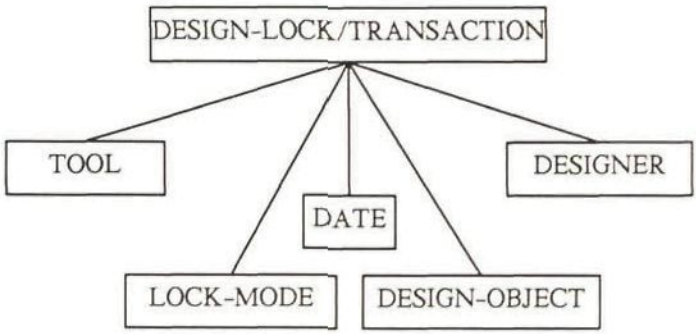


Figure 7.10. Design-lock as the aggregation of its relevant properties

Because there is a one-to-one correspondence between a design lock and a design transaction, we identify the design-lock object type with a design-transaction object type. This type aggregates a design object, a tool, a designer, a date and a lock mode into a meaningful entity: a design transaction.

Not only the in-progress transactions are administered this way. Based on this data schema we record *all* transactions in which a design object has been involved, switching the mode to *completed* or *failure* upon CheckIn. In this way a complete update and verification *history* is maintained for the designer.

Simple queries permit the retrieval of information about transactions that have been issued on a certain design object, by a certain designer, with a certain tool or mode, since a certain date, etc. For instance:

```
GET design-lock / transaction
    ITS design-object ITS name, lock-mode
    WHERE designer = 'Gilmour'
    AND tool = 'DRC'
    AND date > 871103
```

7.5.1.3 State Management: Towards Powerful Design Management

As was demonstrated by a prototype implementation of a *state manager* [Willems87], powerful design management facilities can be offered by exploiting the transaction information. This state manager relies on two types of information to perform its tasks.

Firstly, it derives *state descriptions* for the individual design objects, by observing the transactions that are performed on them. In fact, the state of a design object is represented by its transaction history, which has already been modeled in the data schema (Figure 7.10). The transaction history reflects "what" has been done "when" to the object, and can be exploited by the state manager.

Secondly, the state manager consults an external rule base containing information about the transactions (state transitions) that tools may execute and their dependencies. Obviously, this information depends on the particular tool set at hand and must be extended when a new tool is added to the system. However, this knowledge is well localized within an external description, instead of being contained in the code of the DMS. Moreover, default cases allow tools to be added without even modifying the rule base.

The prototype state manager uses the information about the states and state transitions to perform the following tasks:

- *Maintain state descriptions for the individual design objects*: The design transactions that are executed by the tools are identified and

administered to maintain an overview of the states of the design objects. The state manager takes into account that new transactions may invalidate previous ones. The dependency information from the rule base is applied to compress the list of valid transactions.

- *Check start conditions of transactions*: The tool-specific start condition of the requested transaction, which is retrieved from the rule base, is checked against the state of the design object upon CheckOut. The state manager decides whether the intended design transaction is allowed for the particular object.

- *Automatic tool activation*: If the design object is not in the proper state upon CheckOut, the state manager tries to bring it into that state. It consults the dependency information from the rule base to see if it can run preprocessor-tool(s) that bring the object into the proper state. If this seems possible, it asks the designer for a confirmation and, if confirmed, runs the tool(s). After successful completion, the state manager reconsiders the original CheckOut request, which will then be granted.

The advantages of this state management facility are clear. The designer does not have to worry any more whether certain mandatory preprocessing steps have already been performed and are still valid, before applying the next design tool. In the future we can even make the dependency information from the rule base available to the individual designer. This will allow him to tune the design system to his personal design style, by adding the proper dependencies. For instance, he might like to have valid DRC results, before starting a LtoC-extraction on a layout.

The state manager has been written in Objective-C [Cox86]. This is a hybrid language that contains all of the C programming language plus a number of extensions for object oriented programming. An important aspect of the prototype implementation is that the state management is completely hidden from the tools. As will be described in section 8, the tools interact with the DMS via the Data Management Interface (DMI). Not

a single extension / modification had to be made to this DMI, and hence to the tools, to incorporate the state manager into the DMS.

7.5.2 *Versioning*

The VLSI design process is both iterative and tentative. While incremental changes are being applied, a design typically evolves through several *versions*. The DMS must support the designer in maintaining multiple versions of his design. As an additional advantage, versioning permits flexible concurrency control and consistency propagation as multiple versions are allowed to coexist; previous versions can be browsed without regard to in-progress update transactions on experimental versions that still have to be verified [Leuken85].

We call a collection of related versions a *module*. The relationship between module and design-object is a one-to-many dependency: a module consists of several design objects (versions), but each design object belongs to one module. This is represented by the following refinement of the data schema:

TYPE module = name, view-type, project
TYPE design-object = module, V#, V-status, designer, date

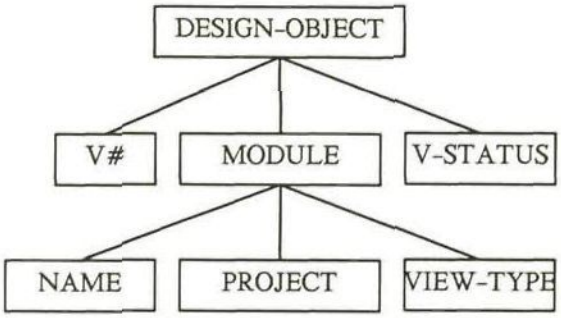


Figure 7.11. Module as a single-view version chain

We introduced the type *module* as a grouping object to which all versions having the same name, view-type and project are related; it is a "family" of

versions. According to the schema, a design object does no longer carry its own name; the name is attached to the more generic module object, to be shared as design-object ITS module ITS name by the versions of the module. The alternative would have been to let each design object (version) carry its own name and simply extend the type design-object with additional attributes V# and V-status. However, we prefer to attach the name to a separate module object, as it corresponds to the designer's idea of a "version chain". This is a piece of design under construction to which a name can be assigned, whose name can be changed, etc. A system-wide identification of a module is given by the triplet (project, view-type, name). Identification of an individual design object requires an additional *version number* (*V#*).

Simply maintaining a collection of numbered versions per module does not really help the designer. Some mechanism has to be imposed on the version propagation, to permit the version selection to be automated. For this purpose, an attribute *V-status* has been added to design-object. A version mechanism that exploits this version status in an attractive way has been described in [Leuken85] and [Wolf86]. We will not present this version mechanism in detail here, but confine ourselves to listing its most important features:

- Rules for default version selection upon CheckOut can be applied, taking tool and CheckOut-mode into account.
- Flexible concurrency control in an environment of hierarchically related design objects.
- A flexible mechanism for consistency propagation in the hierarchical multiview context.

7.6 *SYSTEM ARCHITECTURE*

7.6.1 *Introduction*

Summarizing the aspects that have been covered in the previous sections, we conclude:

- The DMS operates as a librarian with the *design objects* as its basic entities.
- A *transaction model* for these design objects has been defined.
- The DMS maintains an administration on top of the design objects: the *meta design data.*
- This meta design data has been modeled according to the following *data schema:*

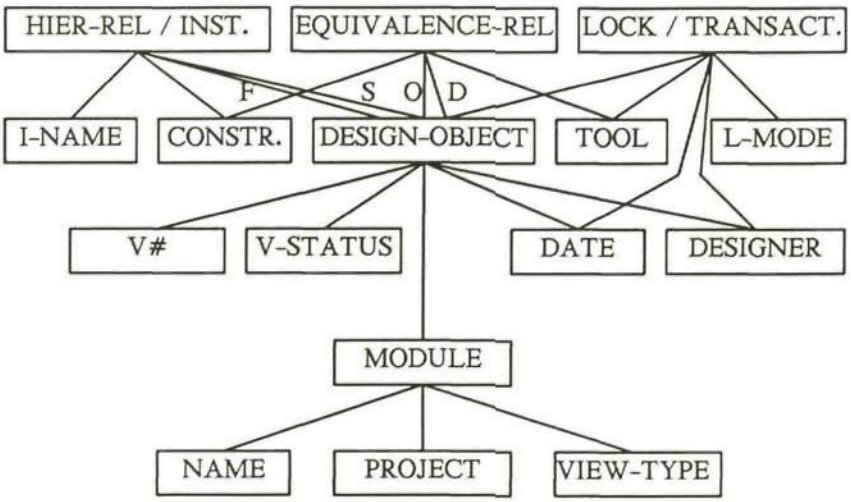


Figure 7.12. The data schema

The data schema defines how the various objects and their relationships are logically organized, including various integrity constraints. It provides the basis for the construction of an intelligent DMS, that makes the information about the structure and status of the design available to both the tools and the designer. Various example queries already demonstrated that a great variety of information can be retrieved from a DMS that is based on this schema.

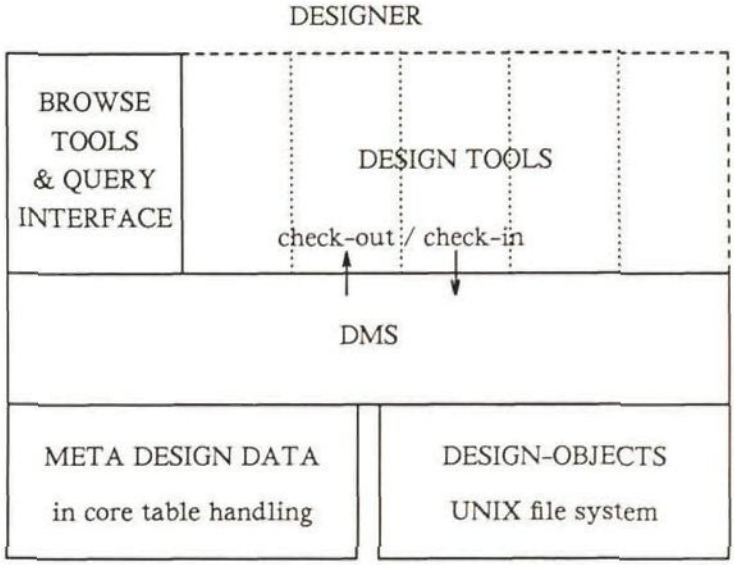


Figure 7.13. System architecture

Starting from these points, we refine Figure 7.1, to come to the system architecture depicted in Figure 7.13. The distinction between the detailed design data, contained in the design objects, and the meta design data is reflected by the use of different storage components. The design objects are simply mapped to a distributed hierarchical file system, with a small layer on top of it to guarantee atomic updates.

The properties of the meta design data are quite different. The meta design data of a project typically is *small* in size when compared to the volume of the corresponding detailed design data. Also, *ad-hoc queries* for small amounts of data are issued *frequently* by the DMS and the (browse) tools. These are primarily read accesses. The meta design data is designer-oriented: browse and query facilities must allow the designer to consult the meta design data. A dedicated storage module has been developed for the meta design data, to deal with these aspects. See section 6.2.

The actual DMS, i.e. the "operational facilities", has been built on top of these storage components. While handling the design transactions via its Data Management Interface (DMI), it consults and updates its meta design data. Important modules of the DMS are the version handler, concurrency controller and state manager. The version handler identifies the design object upon CheckOut, assigns a V# and V-status upon CheckIn and checks / propagates consistency across relationships upon installation of a new version in an existing design hierarchy. As described in 5.1, the concurrency controller decides on concurrency matters upon CheckOut. The state manager inspects the state of the design object upon CheckOut and administers the transaction upon CheckIn. See section 5.1.

Among the tools on top of the DMS we can distinguish the *design tools*, e.g. editors, checkers and simulators, and the *browse tools*. The browse tools only access meta design data and do not access design objects. Both types of tools interface to the DMS via the same DMI.

7.6.2 *The Storage Module for the Meta Design Data*

7.6.2.1 *Meta Design Data Access*

As we mentioned in the introduction of this section, a dedicated storage module has been developed for the meta design data. To its clients, i.e. (browse) tools and the DMS itself, it appears as an implementation of the OTO-D data model. Via its programming interface, queries can be issued to store or retrieve information that is structured according to the data schema. The queries can simply be passed as strings, according to the OTO-D Data Manipulation Language (DML).

The data schema has not been hard-wired in this storage module, as we do not assume that the schema of Figure 7.12 represents the "fixed-for-all-times" data schema in full detail. Instead, the storage module reads its schema from an external description at startup. This allows our DMS to evolve without extensive programming efforts at its lower levels.

A more or less classical transaction facility permits clients to issue multiple queries as one transaction, that is, without interference from other clients. This facility permits certain DMS-operations, e.g. CheckOut or CheckIn, to be implemented as atomic actions. These low level transactions should not be confused with the high level design transactions as presented in section 5.1. The low level transactions are restricted to access to the meta design data and are assumed to be of short duration. In fact these transactions are used to implement the long design transactions on design objects.

7.6.2.2 Implementation

The implementation of the storage module consists of a query layer on top of a generic table handler. The table handler operates as a storage module for tabularly structured data. The query layer on top of it takes care of query interpretation and query resolution. It maps each composed type of the data schema to one or more tables.

To offer a transaction facility, concurrency control and recovery mechanisms have to be employed. Concurrent access to the meta design data must be controlled to preserve its consistency, in particular when multiple queries are issued as one transaction. Recovery facilities have to be such that results of committed transactions are persistent across hard- or soft crashes, and results of incomplete transactions can always be rewinded.

The table handler provides basic concurrency control and recovery mechanisms. Access to the tables is controlled by an open / close strategy: They must be opened before their contents can be accessed to resolve a query request. The tables can be opened for "write", which is exclusive, and for "read-only", which is non-exclusive with respect to other read-onlys. Multiple tables can be opened at once. The correct operation of the open / close strategy is guaranteed by semaphores supplied by the UNIX operating system. The query layer implements its transaction strategy on top of the open / close strategy of the table handler. All tables that are required to perform the transaction are opened at the start of the transaction and closed when the operations have been finished, thereby committing the transaction.

As all requested tables are opened at once no deadlocks can occur.

Special attention has to be paid to the communication mechanisms that are employed internally by the storage module, as a number of requirements for the meta design data access have to be satisfied:

- *Fast access to the meta design data.*
  The potentially large number of accesses by DMS and tools must be handled efficiently.
- *Sharing of the meta design data.*
  The module must offer high accessibility in a concurrent environment.
- *Local as well as remote data communication.*
  The module has to operate in a distributed environment.

Our solution (Figure 7.14) is a combination of the use of shared memory techniques and the use of a server process per project that responds to messages that appear in its message queue.

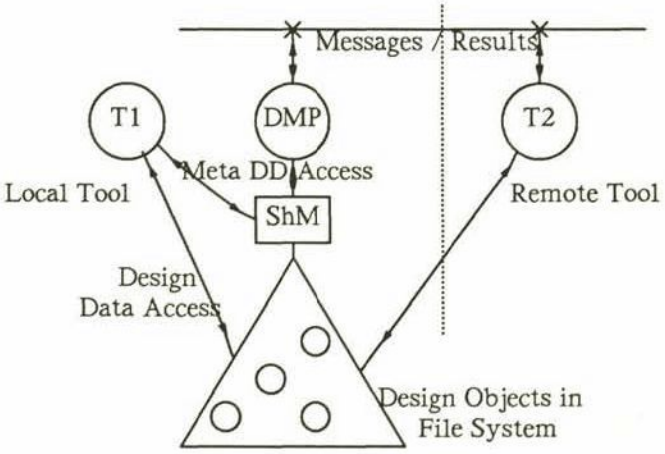


Figure 7.14. Mechanisms for (meta) design data access

When a project becomes "active", a Data Management Process (DMP) is started which maps the meta design data of this project into shared memory (UNIX system V and BSD 4.3). Tool processes that run on the same machine as the project and its DMP can attach this shared memory to their data space. That is, several processes can share data directly, by having some pages of their virtual memory referring to the same physical memory. Queries from these local processes are executed *efficiently* by functions that have direct access to the meta design data in shared memory.

If a tool process runs on a different machine than the project with its DMP, its access functions can not directly access the meta design data in shared memory. In this case, the functions send their requests to the appropriate DMP [Birrell84]. At the side of the DMP, the requests from the remote processes appear in a message queue. The DMP executes the requests for the remote processes and sends the result back to them.

The physical distribution of the (meta) design data is made transparent to the higher layers of the system, by hiding the local / remote selection in the "low level" access functions. These functions also have the possibility to defer access, for example when the requested part of the meta design data is being updated; a short wait may occur when a table has already been opened in a conflicting mode by another process. As only short transactions occur at this level, such a split-second delay is felt to be a minor inconvenience.

An interesting efficiency aspect in this concurrent environment results from the distinction that we make between the detailed design data contained in the design objects and the meta design data on top of it. Concurrency issues concerning the detailed design data are resolved at the meta design data level upon CheckOut (section 5.1). Once access to a design object has been granted, a tool can freely dive into the corresponding design data. Access functions permit efficient retrieval (storage) of the detailed design data from the file(s) in which it is contained, without the overhead for concurrency control. This is particularly important as relatively large amounts of data have to be handled at this level.

7.7 THE DATA MANAGEMENT BROWSER

The Data Management Browser (DMB) enables the designer to browse through the meta design data, which is maintained by the DMS. The designer can start a DMB and open one of his projects. The DMB tries to establish communication with the particular project and opens a window on the screen. After a successful connection has been established, the designer has access to the meta design data of that project. The browse facilities fall apart in two categories: a generic query interface and dedicated browse facilities. The query interface is generic in that it can handle arbitrary data schemas. The implementation of the dedicated browse facilities, on the other hand, depends on the data schema at hand.

Via the query interface the designer can issue OTO-D query requests. While deriving the schema of Figure 7.12, we already presented various queries to illustrate the use of the OTO-D DML for information retrieval. However, we have not yet discussed how the interaction with the designer actually proceeds. At the moment the DMB accepts the queries in textual form, passes them directly to the storage module for the meta design data and displays the responses. For instance, with a simple query the designer can ask the DMB to list all modules in the particular project. A more sophisticated type of browsing is offered by supporting wildcards. For example,

```
GET module
    ITS name, view-type
    WHERE name = 'F*'
```

returns the names and view-types of the modules whose name starts with a "F":

module	name	view-type
m100	FlipFlop	layout
m118	FullAdd	layout
m120	FlipFlop	circuit
m133	FlipFlop1	circuit
m149	FullAdder	logic

At the moment we are working towards an implementation of the DMB that lets the designer enter his queries in a more convenient way. It graphically displays the data schema together with some menus containing, among other things, the OTO-D keywords. With this facility the designer can interactively compose correct query requests by pointing with his mouse at the proper keywords, types and attribute relationships.

Dedicated browse facilities allow the designer to inspect particular aspects of his design in an attractive way. For instance, browse tools have been constructed to interactively visualize the hierarchical multiview structure, to inspect the version chains, or to display the transaction histories of the design objects. They exploit graphical facilities, pop-up menus, etc. to present graph-like representations of the structure of the design, which can easily be browsed by the designer.

The DMB has been written using a distributed windowing system: X-windows. The X-windows system [Gettys86] is the only windowing system that has this functionality and is a well accepted standard.

7.8 THE DATA MANAGEMENT INTERFACE (DMI)

7.8.1 Introduction

The tools communicate with the DMS via the *Data Management Interface* (DMI) [Meijs87]. This is a set of library functions that can be used by the tool developer, without the necessity to have a detailed understanding of the implementation of the DMS. By means of this DMI the tools obtain access to the design data, while taking advantage of the facilities that are provided

by the DMS.

With our data modeling activities we have been aiming at a DMS that is based on the invariants in VLSI design rather than the features of a particular tool set or design representation. Although this permits our DMS to be relatively stable over time, we have to be aware of system evolution. The implementation of all required DMS facilities is a huge task, which implies iteration over several "releases". Secondly, we do not pretend to have a complete view of all DM-issues at this moment. We must allow new facilities to be added and existing ones to be modified whenever necessary.

Consequently, the tools should depend as little as possible on the DMS to avoid extensive tool modifications with each new release of the DMS. Furthermore, the DMS must be open-ended: It should be easy to add new tools to the system in such a way that they become a consistent part of the design environment. Thus, what we need is a DMI that somehow *decouples* the software development and evolution of the DMS on the one hand and the tools on the other hand. We will introduce such a DMI, based on a transaction schema that formalizes the procedural aspects of the communication between the tools and the DMS.

7.8.2 *DMI Requirements*

The following requirements have to be satisfied by a DMI:

- The DMI must bring about efficient interaction between the tools and the DMS. For instance, efficiency partially results from choices that have been made in the previous sections, such as the distinction between meta design data and the design objects. The DMI should somehow adhere to these choices.
- The DMI should be independent of specific tool features or design methodologies. For example, the DMI must not prescribe data formats. It should be universal, to result in an *open-ended* design system where the DMS acts as a free-for-all public repository that can communicate with any type of tool and environment.

- The DMI should be independent of specific features of a DMS. For example, it must allow interfacing to DMSs with or without versioning, concurrency control, multiple view-types, etc. When this requirement is met, the tools can actually be "plugged in" in the same way in different DMSs.

In summary, a DMI should offer some degrees of freedom, but at the same time the necessary discipline to facilitate software evolution and exchanges. Our opinion on how to introduce this discipline is expressed most concisely by the following thesis:

Thesis

The optimal way to decouple the development and evolution of the DMS and the tools is to agree on a common *transaction schema*, and reflect this in the definition of the DMI.

A transaction schema (not to be confused with a data schema) is a procedure according to which the tools obtain access to the design data. Our transaction schema is based on a number of assumptions that we believe to be completely general within the context of chip design.

We assume that the design data is organized on a *per project* basis. A *project* offers the designer a local context in which a collection of *design objects* is present. As elucidated before, a design object describes a functional part of an integrated circuit in terms of certain primitives as well as references to other design objects, and is the appropriate unit of (exclusive) access for manipulation of the design data. Within a design object the actual design data is organized as a set of *streams*, but no assumptions are made on the contents of these streams.

The agreement on these assumptions permits the definition of a transaction schema, and hence a DMI, that localizes the interaction between the tools and the DMS. Any tool modifications that are required to adapt the tool to other implementations of a DMS will then be *strictly local* and will not alter the *structure* of the program. Consequently, they can be done with much

less effort.

7.8.3 DMI Transaction Schema

As a consequence of the recognition of project, design object and stream as units of access, the transaction schema is a layered one. The effect of a tool on a design environment is called a *tool-execution*. It is a (possibly interleaved) sequence of *project transactions* bracketed by an *initialization* and a *termination*. Similarly, a project transaction is a (possibly interleaved) sequence of *design transactions*, i.e. transactions on design objects (section 5.1), bracketed by an *open project* and a *close project*. A design transaction is a (possibly interleaved) sequence of *design data transactions* bracketed by a *CheckOut* and a *CheckIn*, while a design data transaction is a sequence of *design data IO operations* bracketed by an *open stream* and a *close stream*. A design data IO operation is either a *read operation* or a *write operation*.

We present these definitions graphically in Figure 7.15. The boxes on one level represent a sequence of actions, executed from the left to the right. Child boxes specify a refinement of the father action. A starred box represents an iteration and boxes with a small circle imply alternatives. This diagram is a variation on a *Entity-Action diagram* as defined in [Jackson83].

7.8.4 The DMI

7.8.4.1 Concepts

Basically, there is one access function in the DMI for each leaf of the tree in Figure 7.15. Access to either the design environment, a project, a design object or a stream can be obtained by executing the corresponding opening bracket function, as represented by the leafs at the left-hand side of the tree. A transaction is terminated by executing the corresponding closing bracket function at the right-hand side. In between, lower level transactions can be performed.

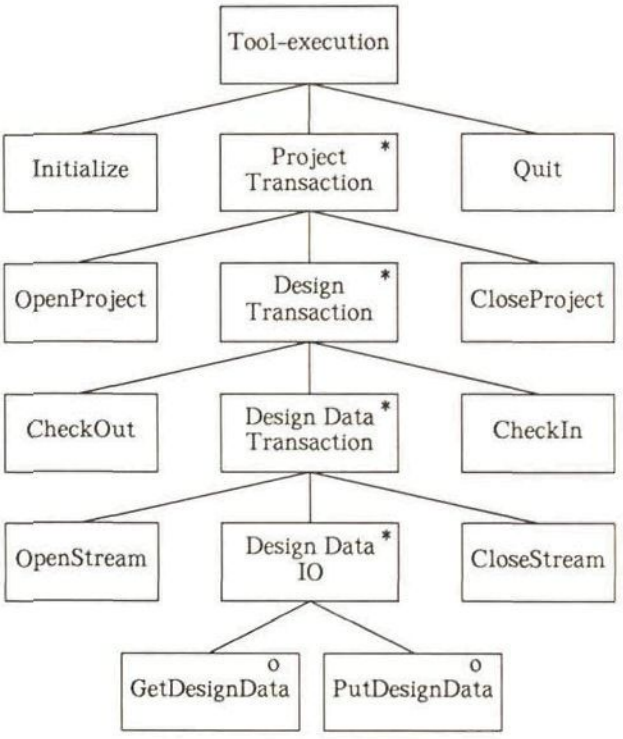


Figure 7.15. Transaction Schema

The functions in the DMI co-operate with each other in such a way that access proceeds in accordance to the transaction schema. They communicate by means of *abstract data types*, called *keys*. The contents of these keys is not fully specified in the DMI definition, but can depend on the particular DMS at hand. There are four types of keys, one for each layer. The key returned by an opening bracket function is part of the argument list of the functions at the next lower level and of the closing bracket function. This allows the interleaving of more than one sequence of calls of lower level functions. The closing bracket function invalidates the key.

After verifying and establishing access, the opening bracket function stores appropriate information in the key that is returned, for further use by the lower level functions. As a direct advantage, the visibility of particular features of the DMS is confined to a small number of places in the DMI. Depending on the particular DMS at hand, the key may contain, for instance, information about the physical location, access permissions and state of the object for which access was obtained. Each key contains a pointer to its "parent key", so that the complete context is known at the lowest levels. Also, all keys with the same parent key are linked together in a list that is attached to this parent key. This facilitates error recovery and automatic clean-up actions.

The opening bracket functions take as arguments, apart from a parent key, an identification of the object for which access is to be obtained and an access mode. In an actual implementation these arguments may reflect certain features of the DMS. In section 5.1 we already saw that the DMS takes the CheckOut mode into account to decide on concurrency matters. As another example: when the DMS provides versioning facilities, the parameters of the *dmCheckOut* function must somehow allow identification of the version to be CheckedOut. Including browse capabilities for object selection in the opening bracket functions may, however, hide such DMS-specific features from the tools.

In an actual implementation of a DMS, appropriate actions will be associated with each function of the DMI. By introducing the right levels of intervention, the DMI provides a natural and universal framework to localize these actions. In the next subsection, the functions at the different layers are presented, together with some examples that illustrate how particular DMS features can be embedded in the DMI.

7.8.4.2 The DMI functions

7.8.4.2.1 Global initialization and termination

Two functions are needed for global initialization and termination. They establish and release contact between the tool and the design environment.

- dmInit (toolname): envkey
- dmQuit (envkey)

DmInit is the opening bracket function of a tool-execution and returns an environment key. This key contains information about the design environment (for example hostname, user-id, process-id, working directory etc.) in which the tool is executed. The tool identifies itself by means of the argument toolname. An action that might be performed by the DMS is to consult a tool database to obtain more detailed information about the tool.

DmQuit is the closing bracket of a tool-execution. It takes care of the necessary clean up operations.

7.8.4.2.2 Project transaction layer

At this level such aspects as projects, libraries and distributed databases can be handled.

- dmOpenProject (envkey, projid, openprojmode): projectkey
- dmCloseProject (projectkey, closeprojmode)

DmOpenProject initiates a project transaction and returns a project key. This key contains information about the particular project, represented by projid, and the access mode, represented by openprojmode. Actions that might be performed are verification of the access rights, retrieval of technology information, setting up network connections or contacting a Data Management Process.

DmCloseProject terminates the project transaction. The details of this operation are specified by closeprojmode. In a physically distributed environment, actions to be performed might include returning local copies, closing network connections, etc.

7.8.4.2.3 Design transaction layer

The functions at this layer take care of aspects of design transactions. To mention are concurrency control, versioning, view-types, maintenance of verification statuses and equivalence relationships, etc. See also section 5.

- dmCheckOut (projectkey, designobjectid, checkoutmode): designobjectkey
- dmCheckIn (designobjectkey, checkinmode)

DmCheckOut is the opening bracket function of a design transaction. Its arguments are a project key, identifying the particular project for which access rights have been obtained by *dmOpenProject*, and an identification of a design object, denoted by designobjectid. The checkoutmode parameter specifies what type of interaction is to take place, so that the DMS can anticipate on it. This mode may, for example, be used by the concurrency controller, as explained in section 5.1. The version handler may take the mode into account for selection of a particular version, e.g. the object having V-status "working" is by default selected for update transactions.

DmCheckIn is called to terminate a design transaction initiated by *dmCheckOut*. Designobjectkey has been obtained from *dmCheckOut*. Checkinmode specifies how the transaction has to be terminated, e.g. whether the transaction should commit or rewind. Actions that might be performed include removal of locks, updating of verification statuses, deletion of scratch data that was created for recovery purposes, etc.

7.8.4.2.4 Design-data transaction layer

These functions are at the lowest level, i.e. closest to the physical IO. They map the design data to and from the physical storage structure being exploited.

- dmOpenStream (designobjectkey, strname, iomode): streamkey
- dmCloseStream (streamkey, closestreammode)
- dmGetDesignData (streamkey, format, arguments)
- dmPutDesignData (streamkey, format, arguments)

DmOpenStream returns a stream key, which is used by *dmGetDesignData*, *dmPutDesignData* and *dmCloseStream*. Strname identifies a stream of data belonging to the design object identified by designobjectkey. Iomode specifies the mode of access to the data, e.g. read-only or write. *DmOpenStream* can check this mode against the CheckOut mode. For example, it should be forbidden to open a stream for write if the CheckOut mode was read-only. A stream can simply be implemented as a file, but

this need not be the case. For example, experiments have shown that it is possible to transparently map IO operations onto shared memory as an efficient channel for inter-tool communication via the DMI.

DmGetDesignData and *dmPutDesignData* perform the actual in- and output of design data. They know the mapping to and from the storage structure employed. They do not restrict the formats of the detailed design data that can be transferred. The mechanism used is very similar to that of the printf() and scanf() functions in C.

DmCloseStream must be called to terminate a design data transaction. Files can be closed or allocated memory can be freed. Closestreammode specifies the details.

7.8.4.2.5 Calling pattern

As an illustration of how these functions co-operate, we present in Figure 7.16 a calling pattern of the functions of the DMI. The layered structure of the transaction schema is reflected in the indentation of the code.

```
envkey := dmInit (toolname);
  projectkey := dmOpenProject (envkey, projid, openprojmode);
    desobjkey := dmCheckOut (projectkey, desobjid, checkoutmode);
      streamkey := dmOpenStream (desobjkey, strname, iomode);
        dmPutDesignData (streamkey, format, arguments);
        dmGetDesignData (streamkey, format, arguments);
      dmCloseStream (streamkey, closestreammode);
    dmCheckIn (desobjkey, checkinmode);
  dmCloseProject (projectkey, closeprojmode);
dmQuit (envkey);
```

Figure 7.16. DMI Calling Pattern

7.8.4.2.6 Meta Design Data Access

The transaction schema presented sofar is mainly concerned with access to the detailed design data; it defines the procedure that must be followed to arrive at the contents of the design objects. However, the tools must also be allowed to access the meta design data, for instance, to obtain a module-list or to ask for and insert equivalences. For this purpose the functions *dmGetMetaDesignData* and *dmPutMetaDesignData* are provided.

- dmGetMetaDesignData (projectkey, request)
- dmPutMetaDesignData (projectkey, request)

DmGetMetaDesignData (DmPutMetaDesignData) can be used to obtain (store) information from (into) the meta design data of the project identified by projectkey. By means of the request argument, the specific retrieval or update operations (queries) can be passed to the DMS. These functions hide the actual implementation of the meta design data storage module.

Which queries can actually be formulated depends, of course, on the data schema employed by the DMS. For instance, a DMS supporting equivalence relationships between design objects of (possibly) different view-types, accepts queries on this information. Tools that interact at this level with the DMS are therefore vulnerable to schema changes. Decoupling can, however, be obtained as long as the queries that the tool formulates on its "application schema" can be mapped to (emulated by) queries on the DMS schema.

7.8.5 Discussion

The DMI introduced here formalizes the procedural aspects of the interaction between VLSI design tools and a DMS that provides a number of facilities to these tools. Openness is retained by avoiding tool-specific aspects in the DMI definition. Furthermore, we illustrated the ability of the DMI to absorb DMS-specific features that not necessarily have to be visible to the tools.

By offering a proper set of "anchor points", the DMI greatly facilitates software exchangeability: Modifications that are required to adapt a tool to another DMS are strictly local and do not alter the structure of the program. In fact, most of the work can usually be done mechanically, for example with a stream editor. As an illustration of the openness, we remark that integration of the KIC layout editor [Billingsley83] via the DMI was completed in about two days.

As an additional benefit, the DMI completely hides operating system dependencies. There are no path-names, file access modes, system calls etc. visible in the tools. Consequently, it should be easy to port the design system to other operating systems by changing the DMI library.

7.9 *CONCLUSIONS AND RESULTS*

We have presented a DMS for VLSI design that provides an open framework for the integration of design tools and relieves the designer from the burden of organizing his design data. After identifying the basic entity that is involved in a design transaction, the design object, we developed a data schema representing the logical organization of the VLSI design data. The OTO-D semantic data model, offering a set of well defined modeling constructs, permitted us to formalize the semantics of the meta design data. Openness of the DMS was secured by avoiding incorporation of features of a particular tool set or design representation.

The resulting data schema is simple (5 composed types), yet powerful enough to reflect the basic concepts of the DMS we outlined in section 1. It incorporates such features as hierarchical decomposition, multilevel design, design transactions / concurrency control, maintenance of verification statuses and versioning. Thus, it provides the basis for a DMS that implements a set of global system facilities to provide a framework for the construction of a powerful design environment. The DMS serves both the tools and the designer by making the information about the structure and status of the design available to them.

This DMS-research has been carried out in the context of the ICD-NELSIS system [Dewilde86a], which currently contains over 40 DMS-intensive CAD/IC tools such as layout editors and generators, design rule checkers, extractors, a placement and routing system, simulators, etc. These tools have all been equipped with the DMI defined in this paper. The decoupling introduced by this DMI turns out to be extremely useful for our software developments, as it permits DMS and tools to evolve separately to a large extent. Work is under way to define a DMI standard in a wider context [SECT87].

In the ICD-NELSIS system, several DMS-experiments have been carried out and a prototype has been produced. Currently, we are working towards a DMS-release that comprises all facilities described in this paper (and more). The meta design data storage module exploiting shared memory is operational, including the query facilities. An attractive version mechanism and state manager have been implemented. Interactive browsing facilities are currently being implemented. Bringing these parts together will yield an operational DMS at short notice, to which all tools can easily be interfaced via the DMI. The implementation of the DMS makes use of well known standards as the C programming language and the X-windows system. From the experience we have gained, it appears that the principles introduced in this paper indeed allow to do proper data management efficiently.

Future work will, among other things, focus on powerful design management facilities that can guide the designer through the design process towards a correct design. The first ideas for this have already been presented in section 5.1. Another area of interest is the management of information about such environmental aspects as technologies, tools, designers, etc, as a logical extension of the information modeling approach presented in this paper.

References

Afsarmanesh84. Afsarmanesh, H. and McLeod, D., "A Framework for Semantic Database Models," *Proc. NTU Symposium on New Directions for Database Systems*, (May 1984).

Batory85. Batory, D.S. and Kim, Won, "Modeling Concepts for VLSI CAD Objects," *ACM Trans. on Database Systems* **10**(3) pp. 322-346. (Sept 1985).

Bekke83. Bekke, J.H. ter, "Database Design (in Dutch)," *Stenfert Kroese*, (1983).

Bic86. Bic, L. and Gilbert, J.P., "Learning from AI: New Trends in Database Technology," *IEEE Computer Magazine* **19**(3) pp. 44-54 (March 1986).

Billingsley83. Billingsley, G. and Keller, K., "KIC: A Graphics Editor for Integrated Circuits," User's Manual, University of California at Berkeley (1983).

Birrell84. Birrell, A.D. and Nelson, B.J., "Implementing Remote Procedure Calls," *ACM Trans. on Comm. Sys.* **2**(1) pp. 39-59 (Feb 1984).

Brouwers87. Brouwers, J. and Gray, M., "Integrating the Electronic Design Process," *VLSI Systems Design*, pp. 38-47 (June 1987).

Buchmann84. Buchmann, A.P., "Current trends in CAD databases," *Computer-Aided Design* **16**(3) pp. 123-126 (May 1984).

Cox86. Cox, B.J., *Object Oriented Programming: An Evolutionary Approach*, Addison-Wesley Publishing Company (Aug 1986).

Dewilde86. Dewilde, P., Annevelink, J., Leuken, T.G.R. v., and Wolf, P. v.d., *Intelligent VLSI Datamanagement*, Delft University of Technology (1986).

Dewilde86a. Dewilde, P., Leuken, T.G.R. van, and Wolf, P. van der, "Datamanagement for Hierarchical and Multiview VLSI Design," pp. 1.1-1.29 in *The Integrated Circuit Design Book: Papers on VLSI Design Methodology from the ICD-NELSIS Project*, ed. P. Dewilde, Delft University Press, Delft (1986).

Gettys86. Gettys, J., Newman, R., and Fera, T. Della, "Xlib - C Language X Interface, Protocol Version 10," *MIT, Cambridge, Mass.*, (1986).

Hardwick87. Hardwick, M. and Spooner, D.L., "Comparison of Some Data Models for Engineering Objects," *IEEE CG&A*, pp. 56-66 (1987).

Jackson83. Jackson, M., *System Development*, Prentice Hall, Englewood Cliffs, N.J. (1983).

Katz83. Katz, R.H., "Managing the Chip Design Database," *IEEE Computer Magazine* **16**(12) pp. 26-35 (Dec 1983).

Katz83a. Katz, R.H. and Weiss, S., "Chip Assemblers: Concepts and Capabilities," *Proc. 20th IEEE Design Automation Conference*, pp. 25-30 (1983).

Leuken85. Leuken, T.G.R. van and Wolf, P. van der, "The ICD Design Management System," *Proc. IEEE ICCAD - 85*, pp. 18-20 (1985).

Lorie83. Lorie, R. and Plouffe, W., "Complex Objects and Their Use in Design Transactions," *Proc. Databases for Engineering Applications, ACM Database Week*, pp. 115-121 (May 1983).

Lyngbaek84. Lyngbaek, P., *Information Modeling and Sharing in Highly Autonomous Database Systems*, Ph.D. Thesis, Univ. of So. California, Los Angeles (August 1984).

Meijs87. Meijs, N. v.d., Leuken, T.G.R. v., Wolf, P. v.d., Widya, I., and Dewilde, P., "A Data Management Interface to Facilitate CAD/IC Software Exchanges," *Proc. IEEE ICCD '87*, (1987).

Newton86. Newton, A.R. and Sangiovanni-Vincentelli, A.L., "Computer-Aided Design for VLSI Circuits," *IEEE Computer Magazine*, pp. 38-60 (April 1986).

Niessen83. Niessen, C., "Hierarchical Design Methodologies and Tools for VLSI Chips," *Proceedings of the IEEE* **71**(1) pp. 66-75 (Jan 1983).

SECT87. SECT,, "Software Environment for CAD Tools," *The Sect Data Handling Committee*, CadLab, CNET, NMP-CAD, ES2, TU Delft, (1987).

Sidle80. Sidle, T.W., "Weaknesses of Commercial Data Base Management Systems in Engineering Applications," *Proc. 17th IEEE Design Automation Conference*, pp. 57-61 (June 1980).

Smith77. Smith, J.M. Smith and D.C.P., "Database abstractions: Aggregation and Generalization," *ACM Trans. Database Systems* 2(2) pp. 105-133 (June 1977).

Willems87. Willems, W.G.H.M., "A VLSI Design Manager Based on State Management," MS-Thesis, Delft University of Technology, Delft (Sept 1987).

Wolf86. Wolf, P. van der, "Conceptual Design of a Design Data Management System for VLSI Design," MS-Thesis, Delft University of Technology, Delft (July 1986).

BIBLIOGRAPHY

Ackland, B., Dickenson, A., Ensor, R., Gabbe, J., Kollaritsch, P., London, T., Poirier, C., Subrahmanyam, P., and Watanabe, H., "CADRE - A System of Cooperating VLSI Design Experts," *Proc. IEEE ICCD '85*, pp. 99-104 (1985).

Afsarmanesh, H. and McLeod, D., "A Framework for Semantic Database Models," *Proc. NTU Symposium on New Directions for Database Systems*, (May 1984).

Afsarmanesh, H., Knapp, D., McLeod, D., and Parker, A., "An Extensible Object-Oriented Approach to databases for VLSI/CAD," *Proc. Int. Conf. on VLDB '85*, (Aug. 1985).

Aho, A.V., Hopcroft, J.E., and Ullman, S.D., "The Design and Analysis of Computer Algorithms," *Reading MA Addison-Wesley*, (1974).

Anceau, F. and Aas, E.J., *VLSI 83, VLSI Design of Digital Systems*, North-Holland (1983).

Annevelink, J., "Object Oriented Data Management for VLSI Design," Internal Report, Delft University of Technology, Delft (Jan 1986).

Apollo, Computer Inc. and Bellerica, N., *Apollo Domain Architecture.* Feb. 1981.

Bachman, C.W., "Data Structure Diagrams," *Data Base* **1**(2) pp. 4-10 (1969).

Backus, J., "Can Programming Be Liberated from the von Neuman Style? A Functional Style and Its Algebra of Programs," *Comm. of the ACM.* **21. no 8.** pp. 613-641 (August 1978).

Bancilhon, F., Kim, W., and Korth, H.F., "Transactions and Concurrency Control in CAD Databases," *Proc. IEEE ICCD*, pp. 86-89

(1985).

Barbacci, M.R., "An Introduction to ISPS," *Technical Report, Department of Computer Science, Carnegie-Mellon University*, (1978).

Batory, D.S. and Kim, Won, "Modeling Concepts for VLSI CAD Objects," *ACM Trans. on Database Systems* **10**(3) pp. 322-346. (Sept 1985).

Bayer, R., Heller, H., and Reiser, A., "Parallelism and Recovery in Database Systems," *ACM Trans. on Database Systems* **5**(2) pp. 139-156 (June 1980).

Bekke, J.H. ter, "Database Design (in Dutch)," *Stenfert Kroese*, (1983).

Bekke, J.H. ter, "De Effectiviteit van Relationele Systemen," *Proc. Conf. Data: Beheer en Controle*, pp. 19-28 NGI, Sectie EDP-Auditing, (May 1985).

Bekke, J.H. ter, "OTO-D: Object type oriented data modeling," Report 86-02, Delft University of Technology, Delft (1986).

Bell, C.G. and Newell, A., "Computer Structures: readings and Examples," *Mc-Graw Hill Book Compagny, New York*, (1971).

Bennett, K.R., "A Practical Application of Data Modelling to Design System Integration," *Proc. IEEE Custom IC Conference*, pp. 436-440 (1984).

Bernstein, P.A., "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys* **13**(2) pp. 185-221 (1981).

Bernstein, P.A., "Query Processing in a System for Distributed Databases (SDD-1)," *ACM Transactions on Database Systems* **6**(4) pp. 602-625 (1981).

Bernstein, P.A. and Goodman, N., "Analyzing Concurrency Control Algorithms When User and System Operations Differ," *IEEE trans. on*

Software Engineering SE-9(3) pp. 233-239 (May 1983).

Bic, L. and Gilbert, J.P., "Learning from AI: New Trends in Database Technology," *IEEE Computer Magazine* 19(3) pp. 44-54 (March 1986).

Billingsley, G. and Keller, K., "KIC: A Graphics Editor for Integrated Circuits," User's Manual, University of California at Berkeley (1983).

Birmingham, W., Joobbani, R., and Kim, J., "Knowledge-Based Expert Systems and Their Application," *Proc. 23rd IEEE Design Automation Conference*, pp. 531-539 (1986).

Birrell, A.D., Levin, R., Needham, R.M., and Schroeder, M.D., "Grapevine: An Exercise in Distributed Computing," *Comm. of the ACM.* **25. no 4.** pp. 260-274 (April 1982).

Birrell, A.D. and Nelson, B.J., "Implementing Remote Procedure Calls," *ACM Trans. on Comm. Sys.* 2(1) pp. 39-59 (Feb 1984).

Bochmann, G.V and Sunshine, C.A., "Formal Methods in Communication Protocol Design," *IEEE tran. on Communications* COM-28(4) pp. 624-631 (April 1980).

Boggs, D.R., Shoch, J.F., Taft, E.A., and Metcalfe, R.M., "Pup: An Internetwork Architecture," *IEEE Trans. on Comm.* **28. no 4.** pp. 612-623 (April 1980).

Brachman, R.J., Fikes, R.E., and Levesque, H.J., "Krypton: A Functional Approach to Knowledge Representation," *IEEE Computer Magazine*, pp. 67-73 (Oct 1983).

Breuer, M.A., "Digital System Design Automation: Languages, Simulation and Data Base," *London Pitman*, (1977).

Brodie, M.L., "On Modelling Behavioural Semantics of Databases," *Proc. 7th IEEE Int. Conference on Very Large Databases*, pp. 32-42 (Sept 1981).

Brodie, M.L. and (Eds), S.N. Zilles, "Proc. of Workshop on Data Abstraction, Databases and Conceptual Modelling at Pingree Park, June 23-26 1980," *Sigplan Notices* **16**(1)(Jan 1981).

Brouwers, J. and Gray, M., "Integrating the Electronic Design Process," *VLSI Systems Design*, pp. 38-47 (June 1987).

Bryant, R., *Third Caltech Conference on VLSI*, Computer Science Press (1983).

Buchmann, A.P., "Current trends in CAD databases," *Computer-Aided Design* **16**(3) pp. 123-126 (May 1984).

Buckley, G.N. and Silberschatz, A, "Beyond Two-Phase Locking," *Journal of the ACM* **32**(2) pp. 314-326 (April 1985).

Bull,, GEC,, ICL,, Nixdorf,, Olivetti,, and Siemens,, *PCTE: A basis for a portable common tool environment*, Third edition, volume I 1985.

Bushnell, M.L. and Director, S.W., "VLSI CAD Tool Integration using the ULYSSES Environment," *Proc. 23rd IEEE Design Automation Conference*, pp. 55-61 (1986).

Chu, K.C., Fishburn, J.P., Honeyman, P., and Lien, Y.E., "A Database-Driven VLSI Design System," *IEEE trans. Computer-Aided Design* **CAD-5**(1) pp. 180-187 (January, 1986).

Codd, E.F., "A Relational Model of Data for Large Shared Data Banks," *Comm. of the ACM.* **13** pp. 377-387 (1970).

Cooper, E.C., "Analysis of Distributed Commit Protocols," *ACM Proceedings of SIGMOD Conference*, pp. 175-183 (June 1982).

Cox, B.J., "The Object Oriented Pre-Compiler: Programming Smalltalk 80 Methods in C language," *ACM Sigplan Notices* **18**(1) pp. 15-22 (Jan 1983).

Cox, B.J., "Message/Object Programming: An Evolutionary Change in Programming Technology," *IEEE Software Magazine*, pp. 50-61 (Jan 1984).

Cox, B.J., *Object Oriented Programming: An Evolutionary Approach*, Addison-Wesley Publishing Company (Aug 1986).

Daniel, M.E. and Gwyn, C.W., "CAD Systems for IC Design," *IEEE Trans. on CAD of Integrated Circuits and Systems* **CAD-1**(1) pp. 2-12 (Jan 1982).

Daniels, D., "An Introduction to Distributed Query Compilation in R*," *Distributed Data Bases*, pp. 291-309 North-Holland Company, (1982).

Danthine, A.A.S., "Protocol Representation with Finite-State Models," *IEEE tran. on Communications* **COM-28**(4) pp. 632-643 (April 1980).

Date, C.J., *An Introduction to Database Systems*, Addison-Wesley Systems Programming Series (1981).

Davio, M., Deschamps, J.P., and Thayse, A., *Digital Systems with Algorithm Implementation*, John Wiley & Sons (1983).

Dewilde, P., Leuken, T.G.R. van, and Wolf, P. van der, "Datamanagement for Hierarchical and Multiview VLSI Design," pp. 1.1-1.29 in *The Integrated Circuit Design Book: Papers on VLSI Design Methodology from the ICD-NELSIS Project*, ed. P. Dewilde, Delft University Press, Delft (1986).

Dewilde, P. ed., *The integrated circuit design book: Papers on VLSI design methodology from the ICD-NELSIS Project*, Delft University Press, Delft, The Netherlands (1986).

Dewilde, P., Annevelink, J., Leuken, T.G.R. v., and Wolf, P. v.d., *Intelligent VLSI Datamanagement*, Delft University of Technology (1986).

Director, S.W., Parker, A.C., Siewiorek, D.P., and Thomas, D.E. Jr., "A Design Methodology and Computer Aids for Digital VLSI Systems," *IEEE Trans. on Circuits and Systems* **CAS-28**(7) pp. 634-644 (July 1981).

EDIF,, "Electronic Design Interchange Format, Version 2 0 0, Reference Manual," *EDIF Steering Committee*, Electronic Industries Association, (1987).

Eastman, C.M., "Database Facilities for Engineering Design," *Proceedings of the IEEE* **69**(10) pp. 1249-1263 (Oct 1981).

Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L., "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM* **19**(11) pp. 624-633 (Nov 1976).

Eurich, J.P., "A Tutorial Introduction to the Electronic Design Interchange Format," *Proc. 23rd IEEE Design Automation Conference*, pp. 327-333 (1986).

Gajski, D.D., "ARSENIC Silicon Compiler," *Proc. ISCAS '85*, pp. 399-402 (1985).

Gajski, D.D., Dutt, N.D., and Pangrle, B.M., "Silicon Compilation (tutorial)," *IEEE 1986 Custom Integrated Circuits Conference*, pp. 102-110 (1986).

Gettys, J., Newman, R., and Fera, T. Della, "Xlib - C Language X Interface, Protocol Version 10," *MIT, Cambridge, Mass.*, (1986).

Glasser, L.A. and Dobberpuhl, D.W., "The Design and Analysis of VLSI Circuits," *Reading MA Addison-Wesley*, (1985).

Goldberg, A. and Robson, D., "Smalltalk80 The Language and its Implementation," *Addison-Wesley Pub. Co*, (1983).

Gonzalez-Sustaeta, J. and Buchmann, A.P., "An Automated Database Design Tool Using the ELKA Conceptual Model," *Proc. 23rd IEEE Design Automation Conference*, pp. 752-759 (1986).

Green, P.E., "An Introduction to Network Architectures and Protocols," *IEEE tran. on Communications* COM-28(4) pp. 413-424 (April 1980).

Guttag, J.V., Horowitz, E., and Musser, D.R., "Abstract Data Types and Software Validation," *Communications of the ACM* 21(12) pp. 1048-1064 (Dec 1978).

Hammer, M., "Reliability Mechanisms for SDD-1: A System for Distributed Databases," *ACM Transactions on Database Systems* 5(4) pp. 431-466 (1980).

Hardwick, M., "Extending the Relational Database Data Model for Design Applications," *Proc. 21st IEEE Design Automation Conference*, pp. 110-116 (1984).

Hardwick, M. and Yakoob, N., "Using a Database System and UNIX to Author CAD Applications," *Proc. IEEE ICCAD - 85*, pp. 53-55 (1985).

Hardwick, M. and Spooner, D.L., "Comparison of Some Data Models for Engineering Objects," *IEEE CG&A*, pp. 56-66 (1987).

Harrison, D.S., Moore, P., Spickelmier, R.L., and Newton, A.R., "Data Management and Graphics Editing in the Berkeley Design Environment," *Proc. IEEE ICCAD-86*, pp. 24-27 (1986).

Hartzband, D.J. and Maryanski, F.J., "Enhancing Knowledge Representation in Engineering Databases," *IEEE Computer Magazine*, pp. 39-48 (Sept 1985).

Haydamack, W.J. and Griffin, D.J., "VLSI Design Strategies and Tools," *Hewlett-Packard Journal*, pp. 5-12 (June 1981).

Haynie, M.N., "The Relational/Network Hybrid Data Model for Design Automation Databases," *Proc. 18th IEEE Design Automation Conference*, pp. 646-652 (1981).

Haynie, M.N., "Tutorial: The Relational Data Model for Design Automation," *Proc. 20th IEEE Design Automation Conference*, pp. 599-607 (1983).

Held, G.D., Stonebraker, M., and Wong, E., "INGRES - A Relational Database Management System," *Proc. 1975 Nat. Computer Conference*, AFIPS Press, (1975).

Ho, W.P.C., Hu, Y.H., and Yun, D.Y.Y., "An Intelligent Librarian for VLSI Cell Databases," *Proc. IEEE ICCD '85*, pp. 78-81 (1985).

Hoare, C.A.R., "Notes on Data Structuring," pp. 83-174 in *Structured Programming*, ed. E.W. Dijkstra, Academic Press, New York (1972).

Hollaar, L., Nelson, B., Carter, T., and Lorie, R.A., "The Structure and Operation of a Relational Database System in a Cell-Oriented Integrated Circuit Design System," *Proc. 21st IEEE Design Automation Conference*, pp. 117-125 (1984).

Horn, E.C. van, "Expressing Product Development Information in Application Terms," *Proc. IEEE ICCD '85*, pp. 82-85 (1985).

Israel, D.J., "The Role of Logic in Knowledge Representation," *IEEE Computer Magazine*, pp. 37-41 (Oct 1983).

Jackson, M., *System Development*, Prentice Hall, Englewood Cliffs, N.J. (1983).

Jullien, C., Leblond, A., and Lecourvoisier, J., "A Database Interface for an Integrated CAD System," *Proc. 23rd IEEE Design Automation Conference*, pp. 760-767 (1986).

Katz, R., "Computer-Aided Design Databases," *IEEE Design and Test*, pp. 70-74 (Febraury, 1985).

Katz, R.H., "A Database Approach for Managing VLSI Design Data," *Proc. 19th IEEE Design Automation Conference*, pp. 274-282 (June 1982).

Katz, R.H. and Weiss, S., "Chip Assemblers: Concepts and Capabilities," *Proc. 20th IEEE Design Automation Conference*, pp. 25-30 (1983).

Katz, R.H., "Managing the Chip Design Database," *IEEE Computer Magazine* **16**(12) pp. 26-35 (Dec 1983).

Katz, R.H. and Lehman, T.J., "Database Support for Versions and Alternatives of Large Design Files," *IEEE Trans. on Software Engineering* **SE-10**(2) pp. 191-200 (March 1984).

Katz, R.H. and Weiss, S., "Design Transaction Management," *Proc. 21st IEEE Design Automation Conference*, pp. 692-693 (1984).

Katz, R.H., Anwarrudin, M., and Chang, E., "A Version Server for Computer-Aided Design Data," *Proc. 23rd IEEE Design Automation Conference*, pp. 27-33 (1986).

Katzenelson, J., "Higher Level Programming and Data Abstractions - A Case Study Using Enhanced C," *Software Practice and Experience* **13** pp. 577 - 595 (1983).

Katzenelson, J., "Introduction to Enhanced C (EC)," *Software Practice and Experience* **13** pp. 551-576 (1983).

Katzenelson, J., *The Enhanced C Programming Language Reference Manual*, Technion - Israel Institute of Technology, Haifa 32000, Israel (March 21, 1985).

Kernighan, B. and Ritchie, D., *The C Programming Language*, Prentice Hall (1978).

Knapp, D.W. and Parker, A.C., "A Design Utility Manager: the ADAM Planning Engine," *Proc. 23rd IEEE Design Automation Conference*, pp. 48-54 (1986).

Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM* **21**(7) pp. 558-565 (July 1978).

Lechner, E., "CLIB - A Database Management System Designed for Standard Cell Applications," *Proc. IEEE ICCAD - 85*, pp. 21-23 (1985).

Leffler, S.J., Joy, W.N., and Fabry, R.S., *4.2BSD Network Implementation Notes*. 1983.

Leffler, S.J., Fabry, R.S., and Joy, W.N., *A 4.2BSD Interprocess Communication Primer*. 1984.

Leuken, T.G.R. van and Wolf, P. van der, "The ICD Design Management System," *Proc. IEEE ICCAD - 85*, pp. 18-20 (1985).

Leuken, T.G.R. van, "A Distributed Design Data Management System.," *ESPRIT project nr. 991. task 3*, (1986).

Leuken, T.G.R. van and Graaf, S. de, "The ICD User's Manual," *ESPRIT project nr. 991. task 3*, (1986).

Lindsay, B.G., Haas, L.M., Mohan, C., Wilms, P.F., and Yost, R.A., "Computation and Communication in R*: A Distributed Database Manager," *ACM trans. on Com. Sys.* **2. no 1.** pp. 24-38 (Feb. 1984).

Linn, J.L. and Winner, R.I., *Engineering Information Systems*, The Institute for Defense Analyses, Alexandria, Virginia (1986).

Lorie, R. and Plouffe, W., "Complex Objects and Their Use in Design Transactions," *Proc. Databases for Engineering Applications, ACM Database Week*, pp. 115-121 (May 1983).

Lyngbaek, P., *Information Modeling and Sharing in Highly Autonomous Database Systems*, Ph.D. Thesis, Univ. of So. California, Los Angeles (August 1984).

Mano, M.M., *Computer System Architecture Second Edition*, Prentice-Hall (1982).

Mano, M.M., *Digital Design*, Prentice-Hall (1984).

McCalla, B., "Chipbuster VLSI Design System," *Proc. IEEE ICCAD-86*, pp. 20-23 (1986).

McCalla, G. and Cercone, N., "Approaches to Knowledge Representation," *IEEE Computer Magazine*, pp. 12-18 (Oct 1983).

McLellan, P., "Effective Data Management for VLSI Design," *Proc. 22nd IEEE/ACM Design Automation Conference*, pp. 652-657 (July 1985).

McLeod, D., "Abstraction in Databases," *ACM special issue, Proc. of the workshop on Data Abstraction, Databases and Conceptual Modelling*, pp. 19-25 (1980).

McLeod, D., Narayanaswamy, K., and Rao, K.V. Bapa, "An Approach to Information Management for CAD/VLSI Applications," *Proc. Databases for Engineering Applications, ACM Database Week*, pp. 39-50 (May 1983).

Mead, C. and Conway, L., *Introduction to VLSI Systems*, Addison Wesley, Reading MA (1980).

Meijs, N. van der, Hoeven, A. van der, Vogel, T., Wolf, P. van der, and Graaf, S. de, "DBM Interface Functions: Applications Notes for Release

2," Internal Report nr. 85-28, Delft University of Technology, Delft (1985).

Meijs, N. van der, Hoeven, A. van der, Vogel, T., Wolf, P. van der, Leuken, T.G.R. van, and Dewilde, P., "DBM Programmers Manual: A Proposal for an ICD Standard," ICD Project Deliverable i.08, Delft University of Technology, Delft (June 1985).

Meijs, N. v.d., Leuken, T.G.R. v., Wolf, P. v.d., Widya, I., and Dewilde, P., "Data Management Interface Definition," *ESPRIT project, code 991, WP1, task 1.*, (Dec 31, 1986).

Meijs, N. v.d., Leuken, T.G.R. v., Wolf, P. v.d., Widya, I., and Dewilde, P., "A Data Management Interface to Facilitate CAD/IC Software Exchanges," *Proc. IEEE ICCD '87*, (1987).

Metcalfe, R.M. and Boggs, D.R., "Ethernet: Distributed Packet Switching for Local Computer Networks," *Communications of the ACM* **19**(7) pp. 395-404 (July 1976).

Moss, J.E.B., "Nested Transactions and Reliable Distributed Computing," *The Proceedings of the Symposium on Reliability in Distributed Software and Database Systems*, pp. 33-39 (July 1982).

Mukherjee, A., "Introduction to Nmos and Cmos VLSI Systems Design," *Prentice Hall*, (1986).

Myers, W., "Introduction to Expert Systems," *IEEE Expert* **1. no 1.** pp. 100-109 (Spring 1986).

Mylopoulos, J., "An Overview of Knowledge Representation," *ACM special issue, Proc. of the workshop on Data Abstraction, Databases and Conceptual Modelling*, pp. 5-12 (1980).

Mylopoulos, J., Shibahara, T., and Tsotsos, J.K., "Building Knowledge-Based Systems: The PSN Experience," *IEEE Computer Magazine*, pp. 83-89 (Oct 1983).

Nagel, L.W., "SPICE2: A Computer Program to Simulate Semiconductor Circuits," *University of California, Berkeley*, (May, 1975).

Navathe, S., Elmasri, R., and Larson, J., "Integrating User Views in Database Design," *IEEE Computer Magazine* **19**(1) pp. 50-62 (Jan 1986).

Newell, M. and Fitzpatrick, D.T, "Exploitation of Hierarchy in Analyses of Integrated Circuit Artwork," *IEEE Trans. on CAD* **CAD-1**(4)(Oct. 1982).

Newton, A.R., Pederson, D.O., Sangiovanni-Vincentelli, A.L., and Sequin, C.H., "Design Aids for VLSI: The Berkeley Perspective," *IEEE Trans. on Circuits and Systems* **CAS-28**(7) pp. 666-679 (July 1981).

Newton, A.R. and Sangiovanni-Vincentelli, A.L., "Computer-Aided Design for VLSI Circuits," *IEEE Computer Magazine*, pp. 38-60 (April 1986).

Newton, A.R., "Electronic Design Interchange Format, Introduction to (EDIFVersion 2 0 0)," *Proc. IEEE CICC '87*, pp. 531-535 (1987).

Niessen, C., "Hierarchical Design Methodologies and Tools for VLSI Chips," *Proceedings of the IEEE* **71**(1) pp. 66-75 (Jan 1983).

O'Neill, L.A., "Designers Workbench -- Efficient and Economical Aids," *Proc. 16th IEEE Design Automation Conference*, pp. 185-199 (1979).

Ozsu, M.T., "Modeling and Analysis of Distributed Database Concurrency Control Algorithms Using an Extended Petri Net Formalism," *IEEE tran. on Software Engineering* **SE-11**(10) pp. 1225-1239 (Oct 1985).

Pin, Y., Foo, S., and Kobayashi, H., "A Knowledge-Based System for VLSI Module Selection," *Proc. IEEE ICCD '86*, pp. 184-187 (1986).

Rieu, D. and Nguyen, G.T., "Semantics of CAD Objects for Generalized Databases," *Proc. 23rd IEEE Design Automation Conference*, pp. 34-40 (1986).

Roberts, K.A., Baker, T.E., and Jerome, D.H., "A Vertically Organized Computer-Aided Design Data Base," *Proc. 18th IEEE Design Automation Conference*, pp. 595-602 (1981).

Rose, C.W., Ordy, M., and Park, F.I., "NmPc: A Retrospective," *Proc. 20th IEEE Design Automation Conference*, pp. 506-514 (June, 1983).

Rothnie, J.B., "Introduction to a System for Distributed Databases (SDD-1)," *ACM Transactions on Database Systems* **5**(1) pp. 1-17 (March 1980).

Roussopoulos, N. and Yeh, R.T., "An Adaptable Methodology for Database Design," *IEEE Computer Magazine*, pp. 64-80 (May 1984).

Rowe, L.A. and Birman, K.P., "A Local Network Based on the Unix Operating System," *IEEE tran. on Software Engineering* **SE-8**(2)(March 1982).

SECT,, "Software Environment for CAD Tools," *The Sect Data Handling Committee*, CadLab, CNET, NMP-CAD, ES2, TU Delft, (1987).

Sarna, C.S., Reddy, G.R., and Hsieh, D., "Managing VLSI Data in a Workstation Configuration," *IEEE Cir. and Dev. Magazine* **2. no 4.** pp. 36-40 (July 1986).

Sequin, C.H., "Managing VLSI Complexity: An Outlook," *Proceedings of the IEEE* **71**(1) pp. 149-166 (Jan 1983).

Shipman, D.W., "The Functional Data Model and the Data Language DAPLEX," *ACM Trans. on Database Systems* **6**(1) pp. 140-173 (March 1981).

Sidle, T.W., "Weaknesses of Commercial Data Base Management Systems in Engineering Applications," *Proc. 17th IEEE Design Automation Conference*, pp. 57-61 (June 1980).

Silberschatz, A. and Kedem, Z., "Consistency in Hierarchical Database Systems," *Journal of the ACM* **27**(1) pp. 72-80 (Jan. 1980).

Smith, J.M. Smith and D.C.P., "Database abstractions: Aggregation and Generalization," *ACM Trans. Database Systems* **2**(2) pp. 105-133 (June 1977).

Spector, A.Z., "Performing Remote Operations Efficiently on a Local Computer Network," *Comm. of the ACM.* **25. no 4.** pp. 246-259 (April 1982).

Sun, Microsystems Inc., *Networking on the Sun Workstation.* Feb. 1986.

Teorey, T.J. and Fry, J.P., *Design of Database Structures*, Prentice-Hall, Englewood Cliffs, N.J. (1982).

Trimberger, S., "A Structured Design Methodology and Associated Software Tools," *IEEE Transactions on Circuits & Systems* **CAS-28**(7)(July 1981).

Tsichritzis, D.C. and Lochovsky, F.H., "Hierarchical Data-base Management: A Survey," *ACM Comput. Surv.* **8** pp. 67-103 (1976).

Tsichritzis, D.C. and Lochovsky, F.H., *Data Models*, Prentice-Hall, Englewood Cliffs, NJ (1982).

Tucker, M. and Scheffer, L., "A Constrained Design Methodology for VLSI," *VLSI Design*, pp. 60-65 (May/June 1982).

Vogel, T., Wolf, P. van der, and Dewilde, P., "Conceptual Database Model ICD," Internal Report, Delft University of Technology, Delft (Oct 1984).

Voss, K., "Using Predicate/Transition-Nets to Model and Analyze Distributed Database Systems," *IEEE tran. on Software Engineering* SE-6(6) pp. 539-544 (Nov. 1980).

Waxman, R., "VLSI - A Design Challenge," *Proc. 16th IEEE Design Automation Conference*, pp. 546-547 (1979).

Weiss, S., Rotzell, K., Rhyne, T., and Goldfein, A., "DOSS: A Storage System for Design Data," *Proc. 23rd IEEE Design Automation Conference*, pp. 41-47 (1986).

Weste, N. and Eshraghian, K., *Principles of CMOS VLSI Design a System Perspective*, Addison-Wesley (1985).

Widya, I., Leuken, T.G.R. v., and Wolf, P. v.d., "Concurrency Control in a VLSI Design Database," *Proc. 25th Design Automation Conference*, (1988).

Wiederhold, G., *Database Design*, Computer Science Press, Mc. Graw Hill (1977).

Wiederhold, G., Beetem, A.F., and Short, G.E., "A Database Approach to Communication in VLSI Design," *IEEE Trans. on CAD of Integrated Circuits and Systems* CAD-1(2) pp. 57-63 (April 1982).

Wiederhold, G., "Knowledge and Database Management," *IEEE Software Magazine* 1(1) pp. 63-73 (Jan 1984).

Willems, W.G.H.M., "A VLSI Design Manager Based on State Management," MS-Thesis, Delft University of Technology, Delft (Sept 1987).

Wolf, P. van der, "Conceptual Design of a Design Data Management System for VLSI Design," MS-Thesis, Delft University of Technology, Delft (July 1986).

Wolf, P. van der, Meijs, N. van der, Leuken, T.G.R. van, Widya, I., and Dewilde, P., "Data Management for VLSI Design: Conceptual Modeling, Tool Integration & User Interface," *Proc. IFIP WG 10.2 Workshop on Tool Integration and Design Environment*, North-Holland, (1987).

Wolf, P. van der and Leuken, T.G.R. van, "A Distributed Data Management System for VLSI Design," *Proc. 25th Design Automation Conference*, (1988).

Wolf, W., "An Object-Oriented Procedural Database for VLSI Chip Planning," *Proc. 23rd IEEE Design Automation Conference*, pp. 744-751 (1986).

Woods, W.A., "What's Important About Knowledge Representation?," *IEEE Computer Magazine*, pp. 22-27 (Oct 1983).

Yalamanchili, S., Malek, M., and Aggarwal, J.K., "Workstations in a local Area Network Environment," *IEEE Computer Magazine*, pp. 74-86 (Nov 1984).

Yannakakis, M., "A Theory of Safe Locking Policies in Database Systems," *Journal of the ACM* 29(3) pp. 718-740 (July 1982).

Zara, R.V. and Henke, D.R., "Building A Layered Database for Design Automation," *Proc. 22nd IEEE/ACM Design Automation Conference*, pp. 645-651 (July 1985).

Zintl, G., "A Codasyl CAD Data Base System," *Proc. 18th IEEE Design Automation Conference*, pp. 589-594 (1981).

Curriculum Vitae

Na zijn geboorte op 14 april 1955 in 's-Gravenhage doorliep René van Leuken achtereenvolgens de kleuterschool, de lagere school, het atheneum en de Technische Universiteit te Delft om in 1983 het diploma voor elektrotechnisch ingenieur te behalen, waarna hij als wetenschappelijk assistent in de vakgroep Netwerktheorie van de Technische Universiteit Delft, onder leiding van Prof. dr. ir. P. Dewilde, onderzoek verrichtte aan database modellering, programma interfaces voor databases en mede leiding gaf aan de ontwikkeling van een systeem voor het ontwerp van *geintegreerde schakelingen.*