National Library of Canada
Bibliothèque nationale du Canada

Ottawa, Canada
K1A 0N4

TC –

ISBN 0-315-23314-1

CANADIAN THESES ON MICROFICHE SERVICE – SERVICE DES THÈSES CANADIENNES SUR MICROFICHE

PERMISION TO MICROFILM – AUTORISATION DE MICROFILMER

• Please print or type – Écrire en lettres moulées ou dactylographier

AUTHOR – AUTEUR

Full Name of Author – Nom complet de l'auteur

[illegible]

Date of Birth – Date de naissance

[illegible]

Canadian Citizen – Citoyen canadien

Yes Oui [✓ No Non]

Country of Birth – Lieu de naissance

[illegible]

Permanent Address – Résidence fixe

[illegible]

THESIS – THÈSE

Title of Thesis – Titre de la thèse

[illegible]

Degree for which thesis was presented
Grade pour lequel cette thèse fut présentée

[illegible]

Year this degree conferred
Année d'obtention de ce grade

[illegible]

University – Université

[illegible]

Name of Supervisor – Nom du directeur de thèse

[illegible]

AUTHORIZATION – AUTORISATION

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to **microfilm** this thesis and to **lend** or **sell copies of the film.**

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de **microfilmer** cette thèse et de **prêter** ou de **vendre des exemplaires du film.**

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

ATTACH FORM TO THESIS – VEUILLEZ JOINDRE CE FORMULAIRE À LA THÈSE

Signature

[illegible]

Date

[illegible]

NL-91 (r. 84/03)

Canada

The University of Alberta

An Object Oriented Database Management System for CAD Applications

by

Bopsi CHANDRAMOULI

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Fall, 1985

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Bopsi CHANDRAMOULI

TITLE OF THESIS: An Object Oriented Database Management System for CAD Applications.

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1985

(Signed) [illegible]

Permanent Address:
36, Thyagaraya Gramani Street,
T'Nagar,
Madras 600 017, INDIA.

Dated October 8, 1985

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **An Object Oriented Database Management System for CAD Applications** submitted by **Bopsi CHANDRAMOULI** in partial fulfillment of the requirements for the degree of **Master of Science.**

[illegible]
Supervisor

W. Armstrong

[illegible]

[illegible]

Date [illegible]

To my family and friends

ABSTRACT

This thesis is concerned with database requirements for computer aided design applications. Software tools greatly ease and simplify complex processes such as the design of VLSI chips, large software systems, mechanical and civil engineering works and the composition of music, etc. It has been well accepted that databases are an important element in building these software tools. Experience with network and relational database management systems (DBMS) in design environments has suggested that design tools require more flexibility, more data definition facilities, and more data manipulation power than what network and relational DBMSs offer. The main reason for this is that these DBMSs are aimed at business applications whose nature is inherently different from design applications.

The main contributions of this thesis are the formulation of a model for the design activity, precise definition of the database requirements of design applications and the design and implementation of a Database Management System to satisfy what we consider to be the most important of these requirements. The interesting features of this DBMS are a simple view of the database as one consisting of Objects and Properties, the provision for procedural derivation of the value of a property, a data type 'SET' for specifying and manipulating all kinds of relationships, special mechanisms for specifying the constraints on the contents and structure of the database and the provision for dynamic updating of the structure of the database. Using the data type 'SET' it is possible to represent and manipulate a complex object as a whole. A Data and Constraints Specification Language (DCSL), used to specify the initial structure of the database, is designed and a compiler for the language is written using Lex and Yacc. The compiler builds the initial structure of the database from the specifications. Several routines are designed and implemented for dynamic definition and manipulation of data and relationships.

Acknowledgements

I would like to thank my supervisor, Dr. Mark Green, for his valuable advice, for making himself available for discussion in spite of his busy schedule, and his encouragement and criticism at each stage of my research. The quality and readability of this thesis depends to a large extent on his careful and patient reading of the rought draft of the thesis. I would also like to express my gratitude for his financial support

Thanks are due to the members of my examining committee, Dr. W.W. Armstrong, Dr. E. Chan and Dr. E. Girczyc, for their helpful suggestions and comments. I would also like to acknowledge the Teaching and Research Assistantships provided by the Department of Computing Science.

Most of all, I would like to thank my family and friends, for their support and encouragement throughout my academic career.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Most of the design environments for VLSI, software engineering, ship design, and other engineering activities are characterized by complex but similar information management requirements. There has been a growing interest in using databases across a variety of such disciplines. In this thesis, we are concerned with the analysis of the database requirements of design environments and the design and implementation of a database management system suitable for design environments.

1.1. Complexity of the Design Activity

To illustrate the nature of the information management requirements of design environments let us consider a fairly complex design domain, namely the design of ships. Getting the design of the ship from the early stages to the point where it represents a stable, fully functioning sea vessel is a process involving thousands of man hours, thousands of plans and drawings and enormous financial resources. This process is characterized by a massive and constantly growing amount of detail as the design evolves towards the complex array of welds, joints, pipes, cables, fittings and other components required to construct the engines, turbines, boilers, reduction gears etc. A similar scenario can be given for other complex design domains like VLSI, Architecture and Software Engineering.

The designers' intellectual productivity is adversely affected by the amount of detail that they have to manage. Throughout the design activity, the designer experiences short bursts of creativity followed by relatively long periods of non-creative arduous battle with the data (technically called "Information Management"). Accumulating and retrieving these data is time-consuming and tedious. Here is where the database management systems come into play. They provide the designers, actually the developers of design tools, with the facilities they need, so they are not concerned with the 'how' of information management.

1.2. The Evolution of the use of databases in design systems

The earliest uses of computers in design were as a superior version of slide-rule, desk calculator or the book of tables. Even though the information management requirements were not less complex at that time, computers were not fast and big enough to support design applic~~ations~~. As the hardware and software technology advanced, new applications were conceived for computers and the design application was one of the major ones.

In the mid and late 1960's, when computerized design environments were beginning to appear, the design systems consisted of a set of programs operating on their own input and output files. One such example is COMMEND I from IBM[38]. These systems faced enormous problems of lack of data integrity and consistency, high amount of redundancy of data, frequent and costly conversion of data from one format to another and lack of program-data independence. This situation is somewhat similar to the one faced by the business environment. But the business environment did not have to spend time and money to develop data structures for modeling since their modeling requirements were considerably simpler. In design environments the modeling requirements were so complex that considerable effort was spent in each system in designing and constructing data structures used for modeling. One exception was the work done by United Aircraft Research Laboratories in cooperation with IBM[50] which is more general in nature than most of the other systems of that time. But even this was specific for modeling 3-dimensional geometric objects. So it is appropriate to say that the database technology has contributed more benefits to the design environments than to the business environments. The early 1970's saw many design systems starting to use database management systems for their information management requirements. Even though the database management systems based on Hierarchical and Network models were not ideal for modeling in design environments, the developers of design tools enjoyed all the advantages that are usually attributed to database management systems. The mid & late 1970's provided another significant improvement in the use of database management systems in design systems when database management systems based on the Relational model were developed[2,56]. The Relational model's simplicity, strong theoretical foundation and the powerful relational algebra[15,16,44] are very useful in modeling and manipulating the design objects[29].

It was realized in the early 1980's that even relational database management systems have some shortcomings in their ability to act as modeling tools in design environments. For example, it is difficult and unnatural to model the complex objects (parts being made out of sub-parts) that one encounters in design applications using the relational model (this and other related topics are discussed in detail in chapter 3).

1.3. The Problem Area

As noted in the last section, it has been generally felt by researchers that the facilities provided by the existing database management systems are not quite sufficient for the design environment. There is a mismatch between what the design tools expect and what the existing database management systems provide. The fundamental reason for this is that these database management systems are all intended for business applications whose requirements are quite different from the requirements of design applications. So there has been considerable research effort in finding what features a database management system should have to meet the expectations of the design environments. Also there is a general lack of an adequate model of the design activity. A sufficiently sound model of the design activity is very useful in viewing its information management requirements in the right perspective. That right perspective is the key for the success and utility of any design software in general and the database management systems in particular. A careful study of such a model can also shed some light in identifying requirements which have not yet been recognized.

1.4. Objectives of the Thesis

The objectives of this thesis are 1) to arrive at an adequate model of the design activity so that research in Computer-Aided Design can be based on a solid foundation, 2) to derive a set of requirements (through a literature survey and through a careful study of the model) that the design activity and hence the design tools expect of a database management system, and 3) to design and implement a prototype of a database management system to provide the features we feel are most important. This prototype can act as an experimental tool for further research in computer-aided design (for example, refinement of the model of the design activity, integration of a design database with a business database etc.) in addition to being used in design tool development and serving a useful and central role in computerized design environments.

1.5. Approach Taken

The database management system we have designed is "object oriented". We call our system "object oriented" because the concept of "Objects" is used to model the entities of the real world. The term "object-oriented" is not new. Many programming languages provide the necessary features to support the object oriented design methodology. Also, there are many database systems which are based on the concept of "Objects". We briefly describe below the object oriented design methodology and mention some of the object oriented database systems.

1.5.1. Object Oriented Design

Abstraction is a fundamental conceptual tool used for modeling real world phenomenon. Accordingly, many problem solving and software design methodologies are based on the concept of abstraction. One such design methodology is the object oriented design methodology. The Object Oriented Design Methodology[6,49,59] is supported by languages such as Simula[17], Smalltalk[22,30,31], Ada[3] and Modula-2[60]. In this methodology, the importance of software objects as actors, each with its own set of applicable operations is recognized. The software designer would first identify objects and their attributes. The designer would then proceed to identify the operations on the objects and then to establish the interfaces. Finally the designer would implement these operations.

1.5.2. Object Oriented Systems

There are many systems which are based on the concept of "Objects". Among the important ones are TAXIS[7,48,61], ACM/PCM[8] and the event model[37]. All these systems are concerned with modeling real world entities and their semantic relationships rather than pure data organization and hence can be classified as work in Semantic/Conceptual Data Models[7]. They all address the problems and difficulties that arise in designing complete database systems. They emphasize the importance of modeling the dynamic/behavioral, not just the static aspects of a real world phenomena and the need to integrate these two facets of the description. It should be mentioned that these systems are concerned with modeling real world phenomenon whose structure does not change dynamically.

1.6. Outline of the Thesis

The organization of the thesis is as follows. Chapter 2 introduces a simple model of the design activity and describes various aspects of that model. The elements of a computerized design environment are discussed. In light of this model, the role of a database management system is pointed out. Chapter 3 presents a detailed discussion on the requirements of a design database management system. Chapter 4 describes the design principles and the model behind the database management system that was conceived after a study of the requirements described in chapter 2. Chapter 5 describes the implementation of that database management system and the associated data definition and manipulation language. Chapter 6 provides examples to illustrate the use and the important features of the database management system. Chapter 7 provides the conclusion and a discussion on the scope for further research in this area. The appendices provide the formal definition of the data definition language and the header declarations of the functions provided by the system.

Chapter 2

A Model of the Design Activity

In this chapter, we present a model of the design activity. This model is presented here to put in proper perspective computer-aided design in general and design database management systems in particular.

The activity "design" involves mainly information processing. This readily suggests that computers can be used to provide support for this activity since computers are basically information processing machines. For any such support to be meaningful and successful it is a fundamental requirement that one should have a clear understanding of the design activity. We need at least a crude model for the design activity so that any decisions that we make on the form of computer support are appropriate. We give below one such model which is quite adequate for our present purposes. Most of the discussions in chapter 3 of this thesis are based on this model. This model is an extended form of the one presented in[51] to include the concepts of versions and abstractions.

2.1. The Model

Fig. 2.1 is a highly abstracted view of the design activity. Several things are worth noting here.

-- the design goal derives out of some need for or desire
 to evolve and is usually formulated outside the design activity.
 Usually, it does not change for at least some period of time;

-- designer's knowhow, in addition to various other resources,
 is an important resource that is required to perform the design;

-- the major output of the design activity is information.
 i.e. the design, which is usually input to various other
 activities like manufacturing, testing, marketing, maintenance etc.

Fig. 2.2 shows a detailed view of the design activity. This figure shows the design activity as a feedback control loop. It should be noted where the two inputs "design

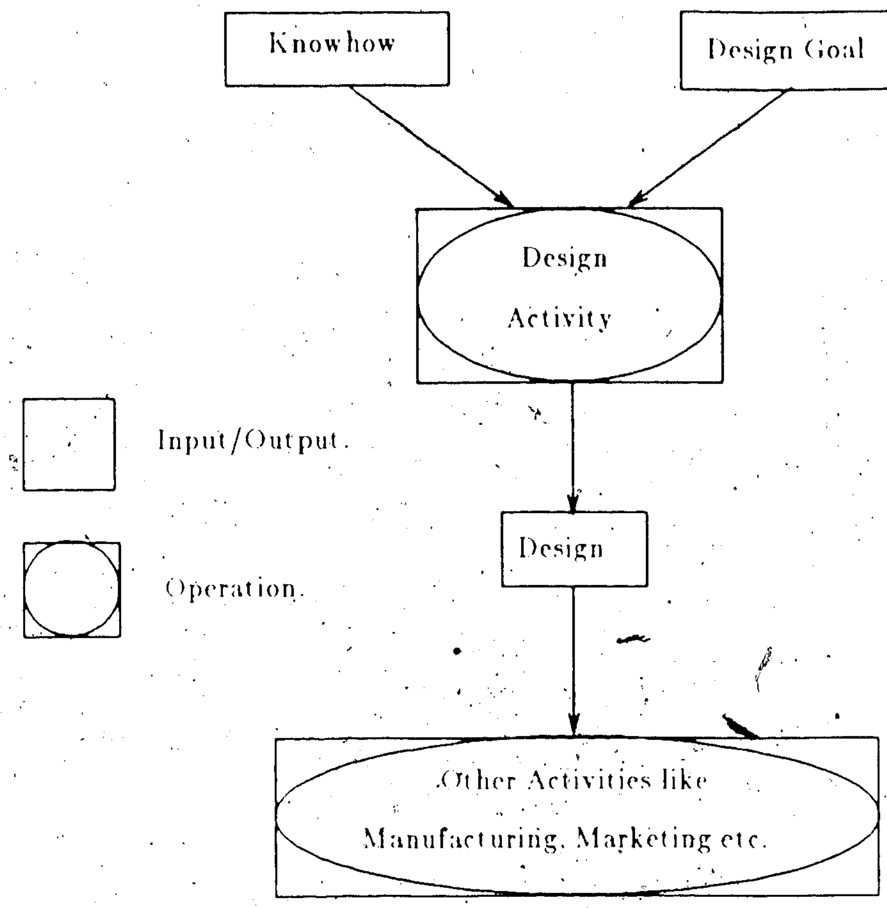

Figure 2.1 Abstract Model of the Design Activity.

goal" and "knowhow" are represented in this figure.

A higher level activity generates the design specifications from the design goal. Based on these specifications, certain important decisions on the design entity are made heuristically. This is the "synthesis" part of the design activity. This is the most creative part of the design activity and involves the designer's technical knowledge, experience, judgement and intuition. In most of the cases the designer is not aware of the consequences of these decisions with respect to the design specifications at this stage. The output of synthesis is an intuitive conceptual model of the entity being designed. It must be "analyzed" and "evaluated" against the design specifications. The construction of the conceptual model is not a once-through sequence of actions but involves the designer going over the model again and again to fill in bits and pieces.

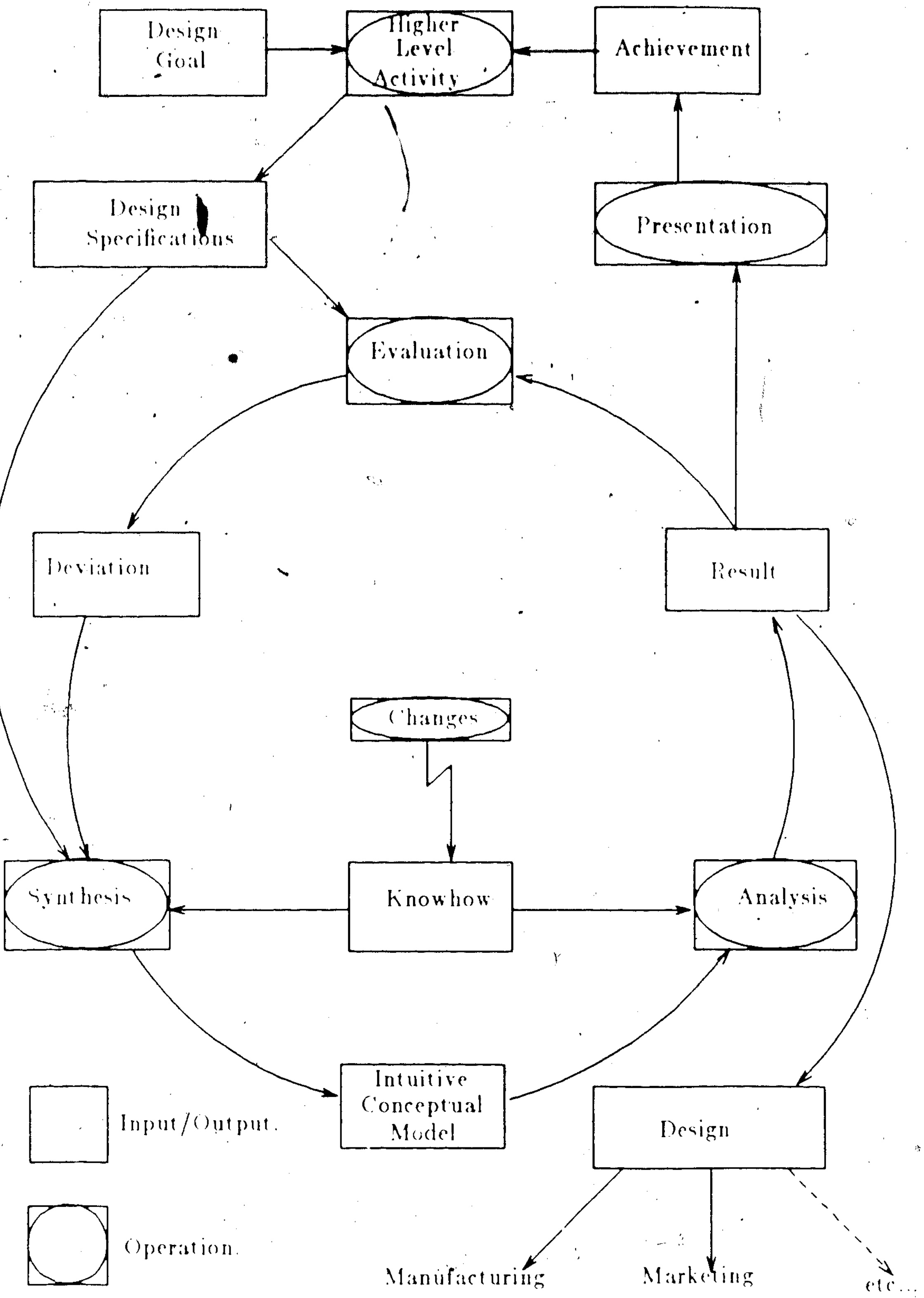

Figure 2.2 Detailed View of the Design Activity

Claims of structured design not withstanding, humans rarely design anything in a completely structured fashion. The end output may be highly organized but in getting there the designer is likely to hop all over the design filling in information with varying degree of details until he and the results of analysis and evaluation agree that the design meets the design specifications. Thus while the deviations from the specifications persist the appropriate design decisions ("synthesis") must be corrected. This is the inner feedback control loop. The results of the inner loop are presented as "achievement" to the higher level activity which initially generated the design specifications. The higher level activity can change the design specifications after comparing the achievement with the design goal. This is the outer feedback control loop. The activities in the inner loop are performed again if the design specification is changed. Thus, especially for large and long-running projects like the chemical power plant, big sea vessels, the space shuttle, big software systems etc., the design specification is a moving target for the inner loop.

2.1.1. Versions

In practice, however, the output of the inner loop, at the same time it is being presented to the higher level activity, may be considered adequate for being sent to activities like manufacturing etc. This essentially means that at the same time development work is being done on the design, the design may be considered as a "finished design". This creates "versions" for the design which is an important aspect of the design activity.

2.1.2. Abstractions in Design

The synthesis task is shown in further detail in fig. 1.3.

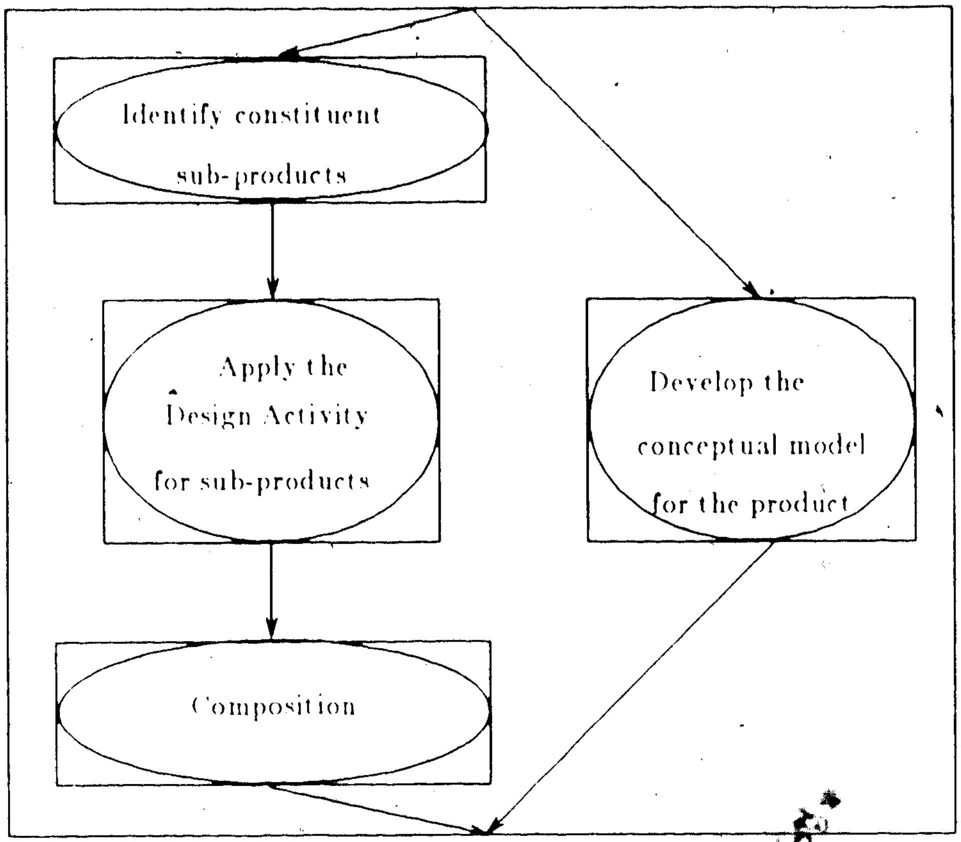
Identify constituent sub-products
Apply the Design Activity for sub-products
Develop the conceptual model for the product
Composition

Figure 2.3 Detailed View of the Synthesis Task.

Fig. 2.3 presents a recursive description of the whole design activity. The synthesis of the product involves

(i) identifying further less complex products which go to make the product;

(ii) applying the design activity to these less complex products;

(iii) composition; make decisions as to how these sub-products can be combined to evolve the product being designed.

Or, if the product can not be broken up further then its "intuitive conceptual model" is formulated by making decisions on the model heuristically. Th [illegible] ts in a network like product structure. Each stage represents an abstraction: the lower the level, the more detailed the abstraction is. In some practical cases, the network gets simplified to a hierarchy and hence the commonly known term "hierarchical design".

It is interesting to note that each level of abstraction can have versions. This is because the design activity is applied to each sub-product and each one of these activities can potentially cause versions. Thus these two concepts namely "Abstraction" and "Version" can cause an explosion of information which is difficult to manage manually.

2.2. The Concept of Computer-Aided Design

As we mentioned earlier in this chapter, design is mainly information processing. With Computer Aided Design (CAD) part of the information processing is delegated to the computer. The concept of Computer-Aided Design can be adequately described as a process by which the computational and memory characteristics of the computer are combined with the imagination and reasoning power of humans for the purpose of designing, analyzing and implementing some complex entity or system[21]. Simplistically, CAD provides computer support for the design activity.

2.3. Computerized Design Environment

In this section we present the software elements of a computerized design environment and indicate the central role played by the database and the database management system.

The model of the design activity helps us to identify the areas where computers can be usefully employed. In other words, it helps us to identify the important and necessary elements of a Computerized Design Environment. It is generally accepted that the synthesis and evaluation tasks are best done by the human with the aid of computers whereas the analysis task is best done by the computer (the knowhow being conveyed to the computer thorough data and programs). The fundamental basis for all the benefits of computerized design environment is that the human performs the tasks he can do best, namely identifying problems, applying economic and engineering judgement and putting forth solutions to the problems, and computers perform the tasks they can do best, namely racing through mountains of data, searching, comparing and calculating with remarkable speed, accuracy and reliability. Thus the whole system is tuned to increase the intellectual productivity of the humans.

Even though the synthesis task is mainly performed by a human, the computer has a very important role to play. The computer can aid the designer in formalizing the conceptual model which is in the mind of the designer. In a computerized design environment, the formalized model is the machine-readable representation of the conceptual model. A machine-readable representation of the conceptual model can be structured as a database. Thus the first and a very important element of the design environment is the "Database". To transform the conceptual model primitives to the database primitives we need a Data Model and the associated Database Management System. Some tools like smooth surface design aids, solid modeling aids etc. are also very useful in constructing this formal model. Also highly desirable for the synthesis task would be an intelligent design assistant which can learn from experienced designers and help the novices in learning the art of synthesis by assisting them with the standard procedures. Another duty of the design assistant can be to help the designer locate what he wants. It can be an intelligent browser through the library of design information locating design entities given an incomplete and fuzzy functional specifications of them. This will be very helpful in the synthesis task when the designer tries to realize each of the identified abstractions of the design object. The design assistant can also do some routine tasks like determining when the database should be restructured, running batch jobs at appropriate times, producing various statistics and reports on the status of the design activity (like schedules for the design tasks, report on incomplete design parts etc.). As techniques of artificial intelligence are becoming more advanced, the day the designer can enjoy such an intelligent design assistant is not far off.

The communication between the designer and the formalized model takes place through an intermediate representation of the model and thus the User Interface is another important element of the design environment.

The design data is used by the analysis task to produce the appropriate information for the evaluation task. There are various tools which accomplish the analysis tasks. Some examples are the different varieties of simulators, translators between various representations of the design data to aid in the analysis task, finite element analysis programs etc. These are generally termed "Analysis Aids".

In the evaluation task, when the results of the analysis task are interpreted, computers can provide support to the designer by generating various tables and charts for comparing and making decisions. Such tools like mathematical subroutine packages are generally termed "Evaluation Aids".

From the above discussion the central role played by the design database is evident. Both the synthesis and analysis tasks use the database. Also, as mentioned in the last section the design is intended for various activities like manufacturing, testing etc. Thus the database provides a common representation of the model for the various tasks of the design activity and for various other activities as well.

To summarize, the important elements of a design environment are,

1) The Database, the Data Model, the Database Management
System and related tools;

2) Intelligent Design Assistant;

3) User Interface;

4) Analysis Aids and

5) Evaluation Aids.

2.4. The Design Database Management System

We have seen in the last section that the database is central to most of the design tasks in a computer-aided design environment. It provides a common representation of the conceptual model for both the synthesis and analysis tasks. The utility of various software tools that are intended to aid in design depends very much on the facilities provided by the database management system. Tools emerge to satisfy a particular need in the design environment and their information management requirements are governed by the nature of the design activity. So the design database management systems should be designed considering the requirements of the design activity. This ensures that they provide the facilities that the tools expect and thus the development of tools become easier and economical. Thus the programmer's productivity, to some degree, depends on the database management system.

Our model also indicates that the design information is intended for, and used by, other activities like manufacturing, purchasing, maintenance etc. In a computerized design environment this essentially means that the database will, at the least, be consulted by these activities. These activities also usually maintain databases for their information management requirements. Their information management requirements may be different in nature from the requirements of the design activity but a substantial amount of the design data will be duplicated to their databases. For example, the list of materials is an important constituent of the database for purchasing and this data originates from the design data. The facilities offered by the design database management system would, to a large extent, determine the efforts involved in transferring appropriate data to other databases. Integration of these databases, though difficult due to the differences in the information management requirements, can solve many problems like integrity and consistency of data among the databases.

Design database management systems will play a key role in bringing about a successful integration.

The above discussion shows that the design database management system is a very important and critical element in a design environment. The success and utility of design environments largely depends on the database management system.

The model of the design activity we have presented serves a very useful role in developing the design database management systems. The information management requirements of the design activity, so far recognized as necessary by researchers in this field, can be identified with particular aspects of the model so that these requirements can be put in proper perspective and further important requirements can be derived with this model as the basis. As we mentioned before, these requirements are the basis on which the design database management systems should be developed so that the gap between what the tools expect and what the database management systems provide is narrowed.

2.5. Summary

In this chapter, we have introduced and discussed an adequate model of the design activity. With this model as the reference we have identified the potential areas where computers can be employed to aid the design activity. Again using the model, we have stressed the central and important role of database management systems in design environments. We have also indicated how the model can be used in developing the design database management system. The next chapter describes the requirements of a design database management system and subsequent chapters describe the development of a database management system based on these requirements.

Chapter 3

Requirements of A Design Database Management System

As we noted in chapter 1, it is the general feeling that there is a disagreement between what the design tools expect and what the existing DBMSs provide. Out of this arose many research efforts to identify what the requirements of design database management systems are[4,34,42,45,52]. The first DBMSs conceived for engineering and design purposes such as the systems that evolved out of the IPAD project[33] are maturing and several attempts have been made to extend existing DBMSs to provide capabilities for supporting CAD applications[27,28,55]. These were practical attempts to determine the novel and intelligent ways to expand the scope of the use of databases and how existing DBMSs and data models can be adapted for CAD applications. These systems have provided insights into identifying the shortcomings of the conventional databases in their use in design environments and in deriving a comprehensive set of information management requirements for design environments. A careful study of the model we have described in chapter 2 helped us to justify some of the requirements and to augment this set of requirements. In this Chapter we describe such a comprehensive set of requirements expected of a CAD DBMS which sets forth the foundation for this thesis. The discussion also sheds light on the direction current research in this area is taking.

3.1. Ability to Represent and Handle Complex Objects

This requirement derives from the designer's need to refer to objects which often consist of tens or even hundreds of items. The DBMS is required to manage the structure of a large complex object with several components interconnected in a potentially complex manner. The structure used for representing such objects may obviously be more complex than single records or sets of homogeneous records. It should be possible to define these objects so that some global operations such as delete, move, copy and lock could be handled by the system on an object basis. Standard DBMSs don't provide the facility to represent a complex object as a whole. For example, in the relational model the representation is entirely left to the user. He must represent a complex object as a collection of tuples in several relations and the relationship between these tuples is expressed by matching values. But this does not tell the system explicitly that all these related tuples form a single complex object. If an object is to be deleted, the application program must use several DELETE tuple commands. Thus the key here is to make the complex object known to the system.'

Let us consider, as an example, the design of electronic components. Even though it is simpler than the objects involved in proper VLSI design it demonstrates the issues involved in representing and manipulating complex objects. Consider a 4-AND gate object built out of three elementary 2-AND gate objects. Figure 3.1 shows the design. The design is simple and self-explanatory.

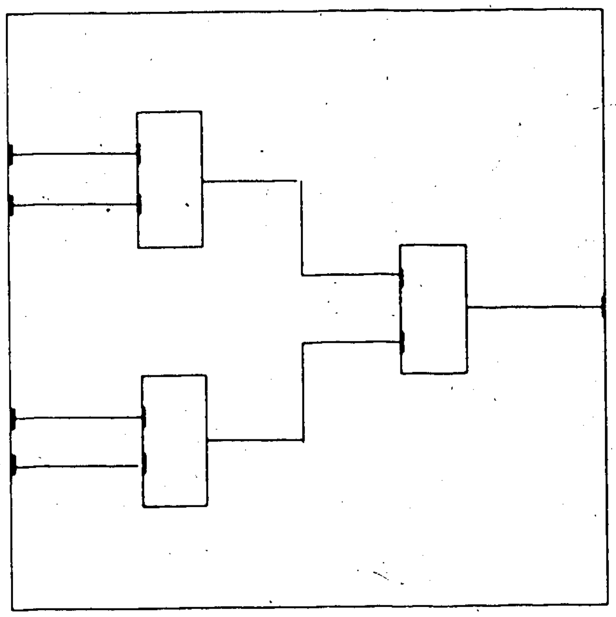

Figure 3.1 4-AND Gate Object

The hierarchical model of the gate object (both 4-AND and 2-AND gates) is shown in figure 3.2. This incidently describes both topological and graphical aspects of the

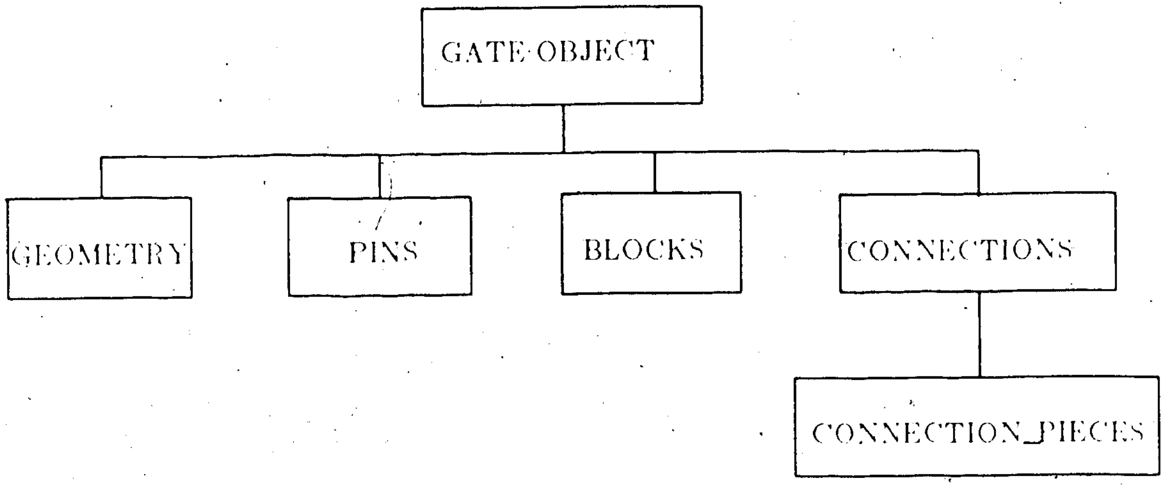

Figure 3.2 Hierarchical Model of the 4-AND Gate Object

objects. The object GEOMETRY contains the information about the coordinates of the gate object. The object PINS specifies the exterior pins of the gate object. The object BLOCKS contains information about objects used inside the gate object. The object CONNECTIONS specifies the topology of the connections between objects. The object CONNECTION-PIECES contains information about how each connection is actually represented graphically.

This model can be mapped to relations as follows.

```
Relation GATE(id : INT, name : STRING,....)
Relation GEOMETRY(gateid : INT, X1:INT,Y1:INT,X2:INT,Y2:INT,....)
Relation PINS(gateid :INT, no : INT;class :STRING,X:INT,Y:INT)
Relation BLOCKS(gateid : INT, no : INT, type : INT,x : INT,y:INT....)
Relation CONNECTIONS(gateid : INT,no: INT,startblock : INT,
            startpin : INT,endblock : INT, endpin : INT)
Relation CONNECTION-PIECES(gateid : INT, conn-no: INT, X: INT, Y : INT)
```

The relations are fairly self-explanatory. The important thing to note here is how the hierarchical relationships are represented in the relations. The child object always carries the primary key of the parent object. It can be shown that for the diagram in figure 2.1 a 4-AND gate is represented by 24 tuples in 6 relations. A tuple in relation GATE actually identifies the object and the other 23 tuples are logically linked together by the key of the various relations to define all the components of the object

As mentioned before, the problem with such a representation is that it does not tell the system explicitly that all these logically linked tuples form a single, complex object. If the 4-AND gate is to be deleted then the application program has to issue delete calls for all the 24 tuples. The same holds true for moving, copying or locking objects as a whole. This has severe data integrity implications. Maintaining the integrity of data requires very carefully programmed applications performing the operations. We call such representations of complex objects as APPLICATION DEFINED COMPLEX OBJECTS in contrast to SYSTEM DEFINED COMPLEX OBJECT wherein the DBMS is aware of the complex object and is capable of doing operations on the object as a whole. Examples of systems which provide the latter facility are described below.

Many attempts have been made to incorporate the facility of SYSTEM DEFINED COMPLEX OBJECT into DBMSs.

3.1.1. IPIP, a DBMS that was developed within the IPAD project at Boeing[33], has a facility for definition, and manipulation of high-level user-defined objects. Changes are propagated within the objects and deletion of the highest-level record in an object triggers deletion of all the records constituting the object.

3.1.2. Stonebraker[57] suggests the definition of abstract data types. By this approach, new data types (Data Structures + Operations on them) for columns of a relation such as boxes, wires and polygons become possible. A.P. Buchmann[9] is of the opinion that this approach may be feasible for relatively simple objects but appears rather complex for more elaborate objects.

3.1.3. R. A. Lorie et al[28,42,43] have extended System R[2] by providing system defined complex objects. They provide three new column types for relations: IDEN-

TIFIER, REF, and COMP_OF. The main features are:

3.1.3.1. The value stored in a column of one of these types is always the IDENTIFIER of some object.

3.1.3.2. IDENTIFIERS are byte strings of uniform length, are unique system-wide, and are generated by the system at tuple creation time. Identifiers are never reused.

3.1.3.3. COMP_OF type signifies that the tuple belongs to the object(tuple) whose IDENTIFIER value corresponds to the value of the COMP_OF column. No cycles are allowed in the tables related by COMP_OF.

3.1.3.4. The REF type is provided for linkages other than hierarchy in a complex object. REF relationships are legal between component tuples of the same object and between component tuples of one complex object and root tuple of another. REF is not allowed between component tuples of two distinct complex objects.

3.2. Multiple Representations

A design may be described by many alternative representations. For example, in VLSI a chip can be described by layout geometries, stick diagrams, block diagrams, logic diagrams, transistor networks, functional specifications, behavioral specifications or just as programs which simulate the functions of the chip. The design DBMS should provide support for such vastly different representations of an object and still retain the vital fact that they are all incarnations of the same object. This is roughly equivalent to the concept of views in conventional DBMSs which are provided for reasons such as security etc. In the design environment multiple representations are meant for separate design activities rather than for privacy and security reasons. It is desirable that the DBMS should not impose an a priori assumption on the various kinds of design representations supported. Rather, it should support multiple representations without it understanding their internal structure. Higher level software should be responsible for choosing the form of the representation and its interpretation[34]. Such a feature provides both power and flexibility. One of the scenarios proposed[14] is that each representation is under the control of a subsystem. All the subsystems use the basic data which deterministically describe the object being designed. Each subsystem utilizes a different model of the object which is more natural to the representation it is controlling. For example, in VLSI design, a simulation subsystem will view a particular device as a simulation ROM with the specific signals appearing on specific input and output pins; a logic diagramming subsystem will model the same device as a set of symbol drawing primitives with the specific signals appearing at specific coordinates on the symbol; and a wire wrap routing subsystem may view the basic data strictly in terms of geometric coordinates. But all the three subsystems will use the same basic circuit data which describe the device.

Nontrivial problems with multiple representations occur with update. When the basic data is updated, the derived data (the various representations) go out of date. To bring them up to date requires long-running programs which may not be run automatically since they may require user input. For example, if a signal is changed the routing and test generation subsystems must be rerun. The cost involved in synchronizing representations with the basic data may be prohibitive and some compromise must be reached regarding the tolerance period during which the representations can be out of sync. More serious problems occur when the representation is updated and hence the basic data goes out of sync as in the situation when a wire wrap connection is changed. To understand the issues involved the following analogy might help. A compiler accepts a symbolic description (the basic data defining the program) as input and derives another representation in terms of machine code (derived representation). If we change the machine code, what is the effect on the symbolic representation? The changes made to one representation can not be propagated by the DBMS to the basic data without an understanding of the semantics of the representation. Related issues in the updating of views in a relational model is discussed in[18] . An easy solution is to prohibit updates to the representations. But this does not seem to be natural in a design environment where the different representations are used for separate design activities. A more practical solution is to allow updates to representations and attempt to flag the discrepancies and try to isolate the update.

3.3. Version Control

As we mentioned in chapter 2, the design activity is not a once-through sequence of actions but is most often iterative. Our model of the design activity shows this explicitly (refer to the discussion of versions in chapter 2). The nature of the synthesis task suggests that the design activity is highly tentative. Thus these two aspects, iterativeness and tentativeness, result in more than one realization of the object which the designer is trying to evolve. Versions can occur in a design environment in basically two ways[15]. First, a version can arise due to the use of different strategies to realize an object. This usually happens during the synthesis stage of the design activity. These versions are alternative design specifications satisfying the abstract functional requirements of the object. Thus these versions are considered functionally equivalent. For example, a software module can be implemented in many ways each representing a functionally equivalent version. The second kind of versions is due to revisions and is the most common one. The old information about the object is changed because it is either incomplete or incorrect. In the context of our model of the design activity, an incorrect design is one which does not meet the design specifications generated by the higher level activity (i.e the design which does not satisfy the inner loop) and an incomplete design is one which does not pass the comparison with the design goal done by the higher level activity (i.e the design which does not satisfy the outer loop). As mentioned in chapter 2, in practical circumstances these designs can be released as finished designs. Also, these versions are usually retained to provide a history mechanism so that the designer can review his earlier work. These two kinds of versions are different since in the former case the versions are functionally equivalent whereas in the latter case they are not. A design DBMS should provide explicit support for both kind of versions. There should be facilities to mark a particular version as 'Released' or 'Verified and Approved' or 'Unupdatable' etc.

3.4. Configuration Control

Humans can effectively deal with only one problem at a time. This general and practical observation has led to the concept of abstraction. Usually the design activity proceeds in different stages of abstraction. Each stage is configured using the objects which are at a more detailed stage of abstraction. Such an abstraction configuration is usually a tree (hierarchy) or a directed acyclic graph. A node represents an abstraction. Examples of nodes in VLSI design are chip, ALU, Shift Registers etc. The node is connected by directed arcs to those nodes which take place in its configuration. Those nodes from which no arcs emanate are the leaf nodes and for the purposes of design are considered available. The information content of each node is composed of the information pertaining to that node plus the information content of those nodes to which it is connected. These are explicitly shown in our model of the design activity (refer to the discussion of abstractions in chapter 2 and fig 2.3). The concept of configuration is simple if there is exactly one version for each node. Then the problem reduces to controlling a complex object. But usually this is not the case as mentioned previously. The DBMS should provide facilities for specifying precisely which versions participate in a particular configuration. So the configuration specification facility should include version specification as well. Such a facility can be helpful in determining the optimum configuration by simulation since a realization of a node which is considered bad singly can in practice be really good as part of a whole object. Also, the ability to control configuration should span across the multiple representations of the object. Thus each representation has multiple configurations.

These three related issues, namely, multiple representations, version control and configuration control are the source of most of the complexity facing the developers of design software. Explicit support from the DBMS to manage these issues in an interrelated manner can be highly useful in breaking the complexity barrier.

3.5. Support for Derived Data

Derived data are those data which are derived from the basic data describing the object being designed. For example, the series of input and output vectors generated by a test generator and the output of a simulator are examples of derived data. Derived data can be present for the low level parts too. To calculate the capacitance between two elements one needs to know their overlap information. This can be derived from the geometry of the elements. Derived data is present in commercial data bases as well. For example, the number of items on order (derived data) equals items ordered minus items received (primary data). Derived data can be stored in the database as a pair consisting of an algorithm and the conditions under which the algorithm is to be executed, or it can be stored as pure data (or as a combination of both). The choice of which method to use depends on the frequency of use of the derived data (the latter method is preferable if the frequency is high), on the frequency of update to the data on which the derived data depends (if the update frequency is high then the former method is better), the volume of the data (if the volume is high then the former method is preferable) and the cost of computation of the derived data (if too much computation is involved then the latter method is better). The test generator output is better stored in the data base in pure data form, whereas it is desirable to run the simulator whenever the simulation data is required. Sometimes it may be beneficial to implement both of the methods.

3.6. Conversational Transactions

In a multi-user environment (as is the case in a design environment) the concept of transactions is very important. A transaction is the basic unit of consistency and recovery. Between its start and end, a transaction executes many read and write operations on the data in the database. Transactions in conventional database applications are generally considered non-conversational; the transactions involve only a few records, typically last for less than a second and do not involve interaction with the user (even though they might have been issued interactively by the user). Transactions of such a nature do not adequately model the interactions encountered with the data in design applications[35,42,53]. In design applications, manipulation of the data is mostly done interactively. The designer "checks out" part of a design (usually a collection of related records), works on it extensively over a long period of time (typically hours and at times days) and finally submits ("check in") the updated version to the database. During this period this part of the design may be inconsistent with other parts of the design. Also, it is highly desirable that the user sessions survive system crashes. This difference in the nature of transactions calls for a difference in the way locks are applied to pieces of information when multiple users try to access the database. It is not acceptable to allow one user to acquire an exclusive lock since other users could suffer intolerable response time on their transactions. A scheme suggested by R. A. Lorie et al[28,42,43] involves the use of private databases. When a designer wants to check out an object from the database the object is copied to the designer's private databases. He works on this private database and when he is done he checks in at which time the object replaces the old version or creates a new version of the object. During check out the object can be locked for read or write. Other designers can still access the old object, for read alone if it is 'write' locked or for both read and write if it is 'read' locked.

Regarding recovery, the usual recovery methods employed by DBMSs are not suitable for design applications. They undo the effect of incomplete transactions and if this is done in a design environment hours and even days of useful work would be lost. The database should be restored to the most recent state possible. This can even be past the last 'save' done by the designer. The information on object locks should be stored in non-volatile storage so that user sessions can survive system crashes.

3.7. Integration with User-Interfaces

As identified in chapter 2, the user interface is an important element in a design environment. It gives the user a human-oriented representation, and facilities for manipulation, of the formalized model of the design object. The practice of handcoding user interfaces will die out as User Interface Management Systems(UIMSs)[25], like the University of Alberta UIMS[26], become more popular. This has created an additional burden (which did not exist when user-interfaces were handcoded) for the designers of data base management systems. Design database management systems should now provide features for easy and successful integration of the DBMS with the UIMS. Cooperation between both systems is necessary for a successful integration. The concept involved is simple, namely the internal (database) representation of the formalized model should be transformed to its external (human-oriented) representation and vice versa. But the ease of implementation of this concept depends equally on both the systems.

3.8. Speed of Internal/External Transformation

As we mentioned in the last section, the database and the user interface participate in the Internal/External transformation and vice versa. The speed of Internal/External transformation is especially critical since if the speed is too low it may cause the designer to be idle for some time. This might affect the productivity and creativity of the designer. The speed of the user interface is a factor determining the response time along with the speed at which data is retrieved from the database. Some requests from the user may involve traversing practically the whole database and this is a time consuming operation. This calls for efficient data access mechanisms and database traversal mechanisms. Certain other factors which are discussed in other sections of this chapter: set oriented information storage and retrieval, keeping a defined set of objects in main memory etc., can contribute to this fast Internal/External transformation.

3.9. User-Controlled Archiving of Data at Object Level

In a design environment the design activity can be so high that it produces a large amount of data. In particular, the graphical information associated with a design tends to increase the amount of data. It is common observation that there are periods of inactivity for some of the data, therefore, it can be archived to more economical secondary storage. It is not advisable to archive on a disk basis; the user will not know what is being archived since it is the DBMS which maps the data to physical locations onto physical storage. It is highly desirable that the user (because he is the person who knows) be provided with the facility to move a meaningful portion of the database to the archive database. Here again the concept of complex objects is highly useful.

3.10. Ability to Handle Heterogenous Data

Present day data models support only homogeneous data (formated data). But in a design environment there are situations where one has to represent and store heterogenous data. Examples of such data include images, text, miscellaneous facts, vectors and matrices. The interface to data should be expanded to accomodate such data. In existing systems if one selects a record and accesses a field in it then this whole field is returned. This is not a desirable policy since the field can be very long, say, 10 Mega bytes. With such a long field it may even be impossible to return the whole contents. The DBMS should provide facilities whereby one can access such long fields piece-wise. An indicator (or cursor) can be initialized as a result of a retrieve operation on such a field. The pieces should then be accessible by moving the cursor along the length of the field.

3.11. Repetitive Access to Data

This is an efficiency issue that can not be ignored. It is a general experience that design applications often need to access data repetitively while executing time-consuming algorithms. When data structures are built in main memory the overhead of accessing a record is far less than the overhead for accessing the record from disk. For example, a placement program will frequently need answers to the questions such as "What are all the signals touching each device? What are all the devices touched by each signal?" and if the algorithms to answer these questions address a disk structure, the total execution time would be astronomical. Another occasion where the main memory structure is better than the disk structure is when derived data is calculated. Therefore, for complex algorithms, one should have a way of bringing logically related records into main memory. It is also a fact that in design applications accesses to data are localized rather than being random. This fact can be used to determine which portion of the database should be brought into main memory.

3.12. Reusability of Previously Specified Information

The use of such a facility is obvious. The DBMS should provide facilities to support object libraries. The library objects should be parametrizable so that they can be used over a wide range of applications. The library objects should be described by their functionality and interface, and not by their realization. The DBMS should distinguish between a linked library object and a copied library object. In the latter case the realization of the object is modifiable by the designer whereas in the former case it is not. If the realization of the linked library object is changed then the change should be reflected in the instances used by the designer, but this will not be the case with copied library objects. If the designer changes the realization of the copied library object and wants it stored in the library, the librarian module of the DBMS should take care of properly versioning the new realization. Thus this issue is closely related to version and configuration control.

It is generally felt that "reinventing the wheel" is quite common in a design environment even where libraries are present. Weinberg[58] noted that the problem with libraries is that "everyone wants to put something in, but no one wants to take anything out." These are caused by unusable or cumbersome access methods, inadequate documentation, poor interface definition of the objects (and hence poor adaptability) and non-availability of intelligent browsers for the library.

3.13. Dynamically Evolving Structure of the Database

It is rightly said that the main purpose of a design database is to be built whereas the main purpose of an administrative database is to be consulted. This calls for a dynamic schema facility. The requirement for dynamic schema arises from the fact that the designer is trying to design complex objects such as buildings, ships process plants, air crafts or digital circuits which does not exist yet in the real world. The designer is constructing the structure of the model itself in addition to specifying data about the model. In the context of our model, both the inner loop and outer loop can cause structural change to the model. The necessity to update the structure of the model is triggered by diverse factors like results of performance measurement (say, by simulation), imagination and reasoning, change in approach, change in technology etc. and as a result can not be anticipated in advance. So the DBMS should allow dynamic restructuring of the database without much cost and performance penalty.

3.14. Constraint Specification Facilities

In business environments, constraints are limitations on the values that the database can have. They are in general called data integrity constraints[46]. This is true of design environments as well. In addition, since the structure of the database can change dynamically we need to specify limitations on the structure of the database too. This calls for new constraint specification mechanisms. Also, since the nature of the design activity is highly tentative as discussed in chapter 2, the data generated by the designer may sometimes violate some of the constraints even though the final product should comply with all the constraints. It will be inappropriate to refuse update to the database when some "non-crucial" constraints are violated. The meaning of the term "non-crucial" is highly subjective. It may happen that a "crucial" constraint can become "non-crucial" and vice versa as the design proceeds. This calls for flexible constraint specification facilities. One of the solutions can be to ask the designer at each violation whether the violation is crucial or not. Non-crucial violations can be recorded and later, on the designer's request, the state of the database can be compared against these violations to report the ones which still exist.

For such solutions to be easily incorporated into the design software and for easily expressing the constraints on the structure of the database, the concept of constraint specifications should be generalized to include the specification of actions[40]. An action will be performed by the system whenever a specified condition occurs. Some researchers call such specifications "Data-dependency Specifications". With this notion of constraint specifications, the concept of derived data, which we discussed earlier, can be equivalently expressed as a constraint specification since the value of the derived data depends on (constrained by) the values of some other data. Such constraints (or data-dependencies) can be enforced by the concepts of triggers[19], production rules[20] or alerters[10]. For example, Garrett and Foley[20], use production rules to enforce data-dependencies. A production rule is an assertion and an action which is performed when the assertion is true. In their data-dependency view of the organization of a graphics application program, they introduce four sets of data-dependencies. There are data-dependencies between graphical input data and graphical output data (for feedback purposes), between graphical input data and application data, within application data (for propagating updates to other dependent data and for enforcing application-dependent design constraints) and between application data and graphical output data.

Thus design database management systems should provide such flexible facilities for constraint specifications.

3.15. Set Oriented Information Storage and Retrieval

Retrievals from a design database are typically set-oriented rather than record at a time[5]. Since the designer is interested in a particular object (usually quite complex) which is composed of highly interrelated records, retrieving pieces of it is not usually very useful. This fact can be used to localize the storage of information which will improve access efficiency considerably. The relational model is set-oriented in the sense that it provides retrieval operations on relations which are sets of tuples. So relational DBMSs usually try to localize the storage of tuples of a particular relation. But a design DBMS should try to localize highly related records. Here again, the notion of system defined complex object is very useful. Since most of the time the designer will be interested in a particular complex object, the storage of records of a complex object can be localized so that the access efficiency is improved.

3.16. Complex Computations

Many CAD applications require both large amounts of data and extensive scientific and engineering computation. Existing DBMSs are usually limited to simple computations and statistical summarizations. The design DBMS should provide facilities for a carefully selected set of engineering and scientific computations. More important than that is the flexibility for inclusion of new computations so that the DBMS can be tuned for specific design domains. Such facilities can greatly reduce the development time of application software (the design tools).

3.17. Design data vs Engineering data

Design automation data can be divided into two types, design data and engineering data. Design data is the set of data which describes the object being modeled. Engineering data is the set of data which is utilized by application programs as a source of information but is not modified by them. Examples of engineering data include various constants like wire thickness, pad size, other engineering constants etc., data pertaining to code of practice, blockage maps for cell routers and symbol descriptions for graphics applications, design rules etc. The structure of the engineering data can be as simple as a list of name-value pairs (as in typical engineering constants) or as complex as geometrical modeling schemes (as in symbol descriptions for graphics applications). Major consideration should be given to providing ownership and control tools for the two types of data in addition to providing an easy and flexible interface to the two types of data. The responsibility of the design data falls on the users (designers) who are authorized to carry out the design. In this case, the users should be given tools with which they can effectively manipulate their portion of the design database. Engineering data is generally maintained and controlled by a small group and is used by all the designers. The management of engineering data involves two major issues: integrity of engineering data and controlling of the impact of change to engineering data. In this case, the small group which manage the above two issues should be provided with configuration management tools.

3.18. Support for both design and corporate databases

It is desirable to merge the computer aided design activity more into the full life cycle of the object being designed. Taking the present day situation in design and manufacturing environments, the CAD activity generally ceases when the design data and documentation leaves the domain of the design database. As a result of this, data which was present in the design database is often duplicated in many places like where-used and bill-of-materials databases. Thus design automation is so far merely a useful front end to the complete life cycle of the product. A sophisticated database management system should provide facilities so that portions of the design database and relevant portions of the corporate databases can be meaningfully merged. This can be as simple as storing data like contract name, manufacturing machine name and location, mill specifications etc. along with the design data, or can be as complex as defining meaningful interfaces (windows) between the design database and various other databases with controlled data redundancy. In a real-life situation, this can have impacts like reduced freedom for design database managers; tools can not be built to suit the purposes of just designers, certain fixed and standard protocols must be agreed upon with the other managers who are active users of the database.

3.19. Support for Documentation

Good documentation can aid in getting the proper perspective to understand a particular piece of design. Carlo Sequin[52] is of the opinion that "Design and Documentation should become synonymous". The design DBMS should provide explicit support for documentation of design objects. Documentation is best arranged as a hierarchy. The DBMS should provide facilities so that the documentation provided for multiple representations of the same object can be treated as a whole. Automatic documentation facilities should be provided wherever possible.

3.20. Requirements Considered for Design

So far, we have presented a comprehensive discussion on the desirable features of a design database management system. We do not claim that the requirements cited in this chapter are by any means exhaustive. Also, more experience with design systems can suggest further desirable features.

It is nontrivial to develop a database management system which can provide all the features expected by design applications. In fact, it is debatable whether some of these features should be provided by the DBMS or by software external to the DBMS. Several compromises have to be reached and several design decisions have to be evaluated under practical circumstances. As an aid to studying some of the important requirements we designed and implemented a prototype database management system. The details of its design and implementation are described in the following chapters. For the purposes of this design, we chose a set of features which we considered important and which could be implemented in the limited period of time available. The following are the features that were chosen.

1) Representation and Manipulation of Complex Objects.

2) Dynamic Restructuring of the Database.

3) Support for Derived Data.

4) Flexible Constraint Specifications.

We chose the issue of representation and manipulation of complex objects for two reasons: First, it is important by itself as we have discussed before in this chapter. Second, many other requirements like archiving of data at object level, set oriented information storage and retrieval, configuration control and object libraries are related to the concept of complex objects and they can be provided later using the support provided by complex objects. The importance of the other three requirements that were chosen, has already been discussed in this chapter. We chose these three requirements for the following reasons : First, we felt that the facilities to satisfy these requirements make the database management system complete in the practical sense and at the same time do not make it complex and unmanageable. Second, we felt that it would not be very difficult to design the facilities to satisfy these requirements and that we would not face any hard implementation problems.

There are other equally, if not more, important requirements which we did not consider at present in order to keep the first version of the system simple and manageable. One of these requirements, namely, "multiple representations" is a complex issue by itself and we have decided to investigate it at a later time. We will revisit the subject of design DBMS requirements in chapter 7.

3.21. Summary

In this chapter, we have discussed the features that the design applications expect of a database management system. We have selected four of these features for the purposes of designing and implementing a prototype database management system. One of those features, namely, the representation and manipulation of complex objects can act as a foundation for building a host of other features. Subsequent chapters describe the design and implementation of the prototype database management system.

Chapter 4

An Object Oriented Design Database Management System

4.1. Introduction

In this chapter we present the design of a Database Management System that provides the four important features that are mentioned in Chapter 3. To recapitulate, the main objective of the design is to provide the following features:

- Representation and Manipulation of Complex Objects.
- Dynamic Restructuring of the Database.
- Support for Derived Data.
- Flexible Constraint Specifications.

We first describe the structural aspects of our system. This includes a detailed discussion on how relationships are modeled and how complex objects are represented. This is followed by the details of the constraint specification facilities provided by our system. We then present the various functions provided for the purposes of data definition and data manipulation. Finally we present the details of the data definition language, called "Data and Constraint Specification Language"(DCSL), which is used to specify the initial state of the database.

4.2. Overview

4.2.1. Database as a model

A database is a model (or representation) of some real world phenomenon[47]. To explain what this really means, we take a short detour via more fundamental concepts on modeling in general and data modeling in particular. This brief discussion is aimed at putting in proper perspective the concept of a database as a model.

There are three realms of interest in the philosophy of information. These realms are the real world, ideas about the real world existing in the minds of humans, and symbols on paper or some other storage medium representing these ideas[1]. The information content in each of these realms differs subtly and significantly from one to another. The fundamental assumption is that the first realm, namely, the "real world" exists in some meaningful sense. No one in the scientific discipline claims to have knowledge about its structure and information content, if at all they are distinct. This "true reality" is "perceived" by humans through the sensory inputs and is transformed by the brain. These perceptions can be transformed into a "perceived model of reality" by a process known as scientific abstraction. This "perceived (mental) model of reality" is the second realm. For all practical purposes we can ignore the first realm. So, hereafter the term "real world" actually refers to the second realm. The third realm comes into picture as a result of the desire to communicate this mental model to someone else or to "something" else. Usually, this communication, again, involves abstraction. While it may be true that the universe is "best" described by the complex interaction of 3.10 raised to the power 81 quarks, the typical engineer abstracts a machine to a combination of shafts, wheels, rivets etc. Different people abstract the "perceived reality" to different levels of detail based on their interests and relevance to their present task. A molecular biologist's view of a human being will be as a complex structure of water, protein molecules, DNA and other assorted chemicals whereas an insurance agent (as far as his professional activities are concerned) may view a human being as not much more than an age, sex, previous health history and checkbook. The third realm is concerned with this abstracted model of the "perceived reality". *The model of the design activity which we portrayed in Chapter 2 is thus a model of the process by which the second realm is abstracted to form the third realm.*

The concept of Database can be described as one of the computerized implementations of the third realm. Thus the database is an abstract model of the perceived reality. The term Database Design refers to the process by which the structure of the second realm is abstracted and transformed into the structure of the third realm. The term Database Processing refers to the process by which the contents and actions of the second realm are abstracted and transformed to the primitives of the third realm.

The above discussion on modeling concepts provides an explanation of the meaning of database from a modeling point of view. We saw that database design and processing is essentially a modeling process. A database model provides the modeling primitives that are used in this modeling process. It is essentially a vocabulary for describing the structure and processing of a database. The part of the vocabulary which is used for describing the structure of the database is called the "Data Definition Language (DDL)" and the part of the vocabulary which is used for describing the processing of the database is called the "Data Manipulation Language (DML)". The database management system is the software which implements the DDL and DML.

4.2.2. Overview of our System

In our system, there are four modeling primitives that are given to modelers. They are "database", "object classes", "objects" and "properties". Objects model the real world entities and properties model the characteristics of the real world entities. Objects can be grouped together into "Object Classes" on the basis of similarities (a process called "generalization"). This is left entirely to the user and there is no constraint imposed by the system on the structure of the objects of an object class except for the restriction that an object can be a member of only one object class. The database is a set of object classes, an object class is a set of objects and an object is a set of properties. The number of object classes in the database, the number of objects in an object class and the number and the values of properties in an object can change over time. These basic modeling primitives, namely, Database, Object, Object class and Property are collectively called "*Modeling Objects*". The hierarchy of the modeling objects is depicted in fig 4.1.

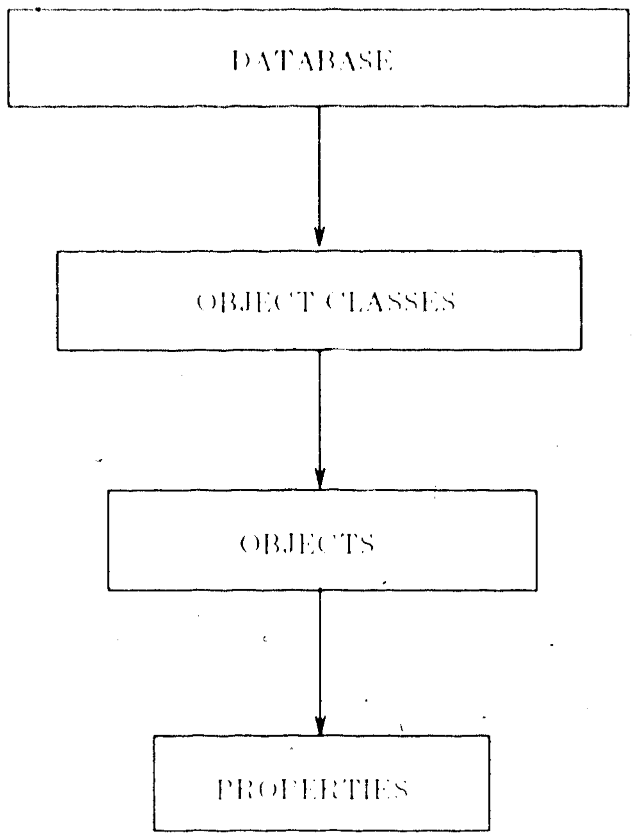

Figure 4.1 Hierarchy of the Modeling Objects

The rest of the chapter describes the design in detail.

4.3. Database

The database models some real world phenomenon. It represents those facts of the real world phenomenon that are of interest to the modelers. In our system, the database is a set of "Object Classes".

4.4. Objects

Objects model real world entities. A real world entity can be a person, place, thing, concept, or event (real or abstract) that is of interest to the modelers. An object is uniquely identified by a system generated name. Objects contain information on the real world entities and on the relationship between the real world entities. An object is similar to a tuple in the relational model or a record in the CODASYL model. The unique name of the object is similar to the concept of the primary key. An object consists of properties which model the attributes (characteristics) of the real world entities.

4.5. Properties

As we mentioned earlier the attributes of the real world entities are modeled by their properties. A property is typically used to model the following aspects of an entity: identification of the entity, characterization of the entity and relationship of the entity with other entities. Each property consists of a name (which is unique within an object), a value and its type and three actions, namely, IFNULL, WHEN-MODIFIED and WHENDELETED.

The types of properties can be INTEGER, REAL, TEXT, OBJECT and SET. The types INTEGER, REAL and TEXT (a string of characters) are the conventional types and need no explanation. A property of type SET can have a set of objects as its value. The type SET is the basic means of expressing relationships between objects. Thus properties of type SET are termed "Relationship Properties". A property of type OBJECT can have another object as its value. Optionally, an object class name can be specified for the properties of type OBJECT and SET. The use of this specification is explained later in this chapter when we discuss the constraint specifiction facilities of the system.

The type of a property cannot be changed. If the type has to be changed, the property should be deleted from the object and then added to the object again with the new type.

The three actions IFNULL, WHENMODIFIED and WHENDELETED are used for constraint specifications and are explained later in this chapter.

Properties are of two kinds, namely, Primary Properties and Derived Properties.

4.5.1. Primary Property

Primary properties are those fundamental properties whose values can not be derived from the values of any other properties. The dimensions of a chip, the name of a module are examples of primary properties.

4.5.2. Derived Property

Derived properties are those properties whose values can be derived from the values of other properties. The area of a chip (since it can be derived from the dimensions of the chip) and the number of transistors in a chip are examples of derived properties. Derived properties have the following advantages: the responsibility of providing the value of the property is taken away from the user; the value of the property is automatically constrained against invalid data. In the case of a derived property, its value is represented by a piece of code and its type is the type of the value returned by that piece of code. Whenever the value of the derived property is accessed, that piece of code is executed and the value returned by that piece of code is returned as the value of the property. The execution of the code each time the value is accessed can be optionally suppressed. In this case the value of the property is the value returned by the previous execution of the code.

Another use of derived properties is that a limited support for non-updatable "views" of objects can be achieved through these properties. For example, if a particular tool is interested only in the wire information of a chip object; an object, say, "chip-view", can be defined with one derived property "wires". The piece of code which derives the value of this property can access the wire information from the chip object and return this information as the value of the derived property "wires" of the object "chip-view".

4.6. Object Class

An object class denotes a set of objects. An object class is uniquely identified by a system generated name. The structure of an object class is same as the structure of an object in the sense that an object class also consists of a set of properties. Some properties of the object class are inherited by its objects at object creation time. Since the properties of an object class are used for defining the structure of its member objects, these properties are sometimes called "definitional properties" in constrast to the term "factual properties" which refers to the properties of the objects[48].

Every database design involves the process called "Generalization" whereby we ignore the differences in objects and combine them into a single category[54]. When we lump objects together we do so on the basis of similarities. For example, transistors have enough similarities to distinguish them from, say, resistors, signals etc., but they do have differences in, say, polysilicon size, diffusion size, the location to put the transistor etc. This similarity is captured by the concept of an "Object Class" (say, TRANSISTOR), and the differences are captured by the properties of individual objects of the object class. The process of finding the similarities (i.e generalization) is highly application dependent (i.e dependent on the real world phenomenon that the database designer wants to model). Hence it is desirable that as much flexibility as possible is provided by the system for this "generalization" process.

In our system, the objects under an object class need not have the same set of properties; even the types of properties of the same name can be different. The main idea behind the concept of object classes is to group objects of similar structure under a common name. The concept of object class helps to easily define and initialize the properties of objects of similar structure. All objects belong to some object class or other. The default object class of an object is 'SYSTEM'. The object class of an object can be changed at any time.

At object creation time, the object inherits all the properties, their initial values and the constraint specifications, if any, of its object class. Subsequently, the values of the properties of the object can be changed, new properties can be added to the object and the inherited properties can be deleted from the object. After the creation time the modifications done on the object class do not affect the object, and vice versa. When the object class of an object is changed, there is no effect on the structure of the object or on the values of the properties of the object. If a common operation on the members of an object class is to be performed, say, updating the value of a property or adding a property, a function provided by the system called VisitObjects, can be used. This function, along with other functions provided by the system, is described later in this chapter.

For example, the structure of transistors in a circuit would be almost identical to each other, except that some transistors might have more or less properties depending on their application. An object class TRANSISTOR can be defined. An individual transistor can be first defined by specifying that it belongs to the object class TRANSISTOR. It will inherit all the properties, their initial values and constraint specifications, if any, of the object class TRANSISTOR. Then its property set can be altered or the default initial values can be changed as necessary.*

An object class can be defined using another object class. For example, if object

* In systems where the object class is homogeneous, i.e. where the structure of the objects in the object class is the same, the collection of objects in an object class is sometimes called the *extension* of the object class [18].

class A is defined using the object class B, then the object class A inherits the properties, their initial values and constraint specifications, if any, of the object class B. But the object class A does not inherit the objects of object class B and there is no association between the objects of object class A and object class B. After the definition, the two object classes are independent of each other. This facility is provided to ease the task of the definition of an object class whose property set is a superset or subset of that of some other object class.

The concept of object class definition is roughly similar to the concept of 'Record Definition' in the CODASYL model or 'Relation Definition' in the relational model. All of them are used to define the structure of a set of similar real world entities. The similarity ends here. As mentioned before, the objects under an object class need not have the same set of properties; even the types of properties of the same name can be different. We also provide the facility to change the object class of an object. Unlike the CODASYL and relational model the concept of relationship does not involve object classes. This is due to the way in which we view the concept of object classes; object classes facilitate object definition and creation and group similar objects under a common name.

The type of inheritance we have adopted is generally called "static inheritance" because the inheritance is done at object/object class creation time. After creation the modifications done to the parent object class do not affect the objects and object classes which are created using the parent object class. This is in contrast to "dynamic inheritance" where the changes made to the parent object class are reflected in the objects and object classes created using the parent object class.

4.7. The Data Type SET and Relationship Properties

A property is a relationship property if its type is SET. The type SET defines the powerset of the set of all objects. Thus the property of type SET can have a set of objects as its value. Relationship properties are the basic means by which relationships between objects are established. Also, these are the means by which complex objects are constructed and manipulated.

The definition of relationship involves the specification that the type of a property is "SET" and optionally, the specification of the name of an object class. Later when objects are included in the set (i.e when the value of the property is updated) the relationship between the owner (of the property) object and the member object is established.

4.7.1. Modeling Relationships

With the primitives mentioned above, it is interesting to note how concepts like 1:N and M:N relationships are modeled in our system. We will use the following example of a university database involving UNIVERSITY, DEPARTMENT, PROFESSORS, STAFF and STAFF_UNION to illustrate the modeling of relationships. There is a 1:N relationship (pure hierarchical relationship) between a university and its departments since a department can belong to only one university. There is an M:N relationship (complex network relationship) between departments and professors since a department can employ 0 or more professors and a professor can be employed by 0 or more departments. There is a 1:N relationship between a department and the staff, and there is a 1:N relationship between a staff union and the staff. This is because a staff member can not work in more than one department and can not be a member of more than one staff union. Staff is said to be in "simple network relationship" with the department and the staff union. The relationships are depicted in figure 4.2. If a staff member can be a member of two staff unions (for our example, we have assumed

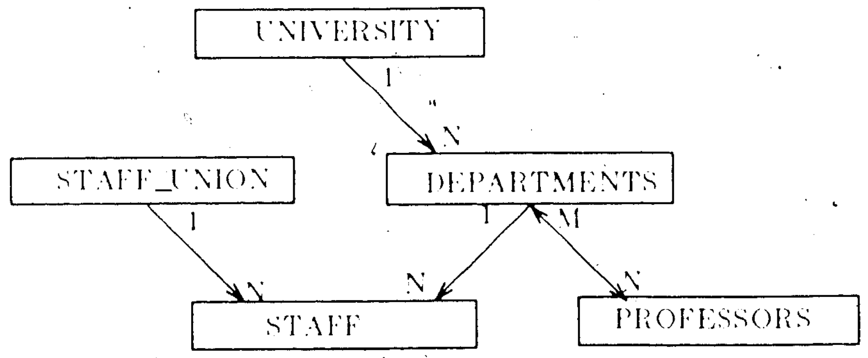

Figure 4.2 Structure of a University Database

that it does not happen) then the simple network relationship breaks down; the relationship between the staff and the staff union then becomes an M:N relationship, i.e a complex network relationship

The way these relationships are modeled in our design is different from other database models. With other database models, the definition of the relationships involves the object classes (or analagous concepts) of the partners of the relationship. For example, in the CODASYL model, to model the 1:N relationship between DEPARTMENT and STAFF, a SET has to be defined with the DEPARTMENT record *(object class)* as the owner and the STAFF record *(object class)* as the member. In such definitions, object classes act as a typing mechanism on the partners of the relationship. In our system, the concept of object class is not involved in the modeling of the relationships. As will be explained subsequently, the name of the relationship property plays an important role in this process. We discuss below the key concepts associated with relationship properties. This helps to explain as clearly and as unambiguously as possible how the various kinds of relationships can be modeled.

Let us assume the following for illustration purposes: there is a university object U with a relationship property "departments"; there are department objects D1 and D2 with the relationship properties "professors" and "staff" and they are members of the relationship property "departments" of U; there are professor objects P1, P2, P1 and

P2 are the members of the relationship property "professors" of both D1 and D2; there is a staff union object SU1 with the relationship property "unionmember". There is a staff object S1 which is a member of the relationship properties "unionmember" of SU1 and "staff" of D1.

4.7.1.1. The value of a relationship property can be a set of objects. This means that the set can contain objects belonging to different object classes and can contain objects with totally different structures.

4.7.1.2. The set is uniquely identified by the pair *(object name,property name)*. The first name is called the *"owner" of the set and the second name is called the "tag" of the set.* The set, as identified by the pair, is called the parent of its members. For example, consider the set of the department object D1 uniquely identified by the pair (D1,"professors"). D1 is called the owner of the set and "professors" is called the tag of the set. The set (D1,"professors") is the parent of the professor object P1.

4.7.1.3. An object can be a member of two (or more) sets (identified by different tags) owned by the same object. In that case, the object is considered to have two (or more) different parents. In our example, there is no such instance.

4.7.1.4. An object "knows" to which set(s) (as identified by the pair) it belongs to. Thus the professor object P1 "knows" that it belongs to the sets (D1,"professors") and (D2,"professors").

4.7.1.5. We say that an object is in *"SIMILAR RELATIONSHIP" through a tag* with two (or more) objects if it belongs to two (or more) sets with this tag, owned by these objects. Thus professor object P1 is in *"SIMILAR RELATIONSHIP"* with department objects D1 and D2 through the tag "professors". It should be noted here that the structures of D1 and D2 can be different and they can possibly belong to different object classes. We say that an object is in "*NONSIMILAR RELATIONSHIP*" with one or more objects if it belongs to two (or more) sets, owned by these objects, with different tags and if it is not in similar relationship with any objects through any of these (different) tags. The staff object S1 is in NONSIMILAR RELATIONSHIP with the staff union object SU1 and the department object D1. This will not be true if S1 is in SIMILAR RELATIONSHIP with other objects either through the tag "union-member" or "staff". It should be noted that an object can be in nonsimilar relationship with a single object. But this is not true of similar relationship.

With these concepts, it is possible to describe the three general notions of relationships. A pure hierarchy (1:N relationship) is achieved when one or more objects are members of the same set and they are not members of any other set. The set (U,"department") with the members D1 and D2 is an instance of a pure hierarchy. We get a simple network relationship (i.e two or more pure 1:N relationships) if an object is in NONSIMILAR RELATIONSHIP with other objects. The sets (SU1,"unionmember") and (D1,"staff") with their (common) member S1 constitute an instance of a simple network relationship. We get a complex network relationship (M:N relationship) if an object is in SIMILAR RELATIONSHIP with two (or more) objects. The sets (D1,"professors") and (D2, "professors") with their (common) member P1 constitute an instance of a complex network relationship.

It can be seen that the name of the tag plays an important role in the modeling of different kinds of relationships which occur in real life situations. Let us take the concept of M:N relationship to explain the role played by the tag. The key point which establishes the M:N relationship between DEPARTMENT and PROFESSORS is the fact that each professor object "knows" all the objects with which it has "SIMILAR RELATIONSHIP" through the tag "professors". This essentially means that each professor "knows" all the departments in which he/she is employed. Thus, *as far as relationships are concerned*, the role played by the tag (the objects having the same tag constitute a set of objects defining one partner of the relationship; in our example, DEPARTMENT) is similar to the role played by the Object Class (or analogous concepts) with other models (in the CODASYL model, the record DEPARTMENT would be used to define one partner of the relationship). Other models assume (implicitly) that objects of the same structure will have the same relationships with other objects, which probably is true for business applications. Our system provides the flexible facility by which situations which do not satisfy the above assumption can also be easily modeled.

The relationship instantiation actually happens when the size of the set *of the owner object* increases from 0 to 1 and not just when the owner object is created. So in our system the conventional differences among relationship definition time, relationship instantiation time and relationship modification time (includes the set member initialization time) have decreased. One of the situations when these three times coincide is when an object is created with a relationship property (relationship definition) and with an initial value for that property (relationship instantiation and relationship modification). Our data definition language, which will be described subsequently, is capable of expressing such situations.

4.7.2. Relationship Data

Sometimes relationships can have some characteristics (attributes). In our example, a professor works in several departments. The salary a professor is paid by a department belongs to the relationship between the professor and the department; it is neither a property of the individual department nor a property of the professor since its *meaning* depends on both the professor and the department. We provide a facility to model such situations. The characteristics of a relationship can be modeled using a special property which comes into existence automatically (i.e. automatically provided by the system) when the relationship is established and continues to exist as long as the relationship exists. The name of the special property is called "SETDATA" which is of type OBJECT. This property does not either belong to the owner of the set or to the member of the set. It belongs to the relationship between the owner and the member. We are restricting the type of this property to OBJECT but this does not sacrifice the generality. The value of this property, viz. any object, can have properties of all allowed types.

4.7.3. Complex Objects

As we mentioned before, relationship properties are the basic means by which the complex objects are constructed. But, it should be mentioned that a relationship property need not necessarily be used for the construction of a complex object. It can be used for other purposes, such as grouping of a set of values under a common name. So, in general, a subset of the relationship properties participate in the construction of complex objects. Thus in the context of complex objects, a reference to the term "relationship property" actually refers to a relationship property from this subset.

4.7.3.1. Representation of a Complex Object

A relationship property basically establishes the connection between a more complex object and a less complex object. The complex object can be depicted as a directed acyclic graph in which the owners and members of all sets are represented by the nodes. The inclusion of an object A as a member of a relationship property of object B is represented as a directed arc from B to A. The tag can be thought of as the name of the directed arc. The information associated with the arc represents the SETDATA. The information associated with the node represents the properties (and their values) that do not participate in the construction of the complex object.

Thus relationship properties are the means by which complex objects are built. Taking the example of the 4-AND gate presented in Chapter 3 (please refer to figure

3.2), the gate object will have relationship properties, say, "geometry", "pins", "blocks", and "connections". The relationship property "geometry" will have a GEOMETRY object as its member, the relationship property "pins" will have PIN objects as its members and so on. An example of a VLSI chip is presented in Chapter 6. This also illustrates how a complex object can be built using relationship properties. It can be seen from these two examples that the complex objects encountered in VLSI design are usually hierarchical in nature.

We provide a function "TraverseComplexObject" for manipulating a complex object as a whole. This function is explained in detail later in this chapter.

4.8. Constraint Specifications

The importance and usefulness of automatic enforcement of constraints on the contents of the data is well understood. The commonly known kinds of constraints are field constraint (i.e. constraint on just the value of a property), intrarecord constraint (i.e constraint among the values of properties of an object) and interrecord constraint (i.e. constraint among properties of different objects and also on the membership of relationship properties of an object)[39]. P. P. Chen[12] reports a classification of constraints which he uses to show how constraints can be specified in his Entity-Relationship model. There can be other constraints which use data that need not necessarily be found in the database. Taking an example from the business environment, there can be a constraint that if an order worth more than a specified limit is received after 4 P.M. on the last day of the week, it should not be accepted. This constraint uses the time of day and the day of the week which need not necessarily be part of the database.

Our view of constraints is general in nature. As a consequence, the primitives we provide for enforcing constraints are very flexible as demanded by the design applications. There are two broad categories of constraints. They are

1) Constraints on the contents of the database.

2) Constraints on the structure of the database.

The necessity to provide constraints on the structure of the database arises from the fundamental fact that we allow modification of the structure of the database.

To determine the database operations that may need constraints, we need to identify what constitute the contents and the structure of a database.

Essentially, the contents of the database is said to depend on the following:

1)The number of objects in all object classes of the database

2)The values of properties of the objects in the database.

3)The existence of the database.

Similarly, the structure of the database is said to depend on the following:

1)The number of object classes.

2)The number of properties in the objects

3)The types of the properties in the objects.

4)The existence of the database.

We can now determine the database operations that may need constraints. The number of objects in the database is changed when an object is created or when an object is deleted. The value of a property needs to be monitored in two situations : when the value does not exist and when the value is updated. The number of object classes is changed when an object class is added or deleted. The number of properties in an object needs to be constrained when a property is added to the object or deleted from the object. The type of a property needs to be monitored when the property is added to the object. Since the type of a property can not be changed, there is no need to monitor the type of a property after the property is added to the object. Of course the existence of the database is to be monitored when the database is destroyed.

We have the following scheme for constraint specifications and enforcement of database integrity. It should be noted that all the actions mentioned below are purely optional.

4.8.1. When the database is created, there are no constraints.

4.8.2. Two actions, namely, WHEN_OBJECT_CLASS_CREATED and WHEN_DATABASE_DESTROYED can be associated with the database. The action WHEN_OBJECT_CLASS_CREATED will be executed automatically by the system when a new object class is created. If it returns YES then the object class is created otherwise it is not created. The default object class "SYSTEM" always exists and hence its creation cannot be constrained. The action WHEN_DATABASE_DESTROYED will be executed automatically by the system when an attempt is made to destroy the database. If it returns YES then the database is destroyed otherwise it is not destroyed.

4.8.3. Two actions, namely, WHEN_OBJECT_CLASS_DELETED and WHEN_OBJECT_ADDED can be associated with each object class. The action WHEN_OBJECT_CLASS_DELETED will be executed automatically by the system when the object class is deleted. if it returns YES then the object class is deleted otherwise it is not deleted. The object class "SYSTEM" always exists and is automatically constrained by the system against deletion. The action WHEN_OBJECT_ADDED will be executed automatically by the system when an object is added to this object class. If it returns YES then the object is added otherwise it is not added.

4.8.4. Two actions, namely, WHEN_PROPERTY_ADDED and WHEN_OBJECT_DELETED can be associated with each object. The action WHEN_PROPERTY_ADDED is executed automatically by the system when a property is added to an object or object class. This action can also monitor the type of the property added. If it returns YES then the property is added otherwise it is not added. The action WHEN_OBJECT_DELETED is executed automatically by the system when an object is deleted. If it returns YES then the object is deleted otherwise it is not deleted.

4.8.5. Three actions, namely, IFNULL, WHENMODIFIED, and WHENDELETED can be associated with each property. The action IFNULL is executed automatically when the value of the property is accessed and is found to be not present (NULL). NULL values should be monitored cautiously. There can be situations where a value of a property can not be NULL and a default value must be provided. Also any two NULL values need not be semantically equal. A detailed discussion of NULL values and the use of the IFNULL action follows subsequently. The action WHENMODIFIED is executed automatically by the system when the value of the property is modified. If it returns YES then the modification is carried out otherwise it is not carried out. The conventional field constraints, intrarecord constraints and interrecord constraints are specified through this action. The action WHENDELETED is automatically executed by the system when the property is deleted. If it returns YES then the property will be deleted otherwise it will not be deleted.

4.8.6. It is interesting to observe the relationship between the modeling object to which an action is associated and the operation the action is intended to constrain. For this purpose please refer to the modeling hierarchy depicted in figure 4.1. If the operation involves addition of a modeling object to another modeling object (i.e creation) then the action is associated with the latter modeling object. Otherwise the action is associated with the modeling object itself. For example, the action constraining the creation of an object is associated with its object class whereas the action constraining the deletion of an object is associated with itself. This relationship can be stated as follows: the actions are associated with the "nearest available ancestor".

considering that a modeling object is the nearest available ancestor of itself after it is created.

4.8.7. The actions return a value and this is the means by the which the constraints are enforced. Also, the actions can cause some side effects. Revisiting our previous example, if an order worth more than a specified limit is received after 4 P.M. on the last day of the week, in addition to denial of acceptance of the order (by not returning a value YES), the action can, say, send a mail message to the supervisor concerned regarding this.

4.8.8. An action is a piece of code in the implementation language (in our case, C). All the action specifications are optional. If the action is not specified then it is equivalent to an action returning a value of YES. There are advantages in expressing an action as statements in the implementation language. First, we achieve full generality. Constraint specifications in addition to ensuring data integrity should be capable of taking arbitrary corrective measures. By expressing the actions as statements in the implementation language, such arbitrary corrective measures can be specified using the implementation language statements and the data definition and manipulation primitives provided by our system. Second, the implementation of constraint specification mechanisms becomes easy. If a separate constraint specification language capable of expressing arbitrary corrective measures is to be used the development, implementation and integration of the language with the system is a much harder task.

4.8.9. The constraint specification mechanism of our system is similar in concept to triggers[19], alerters[10] or production rules[20]. We have already discussed the concept of production rules in Chapter 3 when we described the importance of constraint specifications. These concepts involve the specification of a condition and an action which is executed by the system when the condition becomes true. In our system, the conditions are specified implicitly. For example, the action WHEN_DATABASE_DESTROYED, has an implicit condition specification, say, "if database destroyed". In other words, the conditions are fixed and pre-specified by the system. We do not lose generality since we have a complete set of such conditions that can affect the contents or structure of the database. On the other hand, the advantages are: the user need not specify the conditions explicitly; and the actions are associated with specific modeling objects which is more natural and meaningful.

4.8.10. We mentioned before that an object class can be optionally specified for the properties of type OBJECT and SET. The database designer, if he/she so wishes, can effectively use this object class specification for restricting the value of these properties through the WHENMODIFIED action. For example, the WHENMODIFIED action can impose such a restriction on a property of type SET by returning NO whenever a member of the set does not belong to this object class.

4.9. The uses of the action IFNULL

We mentioned previously that the IFNULL action can be associated with each property. It is automatically executed by the system if the follo[illegible]g condition occurs: on an attempt to access the value of the property, the value of the property is absent (i.e NULL). For a derived property this condition occurs if the action which derives the value of the property returns -1 (the exact details of this are explained in Chapter 6). We will now take a detailed look at the usefulness of the IFNULL action.

The IFNULL action can provide a default value for the property. Provision of default values can be viewe[illegible]nstraint mechanism. If the constraint is that the value of a particular propert[illegible]nnot be NULL, one of the solution can be to deny those updates which violate this constraint. As we discussed in Chapter 3, this solution is not appropriate for design applications. The update can be allowed but the constraint is modeled as the action IFNULL which can either provide a default value, generate a warning message, or take some other appropriate action.

Another important use of the IFNULL action is to interpret the meaning of NULL values. The NULL values are in general a problem to handle and their interpretation can not be generalized and built into the system. In the relational model, due to the lack of a mechanism to interpret NULL values, unexpected and surprising results can occur during join, relational and boolean operations[39]. If a property has a NULL value then it can be interpreted in many different ways depending on the context and hence two NULL values can not in general be considered semantically, logically or relationally equal. First, NULL values of two properties of different types are inherently different in semantics. Second, a NULL value can mean that the value is unavailable or that the value is inapplicable at that time or it can mean many other things. Taking an example from software design, a module imports the procedure, say, AllocateMemory and the module from which this procedure is imported has a value of NULL. This can mean many things including:

1) At the present moment the module name is not known.

2) There are several modules which supply AllocateMemory and a decision on which module to import it from has not been made.

3) At the present moment the procedure AllocateMemory is not needed

4) At the present moment the designer is not sure if the procedure AllocateMemory is needed or not.

Thus the interpretation is very much application specific. The tool developer (or the database designer) is the best person to interpret them, and the interpretation can be expressed through the action IFNULL.

There has to be some mechanism by which the interpretation can be quantified so that it can be made known to those who access the value. Basically, this calls for some conventions to be established. It is very difficult to arrive at a general (across all applications) set of meanings for NULL values. So it will be unnatural to predetermine and build the interpretations of NULL values into the system. The following is the simple convention scheme that we have decided to follow. The "IFNULL" action should return an integer. The value of 0 is reserved for indicating that the value is not NULL. If a value of -1 is returned, then it means that the property does not want its NULL value to be interpreted and is equivalent to the situation where no "IFNULL" action is associated with that property. For relational operations (like equal, greater than and less than etc) on NULL values, the numerical value returned by the "IFNULL" action can be used. All other interpretations are to be agreed upon by the parties involved : the one who accesses the value of a property, the one who supplies the interpretation of the NULL value of the property and the "IFNULL" action associated with that property. The second and third parties will be the same in most of the circumstances. By such agreement, the interpretations can be passed to whoever is interested in them. This is roughly similar to the concept of error numbers that are used by some operating systems to indicate the occurrence of certain conditions. The error number and its interpretation are agreed *apriori* by the user programs and the operating system.

In the example of software design mentioned above, suppose that an automatic design documentation tool exists. The tool can report why (i.e the interpretation) the name of the module from which the procedure AllocateMemory is imported, is NULL. In this situation, the interpretation of this NULL value has to be made known to the IFNULL action by the user (i.e the software designer) so that it can pass the interpretation to the documentation tool. The user should provide the interpretation in this case since he knows what this NULL value means. The way this is provided can take various forms including the following: the IFNULL action can ask the user for the interpretation; or the user can prespecify the interpretation as the value of a property and the IFNULL action can access this value. This is to be decided by the database designer. Whatever way this is done, the documentation tool is quite unaware of this. It only needs to know what each integer value returned by the IFNULL action means.

Another interesting example of the interpretation of NULL values occurs in VLSI design. Let us assume that a cell contains other cells (called "inner cells") and an important property of each cell is its location with respect to the immediately surrounding cell. What are the interpretations of the NULL value of the property "location"? The database designer might think of the following.

4.9.1. The cell is an inner cell and the value of "location" is not known yet. The database designer assigns a value of 1 to this NULL value.

4.9.2. The cell does not use any other cell and is developed quite independently of other cells. It can be used at a later date as the innermost cell of other appropriate cells. NULL seems to be the appropriate value in this case. The database designer assigns a value of 2 to this NULL value.

4.9.3. The cell is the outermost cell. The value NULL is stored instead of a value of (0,0) due to the dynamic nature of the design activity. An outermost cell today can become an inner cell of some other cell tomorrow. So until it is known for certain that a cell will be the outermost cell it seems natural to set the value of "location" to NULL. The database designer assigns a value of 3 to this NULL value.

A tool which draws the stick diagram of the cells needs the interpretation of these NULL values for drawing the diagrams properly. An IFNULL action associated with the property "location" can determine by itself to which of the three categories a cell belongs and accordingly return the interpretation value. Let us assume that a cell uses the relationship property "Call" to store the names of the cells it contains. If the cell is not a member of a set with the tag "Call" (which means that the cell is presently the outermost cell) then the IFNULL action should return 3. If the relationship property "Call" does not have any value and the cell is not a member of a set with the tag "Call" (which means that the cell does not use any other cells and is developed independently of other cells), then the IFNULL action should return 2. In other cases it should return 1 to indicate that the cell is an inner cell and the value of its property "location" is not known yet.

4.10. The Data Definition and the Data Manipulation Functions

We present below the various functions that are provided for the purposes of insertion, deletion, modification and querying of the contents of the database and for the purposes of changing the structure of the database. Those functions which are concerned with the structure of the database are the means by which dynamic restructuring of the database is achieved.

We give a listing of the names of the functions, a brief functional descriptions of them and the actions they execute, if any. For all the functions the name of the database is passed as one of the parameters. For the sake of brevity this and other obvious parameters are not mentioned below. The full header declarations in the syntax of the programming language C are given in Appendix 2.

4.10.1. CreateDatabase. This function creates a database.

4.10.2. DestroyDatabase. This function destroys a database. It executes the action WHEN_DATABASE_DESTROYED.

4.10.3. NewObject. This function creates a new object class, or a new object under an object class and returns its name. It executes the actions WHEN_OBJECT_ADDED and WHEN_OBJECT_CLASS_CREATED

4.10.4. DeleteObject. This function deletes an object or object class. It executes the actions WHEN_OBJECT_DELETED and WHEN_OBJECT_CLASS_DELETED.

4.10.5. GetObjectClass. This function returns the object class of an object.

4.10.6. SetObjectClass. This function adds an object to an object class. It executes the action WHEN_OBJECT_ADDED.

4.10.7. AddProperty. This function adds a property to an object or object class. It executes the action WHEN_PROPERTY_ADDED.

4.10.8. DeleteProperty. This function deletes a property from an object or object class. It executes the action WHENDELETED.

4.10.9. SetValue. This function stores the value of a property. For the type SET, the new members are included in the set. The old members are discarded. It executes the action WHENMODIFIED.

4.10.10. GetValue. This function retrieves the value of a property. For the type SET, all the members of the set are returned. It executes the action IFNULL if the value is NULL and returns the interpretation provided by the IFNULL action. It returns 0 if the value is not NULL.

4.10.11. GetType. This function returns the type of a property.

4.10.12. GetObjectClass_PROPERTY. This function returns the name of the object class that is specified for the properties of type SET and OBJECT. It takes the object and property name as parameters.

4.10.13. SetObjectClass_PROPERTY. This function stores the name of the object class for the properties of type SET and OBJECT. It takes an object, a property name and an object class as parameters.

4.10.14. VisitObjects. VisitObjects takes the object class and a function as parameters. It visits each object in the object class and calls the specified function with the object name as parameter. If this function returns -1 for an object then VisitObjects quits visiting the rest of the objects in the object class. The value returned by VisitObjects is the name of the last object it visited.

4.10.15. VisitObjectClasses. VisitObjectClasses takes a function as parameter. It visits each object class and calls the specified function with the name of the object class as parameter. If this function returns -1 for any object class then VisitObjectClasses quits visiting the rest of the object classes in the database. The value returned by VisitObjectClasses is the name of the last object class it visited

The following are the operations which are specific to relationship properties.

4.10.16. AddMember. This function adds a member to a set. It executes the action WHENMODIFIED.

4.10.17. DeleteMember This function deletes a member from a set. It executes the action WHENMODIFIED.

4.10.18. GetSETDATA. This function takes the names of the owner, member and tag of a set as parameters. It returns the value of SETDATA, i.e the relationship information between the owner and the member.

4.10.19. SetSETDATA. This function takes the names of the owner, member and tag of a set, and the value of SETDATA as parameters. It stores the value of SETDATA.

4.10.20. VisitMembers. VisitMembers takes the owner and tag of a set, and a function as parameters. It visits each member of the set and executes the specified function with the owner, member and tag as parameters. If this function returns -1 for a member then VisitMembers quits visiting the rest of the members. The value returned by VisitMembers is the name of the last member object it visited.

4.10.21. VisitOwners. VisitOwners takes the member and the tag of a set, and a function as parameters. If the tag is "ALL" then VisitOwners visits all owners of the member and executes the specified function with the owner, member and the tag as parameters. Otherwise, it visits each of the owner object with which the member is in "SIMILAR RELATIONSHIP" (through the tag) and executes the specified function with the owner, member and the tag as parameters. If this function returns -1 for an owner object then VisitOwners quits visiting the rest of the owners. The value returned by VisitOwners is the name of the last owner object it visited.

4.10.22. TraverseComplexObject. The function TraverseComplexObject is used to manipulate a complex object as a whole. This function takes the name of an object (the root of the complex object) and the name of a function as parameters. It visits systematically all the objects of the complex object and calls the specified function with the object name as parameter. If this function returns -1 for an object then TraverseComplexObject does not visit the members of the relationship properties of that object. For each object it then calls a user-written function "Provide_Relationship_Properties" (with the name of the object and a structure as parameters) to get those relationship properties of the object which participate in the traversal. If the user wants all the relationship properties of the object to participate in the traversal then the user can use the function "retrieve_all_rel_properties" which is provided by the system.

4.10.23. retrieve_all_properties. This function retrieves all the properties of an object or object class. The traversal functions described above, namely, VisitObjects, VisitObjectClasses, VisitMembers, VisitOwners and TraverseComplexObject, take a function as one of the parameters. This function can use "retrieve_all_properties" to access all the properties of an object or object class.

4.11. The Data Definition Language

For most of the applications the initial structure of the database will be known in advance. The data definition language aids in building that static structure of the database.

The data definition language we have designed is called "Data and Constraints Specification Language"(DCSL, pronounced "dee-see-yes-el"). It provides facilities for object class definition, object definition, instantiation and initialization, and Constraint Specification. A brief description of DCSL is given below followed by a sample data definition in DCSL.

4.11.1. Description of DCSL

We give below a brief description of the database definition language DCSL. The formal definition of the language is given in Appendix 1.

The Object Class definition and Object definition are distinguished by a header. The header "OBJECT CLASS" denotes an object class. The header "OBJECT" denotes an object. What follows immediately after the header is the name of the object or object class. This name is used to refer to the object or object class elsewhere in the specification. After the name of the object or object class the specifications of its properties are given. The specifications of the properties of the object or object class are given sequentially. That is, the complete specification of a property is followed by the complete specification of the next property and so on.

Each property is given a name and is followed by a separator ":". This name is used to refer to the property elsewhere in the specification. After this separator follows the rest of the specifications for the property. The key word DERIVED specifies that the property is a derived property. The additional specification STATIC means that the action which derives the value of the property need not be executed each time the property is accessed. The rest of the specification of a property consists of type specification, value specification and constraint specification.

The valid type specifications are INTEGER, REAL, TEXT, OBJECT and SET. The type specification for derived properties involves specifying the type of the value returned by the action. The action can return any of the valid types mentioned above. For the types SET and OBJECT there can be an optional specification of an object class. The name of this object class is given after the key word OF. As we mentioned before, the name of this object class is not used by the system as such. It is specified for two reasons : first, it improves the readability of the specification. Without this specification the structure of the database may not be clear. So the name of the object class should be specified if it is known. Second, the WHENMODIFIED action can use this object class name for restricting the value of these properties. For the type SET, there can be an additional (optional) specification of an object class. The name of the object class follows the key words SETDATA OF. This object class name specifies the object class of the object which will store the relationship data. Presently this is provided for documentation purposes only and the compiler ignores this specification. If, as a future extension to the system, the WHENMODIFIED action can be specified for the property SETDATA too, then this specification should be stored for use by such WHENMODIFIED actions.

The value of a property is given between the delimiters "%[" and "%]". For derived properties, the action which derives the value of the property is specified between the delimiters.

The three constraint specifications for properties, namely, IFNULL, WHENMODIFIED and WHENDELETED should be specified in that order but any of them can be omitted. These specifications consist of two parts: the name of th[illegible]ion and the action. The action is specified between the delimiters "%[" and "%]".

For a property, the type definition is mandatory; the initial value as well as the constraint specifications are optional.

A limited support for commenting is provided. Comments should start with "%#" and should be in a separate line.

4.11.2. Sample DCSL Specifications

We give below a sample data definition in DCSL. This definition does not model any real life situations. Some practical examples are presented in Chapter 6. We use this definition to describe the miscellaneous features of DCSL not described above.

```
OBJECT CLASS object_class1
  property8 : TEXT %[ "This is a string" %]
  property1 : SET OF object_class4
            SETDATA OF object_class5
            %[ object1,object2 %]
            WHENDELETED %[ return(canitbedeleted()); %]
  property2 : REAL %[     29.70     %]
            IFNULL %[ checknull(); %]
            WHENMODIFIED %[ checksetmembership(); %]
  property3 : OBJECT OF object_class1 %[object2%]
            WHENMODIFIED %[ checksetmembership(); %]
  property5 : DERIVED INTEGER %[ derivedvalue->int_value = 100; %]
            WHENMODIFIED %[ checksetmembership(); %]
  property6 : DERIVED SET %[derivedvalue.set_value = derivesetvalue();%]
END

%# We specify that object1 belongs to the object class
%#  object_class1

OBJECT object1
  CLASS : object_class1
  property1 : INTEGER %[ 33 %]
  property7 : INTEGER

%# An example of a derived property for which STATIC is specified.
%# Also illustrates that the action returns -1 to indicate that the
%# value can not be derived and hence the value is NULL.

  property3 : DERIVED STATIC REAL %[ if(calc_real(derivedvalue)== 0)
                          return(-1); %]
            WHENMODIFIED %[ checksetmembership(); %]
```

```
END.

OBJECT CLASS objectclass2
 CLASS : object_class1
 WHEN_OBJECT_DELETED : %[ return(checkdeletion()); %]
 WHEN_PROPERTY_ADDED : %[ return(checkaddition()); %]
 property1 : REAL %[ 1234.456 %]
 WHEN_OBJECT_CLASS_DELETED : %[ return(IsItCrucial()); %]
 property10 : SET
 WHEN_OBJECT_ADDED : %[ return(checkobjectaddition()); %]
END

OBJECT object2
 CLASS : objectclass2
 REMOVE : %[ property8, property4 %]
END

OBJECT CONSTRAINTS
 WHEN_DATABASE_DESTROYED : %[ return(OK_to_commit_suicide()); %]
 WHEN_OBJECT_CLASS_CREATED: %[ return(monitorstructure()); %]
END
```

The specification should be fairly self-explanatory. Several things are worth noting.

The special property "CLASS" is used to specify the object class of an object and also to specify an object class using the definition of another object class. "REMOVE" is used to remove properties from an object or object class and this will typically be used to remove properties inherited from the object class.

If the CLASS property is not specified then the object belongs to the object class SYSTEM. This object class can also be explicitly specified for the object.

For initializing properties of type SET and OBJECT, the member objects should be defined elsewhere in the data definition; otherwise it is an error.

It should be noted how the various constraints are specified. The actions "IFNULL","WHENMODIFIED" and "WHENDELETED" are associated with the properties concerned as mentioned before. For other actions, the name of the action is specified as the name of a special property and the property is added to the "nearest available ancestor" as discussed before. The action itself is specified between the delimiters "C[" and "C]" as the value of the special property. For example, the action "WHEN_OBJECT_DELETED" which monitors the deletion of an object is spec[illegible] the value of a property "WHEN_OBJECT_DELETED" and the property is added to that object. The actions "WHEN_OBJECT_DELETED" and "WHEN_PROPERTY_ADDED" actually belong to the objects but they can be specified as properties of an object class too, so that all the objects of this object class will inherit them. The special object CONSTRAINTS has two properties. They are used to store the actions that can be associated with the database.

An action is a piece of code in the implementation language. In our case, an action is a piece of code in the programming language C. The syntax of this code is checked by the C compiler and not by any component of our system. As will be explained in Chapter 5, the actions are formed into functions and these functions are executed whenever the specific condition for the action occurs. These functions are passed appropriate parameters. For example, for the derived property property5 of object_class1, the action which derives the value of the property uses the parameter "derivedvalue" to return the derived value of the property. The details of these parameters are explained in Chapter 5. It should also be noted that in property3 of object1 the action which derives the value of the property uses the "return" statement to return -1 to indicate that the value can not be derived and hence the value is NULL.

4.12. Summary

In this chapter we have presented the design of an object oriented database management system. The database is viewed as a model (or representation) of some real world phenomenon. The basic concepts regarding the modeling primitives are discussed. The details of how relationships are established and how constraints are specified are presented. The mechanism by which relationships are represented and manipulated enables the handling of complex objects easy. The constraint specification scheme is designed to be flexible and powerful as demanded by the design applications. The various primitives for processing of the database and for maintaining the structure of the database are listed. The design of a Data Definition Language and its description are presented. The next chapter discusses the implementation details of our database management system.

Chapter 5

Implementation

5.1. Introduction

In this chapter, we discuss the implementation details of our prototype database management system. The current implementation runs under the UNIX (4.2 BSD) operating system on a VAX 11/780 and is written in the programming language C[36]. The compiler for the Data Definition Language DCSL is written using Yacc[32] and Lex[41]. The system is implemented using the data management routines of FDB[23,24]. A short description of FDB is provided followed by the implementation details of our system.

5.2. FDB

FDB is a frame based database management system. It is based on the model of "frames" and "slots". The database consists of frames and a frame is a collection of slots. There are two kinds of frames, namely, "ordinary" frames and "meta" frames. The "meta" frames are used to describe the structure of other frames and "ordinary" frames are used to store data. Each slot has a name, a value and a type field. The type of the value of a slot can be INTEGER, REAL, STRING or FRAME. A value of type FRAME is a pointer to a frame and is the means by which complex structures are built in FDB. There are various data management routines available for the following primitive tasks : creation and destruction of frames, addition of slots to a frame and deletion of slots from a frame, storing and retrieving the value of slots.

FDB, though it provides only primitive facilities, has been successfully used for building applications[11,23]. It is also an invaluable tool for prototyping other higher level database management systems. Its flexible facilities are very useful in testing out new ideas in a short time. It abstracts out the low level details of storage management and provides a simple and adequate set of data management routines (based on the model of frames and slots). We have successfully utilized this aspect of FDB for prototyping our system. The rest of the chapter describes the implementation details of the various components of our sy[illegible]n.

5.3. Objects and Object Classes

Each object is an FDB "ordinary" frame. The name of the object is the name of the "ordinary" frame. Each object class is an FDB "meta" frame. The name of the object class is the name of the "meta" frame. Since the frames are identified with unique integers the names of objects and object classes are also integers. The type of these names is actually OBJECT which is a "typedef" of int. Frames are also used to store some system information and to build the structures required for storing the information associated with the properties.

5.4. Properties

Properties are implemented using the "slots" of FDB. The name of a property of an object (or object class) is the name of a slot and this slot is added to the frame which implements that object (or object class). Consequently, the type of the name of a property is a string (i.e char *). Let us call this slot the "start slot" of a property. The concept of a property is at a higher level than the concept of a slot. So the information associated with each property is stored in a structure and this structure can be accessed through the "start slot". The details of this structure for a primary property and a derived property are slightly different. Also, the structures for properties of type SET and OBJECT store more information than the structures for properties of other types. The structures are explained below.

5.4.1. Primary Properties

The structure associated with a primary property is shown in fig 5.1.

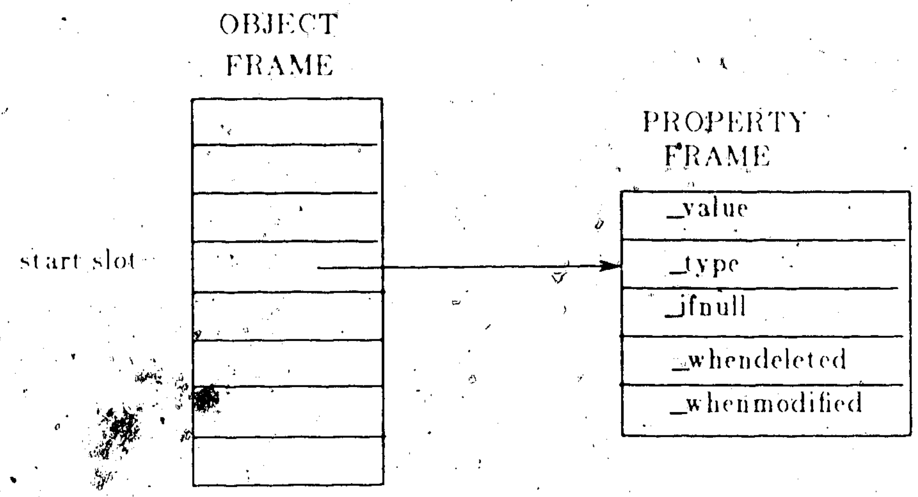

Figure 5.1 Structure for a Primary Property

The OBJECT FRAME implements the object and the PROPERTY FRAME stores the information associated with the property. The value of the start slot is the PROPERTY FRAME. The PROPERTY FRAME has four slots. The slot "_value" stores the value of the property and the slot "_type" stores the type of the property. The other three slots are used to implement the constraint specification mechanisms and are explained subsequently.

5.4.2. Derived Properties

The structure of a derived of property is shown in figure 5.2.

The OBJECT FRAME implements the object and the PROPERTY FRAME stores the information related to the property. The value of the start slot is this PROPERTY FRAME. The values of the various slots of PROPERTY FRAME are explained below. The piece of code which derives the value of a derived property is converted into a function (the details of this follow later in this chapter) and the name of this function is stored as the value of the slot "_procname". The derived value

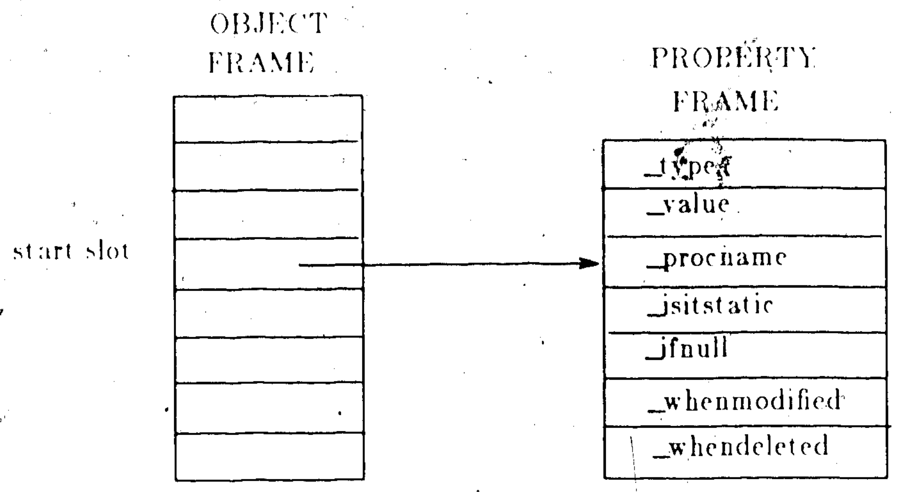

Figure 5.2 Structure for a Derived Property

is stored in the slot "_value" and the type of this value is stored in the slot "_type". If the slot "_Jsitstatic" has a value of 1 then on subsequent accesses of the property value, the value stored in the slot "_value" is returned without executing the function. The slots "_jfnull", "_whenmodified", "_whendeleted" are used to implement the constraint specification mechanisms and are explained subsequently.

5.4.3. Structure of Type OBJECT

For the type OBJECT, the PROPERTY FRAME has an additional slot "_objectclass" which stores the name of an object class. As mentioned before, the value of this can be used by the WHENMODIFIED action to monitor the value of a property of type OBJECT.

5.4.4. Structure of Type SET

For the types INTEGER, REAL, TEXT and OBJECT the "_value" slot of the PROPERTY FRAME directly stores the value of the property (primary or derived). But for the type SET the "_value" slot stores another structure as its value.

The structures associated with the type SET are shown in figures 5.3 and 5.4. Figure 5.3 depicts the structure associated with the owner object:

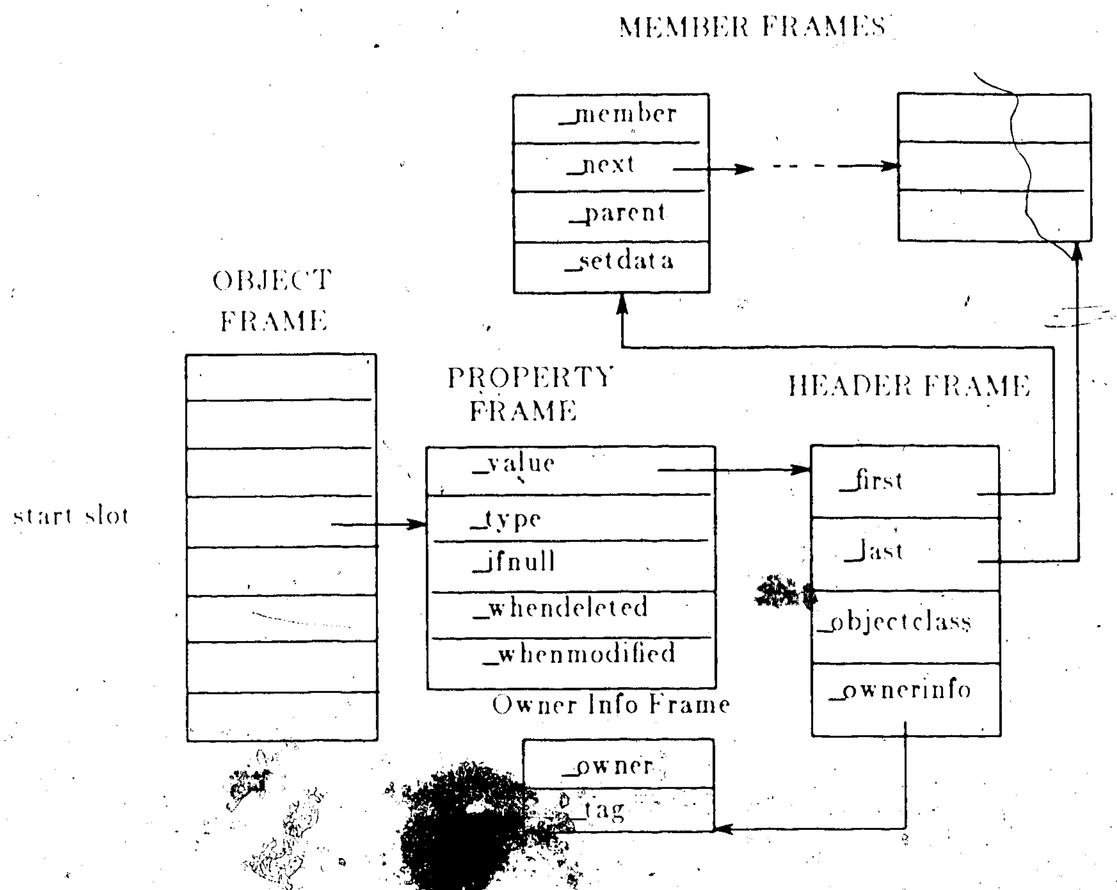

Figure 5.3 Structure for the Type SET Associated with the Owner

For each member of the set there is a MEMBER FRAME. A linked list of the MEMBER FRAMEs of the set is maintained. The HEADER FRAME stores the names of the first and last MEMBER FRAMEs in the linked list in the slots "_first" and "_last" respectively. The slot "_objectclass" stores the name of the object class that is specified for the set. As mentioned before this value can be used by the WHENMODIFIED action to restrict the membership of the set. The HEADER FRAME stores the owner information, namely, the name of the OBJECT FRAME and the name of the tag (i.e the name of the property) in the slot "_ownerinfo". The value of the "_value" slot of PROPERTY FRAME is the HEADER FRAME. Each MEMBER FRAME has four slots. The slot "_member" stores the name of the member object, the slot "_parent" stores the name of the HEADER FRAME, the slot "_next" stores the name of the next MEMBER FRAME and the slot "_setdata" stores the relationship data (i.e

"SETDATA") between the owner and the member. Figure 5.4 depicts the structure associated with the member object.

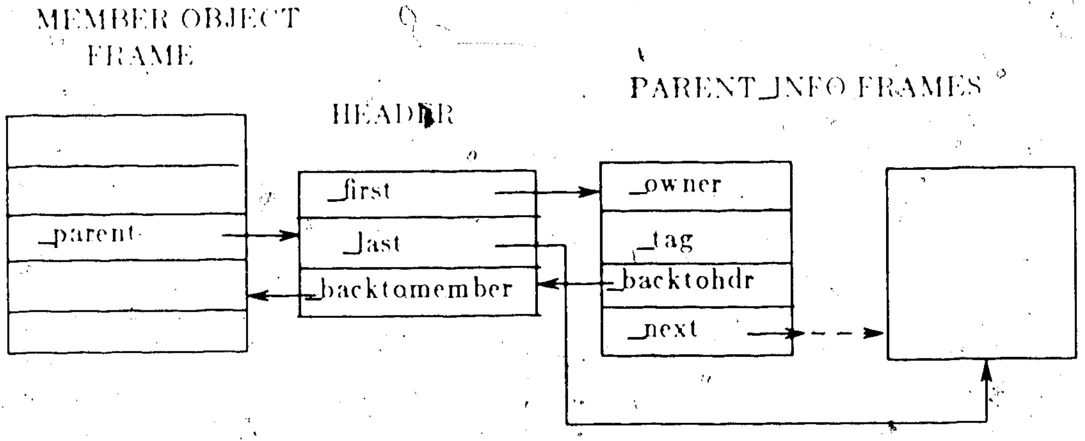

Figure 5.4 Structure for the Type SET Associated with the Member

The MEMBER OBJECT frame implements the member [illegible] The owner and tag information for each set in which the object is a [illegible] stored in a PARENT_INFO frame. A linked list of PARENT_INFO F[illegible]tained. The HEADER frame stores the name of the first PARENT_INFO frame, and the name of the last PARENT_INFO frame in the slots "_first" and "_last" respectively. The slot "_backtomember" of the HEADER frame stores the name of the MEMBER OBJECT frame. The "_parent" slot of the MEMBER OBJECT frame stores the name of the HEADER FRAME. Each PARENT_INFO frame has four slots. The slot "_owner" stores the name of the owner of the set, the slot "_tag" stores the name of the tag of the set, the slot "_next" stores the name of the next PARENT_INFO frame and the slot "_backtohdr" stores the name of the HEADER frame.

5.5. Communication between the Application and the DBMS

This section explains how the values of various types of data are communicated between the application programs and the DBMS. The functions which are concerned with this task can be classified into two groups. The first group contains the functions GetValue, SetValue, the function which derives the value of a derived property, and the IFNULL function (for passing the default value). These functions deal with data of several types. The second group contains functions like GetSETDATA, SetSETDATA, GetObjectClass, SetObjectClass, GetType and the IFNULL function (for passing the interpretation). These functions deal with the type INTEGER only. We decided to adopt different strategies for the two classes of functions. On a choice between orthogonality and ease of use we chose the latter.

The functions of the second category pass values as follows: Functions which pass INTEGER data to the DBMS carry the value in a parameter of type INTEGER (for example, SetSETDATA, SetObjectClass etc.). Functions which pass INTEGER data from the DBMS do so by using the "return" statement (for example, GetSETDATA, GetObjectClass, GetType etc.).

The functions of the first category pass values by passing a pointer to a union variable of type "value_header" whose declaration is given in figure 5.5.

```
union value_header {
  int int_value;
  float real_value;
  char *text_value;
  OBJECT object_value;
  struct set_header set_value;
}
```

Figure 5.5 Declaration of "value_header"

The system will use the appropriate field of the union structure to retrieve (for Getvalue) or store (for SetValue) the value since it knows the type of he property. It is also the responsibility of the application to use the correct field for setting or retrieving the values. Since C does not allow passing unions to functions, a pointer to a union has to be passed to SetValue too, even though it does not modify the value of the union variable. The various fields of the union (except "set_value") should be self-explanatory.

The field "set_value" of the union "value_header" is used for setting or getting the values of type SET and is a structure variable of type "set_header" whose declaration is given in figure 5.6.

```
struct set_header {
    OBJECT owner;
    char *tag;
    struct set_info *first_member;
}

struct set_info {
    OBJECT member;
    OBJECT set_data;
    struct set_info *next;
}
```

Figure 5.6 Declaration of "set_header" and "set_info"

It is essentially a linked list. The first element is the header of the set. The data item "owner" contains the name of the owner object of the set and the data item "tag" contains the "tag" of the set. The rest of the elements in the list are of type "set_info" whose declaration also is given in figure 5.6. The data item "member" contains the name of the member object and the data item "set_data" contains the name of the object which stores data about the relationship. The set header information which may not be used in some circumstances actually facilitates the development of general purpose set processing routines.

As an example, if the application wants to store an integer value it should pass "SetValue" a pointer to the integer value. If the application wants to retrieve a

TEXT value then it should pass "GetValue" a pointer to a string (which is again a pointer to a character). Similarly for a property of type SET, "GetValue" will construct the linked list explained above and pass a pointer to the first element (header) of the list. Likewise, for setting the value of a property of type SET (through SetValue), the application should construct the linked list and pass a pointer to the first element of the list.

5.6. Implementation of Actions

The implementation of the various actions, namely, the action which derives the value of a derived property and the actions which specify the constraints, are explained in this section.

Each action is a piece of code in the programming language C. The code is converted to a function with parameters by adding the necessary C syntax. Each function is given a unique name. The actions, though they are not defined as functions by the user, can reference the formal parameters and can use the "return" statement. The functions which derive the value of derived properties are called "deriving functions" and the functions which specify the constraints are called "constraint functions".

At the beginning of the run of the program, the addresses of all the deriving functions and all the constraint functions are obtained from the symbol table and are stored in a main memory table. The main memory table establishes the mapping between the name of a function and its address. Whenever a function is to be executed, its address is obtained from the main memory table and, is executed with the appropriate actual parameters. The details of the formal parameters of these functions and where the names of these functions are stored, are explained below.

5.6.1. The Deriving Functions

The name of a deriving function is stored as the value of the slot "_procname" of the PROPERTY FRAME (figure 5.2). This function is called by the function "GetValue". The formal parameters of a deriving function are "object", the name of the object containing the derived property, "property", the name of the derived property and "derivedvalue", a union variable of type "value_header" for returning the derived value. This function should return -1 (using the "return" statement) if the value can not be derived, to indicate to GetValue that the value is NULL.

5.6.2. The Constraint Functions

The details of where the name of the constraint functions are stored, and the number and names of their formal parameters depend on the database operation with which the constraint is associated. They are explained below.

5.6.2.1. IFNULL, WHENMODIFIED and WHENDELETED for Properties

The slots "_ifnull", "_whenmodified" and "_whendeleted" of the PROPERTY FRAME shown in figures 5.1 and 5.2 are used to store the names of these constraint functions.

The "_ifnull" slot stores the name of the IFNULL constraint function and the formal parameters of this function are "object", the name of the object containing the property, "property", the name of the property, "defaultvalue", a pointer to a "value_header" union variable for returning the default value and "defaultset", a pointer to an integer variable for indicating if a default value has been provided. The "_whenmodified" slot stores the name of the WHENMODIFIED constraint function and the formal parameters of this function are "object", the name of the object containing the property, "property", the name of the property and "value", the new value for the property. It is expected that the old value will be accessed by this constraint function using the GetValue function provided by the system. The "_whendeleted" slot stores the name of the WHENDELETED constraint function and the formal parameters of this function are "object", the name of the object and "property", the name of the property. It is expected that the value of the property will be accessed by this constraint function using the GetValue function provided by the system.

The default value of a property and the interpretation of the NULL value of a property is conveyed by the IFNULL constraint function as follows. The IFNULL constraint function is called by GetValue when it finds a NULL value. As mentioned before, for derived properties, this condition occurs if the function deriving the value returns -1. If the IFNULL function wants to provide a default value then it does so through the parameter "defaultvalue". The variable pointed to by the parameter "defaultset" is set to YES or NO to indicate whether a default value has been provided. It should use the "return" statement to return the interpretation value of NULL. As we mentioned in Chapter 4, it returns -1 if it does not want to interpret the NULL value. GetValue, in turn, sets the global variable "DEFAULTSET" to the value of the variable pointed to by the parameter "defaultset", and returns the interpretation of the NULL value given by the IFNULL function. As we mentioned in Chapter 4, GetValue returns 0 if the value is not NULL.

The application program can effectively use the general strategy given in figure 5.7 if it wants to process the NULL value by itself:

```
interpret = GetValue(object,property,&propertyvalue);
if (interpret == 0 || (interpret != 0 && DEFAULTSET == YES))
  /* Value is available; proceed */ ;
else
  /* Value is NULL. Use the interpretation if necessary. The default
   value is not provided. So the value in "propertyvalue" is junk. */ ;
```

Figure 5.7 General Strategy for Processing the NULL Values

The parameter "propertyvalue" of GetValue is used to get the value of the property

The code following the GetValue call is not necessary if the IFNULL action always provides a default value.

5.6.2.2. WHEN_OBJECT_DELETED and WHEN_PROPERTY_ADDED

Two slots, namely, "_whenobjectdeleted" and "_whenpropertyadded" are added to the frame which implements an object. The slot "_whenobjectdeleted" stores the name of the constraint function WHEN_OBJECT_DELETED. The formal parameter of this constraint function is "object" which is the name of the object. The slot "_whenpropertyadded" stores the name of the constraint function WHEN_PROPERTY_ADDED. The formal parameters of this constraint function are "object", the name of the object or object class to which the property is added, "property", the name of the property and "type", the type of the property.

5.6.2.3. WHEN_OBJECT_CLASS_DELETED and WHEN_OBJECT_ADDED

Two slots, namely "_whenobjectclassdeleted" and "_whenobjectadded" are added to the frame which implements an object class. The slot "_whenobjectclassdeleted" stores th[illegible]me of the constraint function WHEN_OBJECT_CLASS_DELETED. The formal parameter of this constraint function is "objectclass" which is the name of the object class. The slot "_whenobjectadded" stores the name of the constraint function WHEN_OBJECT_ADDED. The formal parameters of this constraint function are "object", the name of the object and "objectclass", the name of the object class to which the object is added.

5.6.2.4. WHEN_DATABASE_DESTROYED and WHEN_OBJECT_CLASS_CREATED

Two slots, namely, "_whendatabasedestroyed" and "_whenobjectclasscreated" are added to a frame maintained by the system. The name of the constraint function WHEN_DATABASE_DESTROYED is stored as the value of the slot "_whendatabasedestroyed" and the name of the constraint function "WHEN_OBJECT_CLASS_CREATED" is stored as the value of the slot "_whenobjectclasscreated". These two functions do not have any parameters (other than the database name).

5.7. The Operators

The logic of the traversal operators (functions) provided by the system are explained below. Detailed header declarations of these functions as well as the other functions not described below are given in Appendix 2.

5.7.1. VisitObjectClasses

This function takes a pointer to a function as parameter. A list of all object classes in the database is maintained by the system. The list of object classes is sequentially traversed. For each object class, the specified function is executed with the object class name as the parameter.

5.7.2. VisitObjects

This function takes a pointer to a function and an object class as parameters. A list of all objects in an object class is maintained by the system. The list of objects in the object class is sequentially traversed. For each object the specified function is executed with the object as the parameter.

5.7.3. VisitMembers

This takes the name of the owner, the tag and a pointer to a function as parameters. It follows the list of MEMBER frames associated with the owner of the set and for each member executes the specified function with the owner object, member object and tag as parameters.

5.7.4. VisitOwners

This function takes a member and the tag of a set, and a pointer to a function as parameters. It follows the list of OWNER_INFO frames associated with the member of a set. For each owner with the specified tag, the specified function is executed with the owner, member and tag as parameters.

5.7.5. TraverseComplexObject

This takes an object and a pointer to a function as parameters. Let us call this function "Process_Object". It first executes "Process_Object" with the object as parameter. Then it executes the user-written function "Provide_Relationship_Properties". It passes the object and a pointer to a structure variable (of type name_list_header) as parameters. The structure declaration is given in figure 5.8. The structure variable is used by "Provide_Relationship_Properties" to provide the names of relationship properties of the object that will participate in the traversal.

```
struct name_list_header {
  char *name;
  struct name_list_header *ptr_to_next_header;
}
```

Figure 5.8 Declaration of "name_list_header"

It can be seen that the value returned by "Provide_Relationship_Properties" is a linked list of character strings. The last element should have the value of 0 for ptr_to_next_header. If all the relationship properties of an object are to participate in the traversal, the system function "retrieve_all_rel_properties" can be used. This function takes an object and a pointer to the above mentioned structure, and fills in the structure with all the relationship properties of the object.

Then TraverseComplexObject visits the members of the relationship properties returned by "Provide_Relationship_Properties". For each relationship property provided, and for each member of the set defined by the relationship property, it recursively calls itself.

The function which is passed as a parameter to all the above traversal operators can use the system provided function "retrieve_all_properties" to access all the properties of an object or object class. "retrieve_all_properties" takes the name of an object or object class and a pointer to the structure name_list_header (figure 5.8) as parameters, and fills in the structure with all the properties of the object or object class.

5.8. The Data Definition Language DCSL

The compiler for the Data Definition Language DCSL is written using the compiler writing tools Yacc and Lex. Lex is used to write the lexical analyzer and Yacc is used to write the Parser. The compiler parses the DCSL specifications and generates the FDB structures as explained previously in this chapter. During the parsing phase, the compiler builds tables (main memory structures) which are used to generate the FDB structures during the generation phase. Two tables are important, they are the "Object Table" and the "Property Table". The structure of the tables are shown in tables 5.1 and 5.2.

Object Name	Status	Object Number

Table 5.1 The Structure of Object Table

Name	Type	Value	Index	derived	static	Ifnull	Whenmod	Whendel

Table 5.2 The Structure of Property Table

In object table, each record stores information about an object or object class. The field "Object Name" stores the name of the object or object class, the field "Status" tells whether the table entry refers to an object or object class and the field "Object Number" stores the OB name of the frame which implements the object or object class.

In the property table, each record stores information about a property. The field "Name" stores the name of the property and the field
"Type" stores the type of the property. For properties of type SET and OBJECT, the field "Type" also stores the name of the object class that is specified for these properties. The field "Value" stores its initial value. "Index" stores a pointer to the corresponding object (i.e object of the property) entry in the object table. "derived" stores the information whether the property is a derived property or not. "static" stores the information whether a derived property is static or not. "Ifnull" stores the piece of code for the action "IFNULL", "Whenmod" stores the piece of code for the action "WHENMODIFIED" and "Whendel" stores the piece of code for the action "WHENDELETED".

5.9. Summary

In this chapter, we have described the implementation details of our prototype database management system. A description of the implementation of the various components of the system including how the communication between the DBMS and the application programs take place and how the various actions are implemented, is provided. The next chapter provides some practical examples to illustrate how our system can be used.

Chapter 6

Examples

In this chapter, we present examples of how to use our system. The various facilities are illustrated with some real life examples. Though care has been taken to present these examples as correctly as possible, we do not make any claim to the correctness of their design. The main aim in presenting these examples here is to illustrate ~~how~~ real life situations can be modeled using our system.

6.1. A Chip Design Database

The following design of a VLSI chip database is taken from the Vdd system[13] implemented at Bell Laboratories. This system uses a database management system based on the Relational model with some extensions. We use this example to illustrate many of the interesting features of our system.

Fig. 6.1 shows the hierarchical model of a chip.

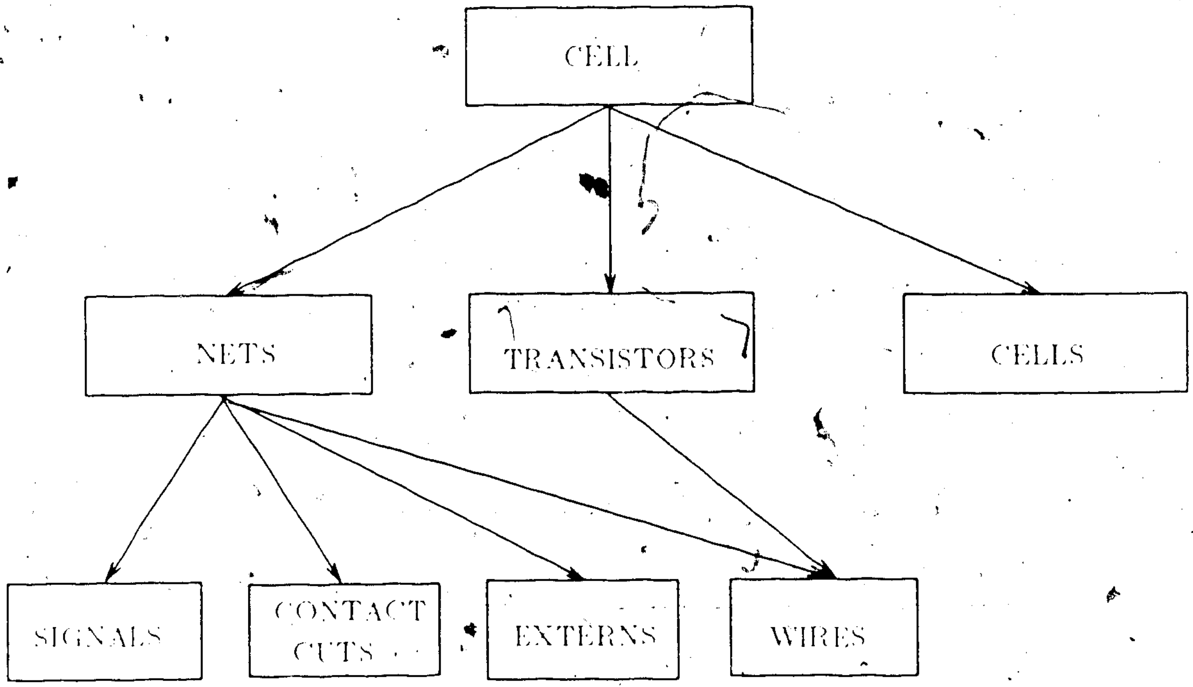

Figure 6.1 Hierarchical Model of a VLSI Chip

For a detailed explanation on the various terms (which are quite specific to VLSI domain) please see[13]. We give below a short description of the various components of a chip.

The chip contains information cells. The components of a cell can be wires, signals, externs (input and output ports of cells), contact cuts, transistors and cells (which is called a "call"). The "call" is the basic mechanism by which the chip hierarchy is built up. Another important concept is the notion of *nets*. A net is a maximal set of wires which are electrically equivalent to each other. A signal name can be assigned to externs of cells at different levels of hierarchy to indicate that the signal is to propagate across these cells. An extern is associated with a net and hence a signal also is associated with a net. Contact cuts and wires are similarly part of some net.

Each chip is organized as one database. The design is specified below in DCSL.

```
OBJECT CLASS cell
 creationdate : OBJECT
 cellname     : TEXT
 versionno    : INTEGER %[ 0 %]

 estimatedmanhours : DERIVED REAL %[
   float estimate(); int version;
    GetValue(object,"versionno",&version);
    if (version == 0 || version == 1) return (-1); else
    derivedvalue->real_value = estimate(object,property); %]

              IFNULL %[
   int version;
    GetValue(object,"versionno",&version);
    if (version == 0) return(1);
    *defaultset = YES;
    defaultvalue->real_value = 500.0;
    return(-1);         %]

  estimatedcost    : DERIVED REAL %[
    union value_header manhours;
    int interpret;
    interpret = GetValue(object,"estimatedmanhours",&manhours);
     if (interpret == 0)
   /* Value is there, no problems */
      derivedvalue->real_value = 1.5*manhours.real_value*65.83;
     else if (DEFAULTSET==YES) {
   /* Value is NULL; but default is provided; proceed as usual
```

```
    but give a warning. */
      derivedvalue->real_value = 1.5*manhours.real_value*65.83.
      printf("cost is based on default manhours. Beware!"); }
     else { if (interpret == 1)
   /* Value is NULL; the interpretation is "it is version 0" */
      printf("For version no. 0 cost can not be estimated0);
   /* Whatever the interpretation is, return a value of -1
    to indicate to GetValue that cost can not be derived. */
      return (-1); } %]

  extern_nets : DERIVED SET OF net
  local_nets  : DERIVED SET OF net
  BoundingBox : OBJECT OF boxspecs
  Transistors : SET OF transistor SETDATA OF transistor_cell
  Signals     : SET OF signal
  Cuts        : SET OF cut SETDATA OF cut_cell
  Externs     : SET OF extern SETDATA OF extern_cell
  Wires       : SET OF wire SETDATA OF wire_cell
  Call        : SET OF cell SETDATA OF call_cell
  Crossinfo   : DERIVED SET OF object_pairs
END

OBJECT CLASS transistor
  name : TEXT
  transistortype : TEXT WHENMODIFIED %[
        return(checktransistortype(value->text_value)); %]
   polysilicon_size : INTEGER
   diffusion_size   : INTEGER
   diffusion_north  : SET OF wire
   diffusion_south  : SET OF wire
   gate             : SET OF wire
   substrate        : SET OF wire
   BoundingBox      : OBJECT OF boxspecs
END

%# An object of the object class transistor_cell
%# stores the relationship data between a
%# transistor and a cell.

OBJECT CLASS transistor_cell
   place          : OBJECT OF coordinates
   reflection     : REAL
   rotation       : REAL
END

OBJECT CLASS net
  name    : TEXT
  wires   : SET OF wire
  cuts    : SET OF cut
  signals : SET OF signal
  externs : SET OF extern
END
```

```
OBJECT CLASS signal
  name     : TEXT
END

OBJECT CLASS cut
  [illegible]     : TEXT
  cuttype  : INTEGER
  cutsize  : OBJECT
END

%# An object of object class cut_cell stores the
%# relationship data between a cut and a cell.

OBJECT CLASS cut_cell
  location : OBJECT OF coordinates
          WHENMODIFIED %[
          return(checklocation(object,property,value->int_value)); %]
          WHENDELETED %[ return(NO); %]

%#    This property can not be deleted.

END

OBJECT CLASS extern
  name     : TEXT
  layer    : INTEGER
END

%# An object of object class extern_cell stores the
%# relationship data between an extern and a cell

OBJECT CLASS extern_cell
  location : OBJECT OF coordinates
          WHENMODIFIED %[
          return(checklocation(object,property,value->int_value)); %]
END

OBJECT CLASS wire
  layer    : INTEGER
  size     : OBJECT
END

%# An object of object class call_cell stores the
%# relationship data between a call and a cell.

OBJECT CLASS call_cell
 location : OBJECT OF coordinates
           IFNULL %[
                 *defaultset = = TRUE;
                 defaultvalue->object_value = setdefault();
         /* The function setdefault() can return the name
           of an object of object class coordinate with
           the coordinates set to 0.0, 0.0  */ %]
  reflection : REAL
  rotation   : REAL
 END

 OBJECT CLASS objectpairs
  object1 : OBJECT
  object2 : OBJECT
 END

 OBJECT CLASS coordinates
  Xcoordinate : REAL
  Ycoordinate : REAL
 END

 OBJECT CLASS boxspecs
  length : REAL
  width  : REAL
 END
```

The DCSL specification should be fairly self-explanatory. Some of the important points are detailed below.

6.1.1. This example illustrates how a complex object can be structured using relationship properties. A property of type SET is the basic means by which the connection between one level and its lower level in the hierarchy is established. In object class "cell" there is a property "Call" which is of type SET and is intended to take cells as members. This establishes the hierarchy. Similarly, the property "Transistors" is intended to have any set of transistor objects as its value. The function TraverseComplexObject is provided for traversing a complex object. The functions VisitChildren and VisitMembers can also be used for this purpose. Example of these functions follow later in this chapter.

6.1.2. The object classes whose names end with "_cell" are templates for the objects that will store the relationship data (in the special property called SETDATA). For example, the object of object class "transistor_cell" will store the relationship data between the objects "transistor" and "cell", and the property SETDATA of this relationship will store this object as its value. These objects need not store the name of the partners of the relationship. This information is maintained by the system.

8.1.3. The derived properties "estimatedmanhours" and "estimatedcost" of "cell" illustrate how the derived values, default value of a NULL value and the interpretation of the NULL value are communicated to the application programs. The action which derives the value of "estimatedmanhours" first checks the value of the property "Version". If the value is 0 or 1 then it returns a value of -1 to indicate that the value is NULL. Otherwise it calls estimate() (we are not concerned with how estimate() estimates the value) to estimate the value. It sets the parameter "derivedvalue" (which is a "value_header" union variable) to the value returned by estimate (using the field real_value since the type of the property is REAL). It should be noted that the action also uses the parameters "object" and "property" whose values in this case are the name of the cell object and the name of the property, namely, "estimatedmanhours". GetValue tests the value returned by this action: if it is -1 then it calls the IFNULL action. The IFNULL action, again, checks the version of the cell: if the value is not equal to 0 then it sets the variable pointed to by the parameter "defaultset" to YES, sets the variable pointed to by the parameter "defaultvalue" to 500.0 and returns -1 to indicate that the NULL is not interpreted; otherwise it returns 1 to indicate that the interpretation of this NULL value is 1.

The action which derives the value of "estimatedcost" first calls GetValue to access the value of "estimatedmanhours". If GetValue returns 0 then the value is not NULL and hence it calculates the cost according to the given formula and accordingly sets the variable pointed to by the parameter "derivedvalue"; otherwise if the global variable DEFAULTSET is set to YES (by GetValue) then in addition to calculating the cost it prints a warning message that the cost is based on a default value. Otherwise it decodes the interpretation of the NULL value, if the value returned by GetValue is 1 then the meaning is "NULL since Version No. of the cell is 0" and it accordingly prints a message, and returns a value of -1 to GetValue to indicate that the value can not be derived (hence NULL).

6.1.4. The property "Crossinfo" of the object class "cell" is of type DERIVED SET. It is also specified that the set will contain objects of object class "objectpairs". Each object of "objectpairs" contains the names of two objects which overlap. A cell can contain many such pairs and hence the type SET. This is a derived property because the overlap information depends on the dimension and location of the objects in the cell.

6.1.5. The properties "local_nets" and "extern_nets" of "cell" are derived properties because their values can be calculated from the values of the properties "Signals", "Cuts", "Externs" and "Wires" of "cell".

6.1.6. The property "size" in object class "wire" is of type OBJECT. We do not specify from which object class the value should come from since it is not known at the present moment.

6.1.7. Some constraint specifications are illustrated. There are many instances of WHENMODIFIED specification for properties. For example, the WHENMODIFIED specification of property "transistortype" of object class "transistor" ensures that only valid names (strings) are stored as the transistor type. As specified, this is achieved through the user-written function "checktransistortype()". The new value of the property "transistortype" (i.e the parameter "value" to the WHENMODIFIED function) is passed as parameter to this function. A sample implementation of this function is given in Appendix 3. A more complicated function is needed to monitor the value of the property "location" of objects of the object classes "cut_cell" and "extern_cell". As specified, the function "checklocation()" does the job. It takes the name of the object, the name of the property and the new value as parameters. A sample implementation of this function is given in Appendix 3. For the property "location" of object class "cut_cell" we have used the WHENDELETED specification to ensure that it will never get deleted. The reason is due to the semantics of the object class "cut_cell". If we allow the property "location" to be deleted, we will lose the important information that there is a property "location" between a cell and a cut. For sake of brevity, we omitted this constraint specification for other properties for which this kind of constraint should have been specified.

6.2. Dynamic Restructuring of the Database

We illustrate below a practical situation in which the facility to restructure the database will be helpful.

A design rule checker checks each cell to determine whether the design of the cell meets a set of design rules. With our organization of the chip, the design rule checker will check the cells recursively. If cell A calls cell B then B must be checked in addition to A. In a situation when only A is modified and not B then such a checker will do unnecessary work by checking B a second time. (Of course, it is necessary to check the interaction between the call of B and other components of A.) A simple solution to prevent this is to add a property "checked" to all cells. This will act as a flag. It can be set to YES to indicate that the cell has been checked since last modified and NO to indicate that it needs to be checked. The checker can safely skip a cell if the cell and all the cells that it calls are marked YES. In our example, the design rule check of A will skip B. This "checked" information could have been stored by the design rule checker outside the database. This is highly undesirable; it involves more work to do that and it is unnatural since that information is actually part of the design information until the design is fully finished (i.e until further update is prevented).

There are several reasons why the property "checked" would not be included in all the cells during the database design stage. Firstly, this problem of unnecessary design rule checking may not have occurred to the database designer. Secondly, even if the property is included at the database design stage, there is not much meaning in keeping that information with the design data after the design has been fully checked and the design has been frozen. With our facility, the properties can be added (using the function AddProperty) when the design rule check is first initiated. When the design of the chip is finally over and no further update is allowed, these properties can be removed (using the function DeleteProperty).

We have taken this real life example of design rule checking to illustrate the utility of dynamic restructuring of the database. There are many other circumstances w re this facility can be effectively used. Some design information which will be with the database always, might not have been thought of at the initial design of the database. For example, a situation like the following will warrant this facility, say, few lines of documentation has to be added to transistor objects which are to be used only in specific places, this decision is taken after finding out that some serious design errors are consistently caused by improper use of these transistors. In our system, a property, say, "GuidelineForUse", of type TEXT can be added to those specific transistor objects and the documentation can be stored as the value of this property. It should be noted here that the flexibility we have allowed for the members of an object class is useful for dealing with such situations. Assuming that all transistor objects are under the object class transistor, a restriction of homogeneous structure of member objects of an object class will force the inclusion of this property to all transistor objects which is probably unnatural and unnecessary.

6.3. Examples of the operators

In this section we present some simple examples to illustrate how the operators (C functions) can be effectively used. The database name is not passed as parameter to the various operators in order to reduce the length of code.

6.3.1. VisitObjects, VisitOwners, GetSETDATA and GetValue

Suppose it is of interest to us to execute the following query: "Find all transistors whose bounding box is within 'X' units of the bounding box of the immediately surrounding cell". The following code will accomplish this. It is assumed that all transistor objects are under the object class transistor.

```
-----------------------------
Rule_check_transistors():
{
 int (*check_surrounding_cell) ():
 VisitObjects(transistor,check_surrounding_cell);
 }
----------------------------
check_surrounding_cell(object)
 OBJECT object:
 {int (*print_if_condition_satisfied) ():
  VisitOwners(object,"transistors",print_if_condition_satisfied):
 }
-----------------------------
print_if_condition_satisfied(owner,member,tag)
OBJECT owner,member:
char *tag:
{OBJECT relationshipdata,cell_bounding_box,transistor_bounding_box;
 int Is_it_satisfied;
 relationshipdata = GetSETDATA(owner,member,tag):
 GetValue(owner,"BoundingBox",&cell_bounding_box):
 GetValue(member,"boundingbox",&transistor_bounding_box):
 Is_it_satisfied = checkcondition(cell_bounding_box,
                         transistor_boundingbox,relationshipdata);
 if(Is_it_satisfied == TRUE) printobject(member):
 }
-------------------------------
```

The functions check_surrounding_cell, print_if_condition_satisfied, checkcondition and printobject are to be written by the user.

The functions VisitObjects and VisitOwners do most of the job. VisitObjects visits each transistor object and calls the function check_surrounding_cell with the name of the object as parameter. check_surrounding_cell calls VisitOwners. Visitowners visits the owners of the transistor object with tag "transistors" and calls the function print_if_condition_satisfied with the cell object, the transistor object and the tag "transistors" (the owner, the member and the tag in the general case) as parameters. In this case there will be only one owner(i.e cell). print_if_condition_satisfied prints the contents of the transistor object if the condition specified before is satisfied.

To check the condition, the relationship information between the cell and the transistor (which is the location information) has to be retrieved. This is accomplished by the GetSETDATA function. Using the function GetValue the bounding box values of the cell and the transistor are retrieved. The user-written function "checkcondition" checks the specified condition. If the condition is satisfied, the user-written function "printobject" prints the contents of the transistor object.

The query we just discussed is fairly general and can be used as a model for a whole class of similar queries.

6.3.2. VisitObjectClasses

Suppose, all objects of the database have to be visited and a function, say, Verify_Time_Stamp has to be executed for each one of them. The following code will accomplish it.

```
Kill_Oldies()
{
 int Operate_on_object();
 VisitObjectClasses(Operate_on_object);
}
-----------------------------------------------
Operate_on_object(object_class)
OBJECT object_class;
{
 int  Verify_Time_Stamp();
VisitObjects(objectclass,Verify_Time_Stamp);
}
```

VisitObjectClasses is used to visit all the object classes in the database. It visits each object class in the database and calls the function "Operate_on_object" with the name of the object class as parameter. The function "Operate_on_object" calls VisitObjects. VisitObjects visits each object in the object class and calls the function "Verify_Time_Stamp" with the name of the object as parameter.

This query can be trivially generalized to execute any function on all objects of the database.

6.3.3. TraverseComplexObject

TraverseComplexObject is very useful in manipulating a complex object as a whole. It is a very general and powerful function and as a result it expects the user to provide some guiding information (in the form of few lines of code) as it traverses the complex object.

For our chip database let us discuss how TraverseComplexObject can be used. As we mentioned in previous chapters, TraverseComplexObject visits all objects of a complex object and executes a specified (user-written or library) function for each one of them. For each object the user should provide all the relationship properties that TraverseComplexObject should visit.

```
TraverseComplexObject(Rootcell,Process_Object);

 Process_Object(object)
 OBJECT object;
 {
  /* Do whatever is to be done with the object, like displaying it on the display
    devices or compiling statistics, checking design rules, applying
    locks, etc. Functions like VisitMembers and VisitOwners can sometimes
    be useful for this purpose.
  */
```

```
Provide_Relationship_Properties(object,name_list)
OBJECT object;
struct name_list_header *name_list;
{
 object_class = GetObjectClass(object);
 switch(object_class) {
  case (cell) : fill_except_Crossinfo(object,name_list);
            return;
  case (SYSTEM): very_specific_processing(object,name_list);
             return;
  default   : retrieve_all_rel_properties(object,name_list);
             return;
  }
```

TraverseComplexObject initially visits the root cell and executes the function Process_Object. Then it executes the function Provide_Relationship_Properties (this name is built into TraverseComplexObject). This function, based on the object class to which the object belongs, provides the necessary relationship properties. For object class "cell" we do not want the property (of type SET) "Crossinfo" to be visited by TraverseComplexObject. This is taken care of by the user-written function fill_except_Crossinfo. For object classes "transistor", "net","wire", "cut", "signal", "extern" the system function retrieve_all_rel_properties is used to fill in all the relationship properties of the objects. The fact that the object classes "wire", "cut", "signal" and "extern" do not have any relationship properties are made known to TraverseObject through retrieve_all_rel_properties. For object class SYSTEM the user-written function very_specific_processing takes care of filling in the relationship properties for these objects.

6.4. Summary

In this chapter, we have provided practical examples to illustrate how to use our system. The data definition of a VLSI chip database is specified in DCSL. Some practical situations, when the facilities provided for dynamic restructuring of the database can be helpful, are illustrated. Examples of how to use some of the functions provided by the systems are provided. Thus this chapter illustrates the various aspects of our system including derived properties, constraint specifications, the facilities for dynamic restructuring of the database, how a complex object is represented, and how the function TraverseComplexObject can be used to manipulate a complex object as a whole. The next chapter provides the conclusions and a discussion on the scope for further research in this area.

Chapter 7

Conclusions

In this chapter, we summarize the contributions of this thesis, point out some limitations of the present design and discuss the scope for further research in this area.

7.1. Summary of Contributions

This thesis is concerned with the analysis of the database requirements of design environments, and the design and implementation of a database management system suitable for the same. First, we presented a model of the design activity. Such a model is very useful to put in proper perspective computer-aided design in general and the requirements of a design database management system in particular. We then presented a comprehensive set of database requirements for computer-aided design applications. We chose four of these requirements for the purposes of designing and implementing a prototype design database management system. The design is aimed at providing the facilities to satisfy the following requirements:

-Representation and Manipulation of Complex Objects.

-Dynamic Restructuring of the Database.

-Support for Derived Data.

-Flexible Constraint Specifications.

Then we presented the implementation details of our system and some real life examples to illustrate how our system is used.

We say that our system is "Object Oriented" since the concept of "Object" is the means by which the real world entities are modeled. We now briefly describe how the facilities offered by our system satisfy the four requirements mentioned above.

The type SET and the operators TraverseComplexObject, VisitMembers and VisitOwners make the representation and manipulation of complex objects possible. The dynamic restructuring of the database is achieved by the provisions to add and delete a property, and to add and delete an object class. The concept of a Derived Property provides the support for derived data The various "actions" provide a flexible and powerful constraint specification mechanism. In addition, the action IFNULL can be used to capture the semantics of the NULL value. In our view, constraint specification is not just specifying when to allow or disallow updates to the database. According to the circumstances, alternate (corrective) measures may need to be taken. Using the various actions, such measures can be specified.

7.2. Limitations

The design we have presented can be described as the first and preliminary version of our system. Two major features are missing from the present design. First, there are no facilities to specify or modify the various actions for an already existing database and there are no facilities for specifying them dynamically. Presently the actions can only be specified through DCSL, i.e. the actions can be specified only when the initial static structure of the database is specified. Since the actions are C language statements, there is a problem of making the dynamically specified or modified actions available in the present run of the program. An immediate solution is to extend DCSL (or design another tiny language) to specify statically the actions (and possibly updates to the structure and contents of the database) for an already existing database Second, the actions IFNULL and WHENMODIFIED can not be specified for the property SETDATA (the action WHENDELETED has no meaning for this property since this property can never be deleted by the user). The main reason why facilities to specify these actions are not provided, is that the property SETDATA comes into existence only when a member is added to a set. Specifying these actions statically has only limited meaning since these actions, then, have to be associated with the tag of the set. However this can possibly be implemented as an immediate solution. We mentioned before that DCSL presently ignores the "SETDATA OF" specification. When the facilities to specify these actions for SETDATA are implemented, the name of the object class specified after the "SETDATA OF" should be stored and made accessible to the WHENMODIFIED action for monitoring the value of the property.

7.3. Suggestions for Further Research

In this section, we discuss the possibilities for further research in this area.

7.3.1. First, the usefulness of the system should be studied by using it in a large scale reallife situation. This experience can be used to pragmatically evaluate the various design decisions and hence modify them appropriately, if deemed necessary. Important questions like

- Is the flexibility allowed for the members of an object class really necessary?
- Does this flexibility have any serious impact on the overall understandability of the structure of the database?

should be answerable after such practical experience with the system:

7.3.2. The constraint specification mechanisms should be studied in further detail. In our system, the constraints are associated with individual object classes, objects and properties. This might prove to be an overkill. This issue should be studied further to determine if there is any loss of generality, in the practical sense, if, say, the constraints of the properties of an object are generalized into one constraint associated with the object.

7.3.3. The other requirements mentioned in Chapter 3 should be studied to determine how they can be, if possible, incorporated into the system. The requirements like handling multiple representations, configuration control and version control should be given priority for such a study. The requirements like archiving of data at object level, object libraries and configuration control should be studied to determine if they can be implemented using the support provided by complex objects. Efforts should be taken to determine how our system and the University of Alberta UIMS[26] could communicate with each other for the purposes of building user interfaces for the design tools.

7.3.4. We have already stressed the importance of the model of the design activity. A model better than the one we have presented should be developed. Presently the model does not represent the following two aspects of the design activity: the interactions of subgoals during the synthesis task and the verification task to verify if a level of abstraction is correctly represented by its detailed abstraction at the lower levels. Also, the model should be revised to more explicitly and accurately depict the design activity in a computerized design environment. In addition, models of individual design activities, such as VLSI and software design, should be developed. These models can greatly help in the design of the database for these applications. Thus, the general model of the design activity can be useful for the design of the database management system whereas the models of the individual design applications can be useful for the design of the databases for these applications.

References

1. ANSI/X3/SPARC, Study Group on Data Base Management Systems: Interim Report, *FDT, Bulletin of ACM - SIGMOD* 7, 2 (1977).

2. M. M. Astrahan et al, System R: Relational Approach to Database Management, *ACM Transactions on Database Systems 1*, 2 (June 1976).

3. J.D. Ichbiah et al. in *Reference Manual for the Ada Programming Language*, Department of Defence, Honeywell Inc., and Alsys, July 1982.

4. M. Atkinson and N. Wiseman, Data Management Requirements for Large Scale Design and Production, *ACM-SIGDA Newsletter 7*, 1 (March 1977), 2-16.

5. J. Bennett, A Database Management System for Design Engineers, *Proceedings of the 19th Design Automation Conference,Las Vegas,Nevada*, June 1982, 268-273.

6. G. Booch, in *Software Engineering With Ada*, Benjamin/Cummings, New York 1983.

7. A. Borgida, J. Mylopoulos and H.K.T. Wong, Generalization/Specialization as a Basis for Software Specification, in *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, M.L. Brodie, J. Mylopoulos and J.W. Schmidt (ed.), Springer-Verlag, 1984, 87-117.

8. M.L. Brodie and D. Ridjanovic, On the Design and Specification of Database Transactions, in *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, M.L. Brodie, J. Mylopoulos and J.W. Schmidt (ed.), Springer-Verlag, 1984, 277-312.

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

10. O.P. Buneman and E.K. Clemons. Efficiently monitoring relational databases. *ACM Transactions on Database Systems 4*, 3 (September 1979), 368-382.

11. B. Chandramouli, Computer Aided Design Tool for Object Oriented Software Design, *CMPUT 511 Course Project Report, University of Alberta*, Dec 1984.

12. P.P. Chen, The Entity-Relationshop Model-Towards a Unified View of Data , *ACM Transactions on Database Systems 1*, (1976), 9-36.

13. Kung-chao Chu, John P. Fishburn, Peter Honeyman and Y. Edmund Lien, Vdd - A VLSI Design Database System, *Engineering Design Applications, Proceedings of Annual Meeting, Sponsored By IEEE and ACM* , 1983, 25-37.

14. P. L. Crampi and J. D. Nash, Concepts in CAD Database Structures, *13th Design Automation Conference*, June 1976, 290-294.

15. E.F. Codd, A Relational Model for Large Shared Databanks, *Comm. ACM 13*, 6 (June 1970).

16. E.F. Codd, Relational Database: A Practical Foundation for Productivity, *Comm ACM 25*, 2 (February 1982), 109-117.

17. O.-J. Dahl and C.A.R. Hoare, Hierarchical Program Structures, in *Structured Programming*, O.-J. Dahl, E.W. Dijkstra and C.A.R. Hoare (ed.), Academic Press, New York, 1972, 175-220.

18. U. Dayal and P.A. Bernstein, On the Updatability of Relational Views, *5th International Conference on Very Large Databases*, 1980, 368-377.

19. K.P. Eswaran, Aspects of a trigger subsystem in an integrated database system, *Proc. 2nd International Conference on Software Engineering*, October 1976, 243-250.

20. M.T. Garrett and J.D. Foley, Graphics Programming Using a Database System with Dependency Declarations, *ACM Transactions on Graphics 1*, 2 (April 1982).

109-128.

21. R. A. Gingell, D. A. Risen and C. W. Rose, Database Concepts Used in the LOGOS Computer-Aided Design System, *ACM-SIGDA Newsletter 7*, 1 (March 1977), 17-32.

22. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison Wesley, 1983.

23. M. Green, M. Burnell, H. Vrenjak and M. Vrenjak, Experiences with a Graphical Data Base System, *Proceedings of Graphics Interface '83*, 1983, 257-270.

24. M. Green, FDB - A Frame Based Database System, *Department of Computing Science, University of Alberta*, 1984.

25. M. Green, Design Notations and User Interface Management Systems, in *User Interface Management Systems*, Springer-Verlag, 1985.

26. M. Green, The University of Alberta UIMS, *Proc. of SIGGRAPH'85*, 1985, 205-213.

27. M. Hardwick, Extending the Relational Database for Design Applications, *Proceedings of the 21st Design Automation Conference*, 1984, 110-116.

28. R. Haskin and R. Lorie, On Extending the Functions of a Relational Database System, *Proc. 1982 ACM-SIGMOD Conference on Management of Data, Orlando, Fl*, 1982.

29. M. N. Haynie, Tutorial: The Relational Data Model for Design Automation, *Proceedings of the 20th Design Automation Conference*, 1983, 599-607.

30. D.H.H. Ingalls, The Smalltalk-76 Programming System -- Design and Implementation, *Conference Reocrds of the Fifth Annual ACM Symposium on the Principles of Programming Languages*, January 1978, 9-16.

31. D.H.H. Ingalls, Design Principles Behind Smalltalk, *BYTE*, August 1981, 286-298.

32. S. C. Johnson, Yacc — Yet Another Compiler Compiler, Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey, July 1975.

33. H. R. Johnson, J.L. Schweitzer and E. R. Warkentine, A DBMS facility for Handling Structured Engineering Entities, *Engineering Design Applications, Proceedings of Annual Meeting, Sponsored By IEEE and ACM*, May 1983, 3-11.

34. R. H. Katz, A Database Approach for Managing VLSI Design Data, *Proceedings of the 19th Design Automation Conference, Las Vegas, Nevada*, June 1982, 274-282.

35. R.H. Katz and S. Weiss, Design Transaction Management, *Proceedings of the 21st Design Automation Conference*, 1984, 692-693.

36. B.W. Kernighan and D.M. Ritchie, in *The C Programming Language*, Prentice-Hall, Inc., 1978.

37. R. King and D. McLeod, A Unified Model and Methodology for Conceptual Database Design, in *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, M.L. Brodie, J. Mylopoulos and J.W. Schmidt (ed.), Springer-Verlag, 1984, 313-331.

38. L. L. Knappe, A Computer Oriented Mechanical Design System, *Proceedings of the Share Design Automation Workshops*, June 23-25, 1965.

39. D.M. Kroenke, in *Database Processing: Fundamentals, Design, Implementation*, Science Research Associates, Inc., 1983.

40. G. M. E. Lafue, Integrating Language and Database for CAD Applications, *Computer-Aided Design 11*, 3 (May 1979), 127-131.

41. M. E. Lesk, Lex — A Lexical Analyzer Generator, Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey, October 1975.

42. R. A. Lorie, Issues in Databases for Design Applications, in *File Structures for Databases for CAD*, J. Encarnacao and L. Krause (ed.), North - Holland

Publishing Company, 1982, 213-229.

13. R. A. Lorie and Wilfred Plouffe, Complex Objects and Their Use in Design Transactions, *Engineering Design Applications, Proceedings of Annual Meeting, Sponsored By IEEE and ACM*, 1983, 115-121.

14. David Maier, in *The Theory of Relational Databases*, Computer Science Press, 1983.

15. D. McLeod, K. Narayanaswamy and K. V. Bapa Rao, An Approach to Information Management for CAD/VLSI Applications, *Engineering Design Applications, Proceedings of Annual Meeting, Sponsored By IEEE and ACM*, 1983, 39-50.

16. A. S. Michaels, B. Mittman and C.R. Carlson, A Comparison of relational and CODASYL approaches to data-base management, *Comput. Surveys 8*, 1 (March 1976), 125-151.

17. N. Minsky, Another Look at Databases, *FDT, Bulletin of ACM, SIGMOD 6*, 4 (1974), 9-17.

18. J. Mylopoulos, P. A. Bernstein and H.K.T. Wong, A Language Facility for Designing Interactive Database-Intensive Applications, *ACM Transactions on Database Systems 5*, 2 (June 1980), 185-207.

49. T. Rentsch, Object Oriented Programming, *ACM SIGPLAN Notices Notices 17*, 9 (September 1982), 51-57.

50. C.E. Robinson, A Data Structure for a Computer Aided Design System, *Proceedings of the Share Design Automation Workshops*, May 16-19, 1966.

51. E.G. Schlechtendahl, CAD Process and System Design, in *Lecture Notes in Computer Science, Computer aided design, Modelling, Systems Engineering, CAD-Systems*, Springer-Verlag, 1980, 389-429.

52. C.H. Sequin, Managing VLSI Complexity. An Outlook, *Proc. of the IEEE 71*, 1 (January 1983).

53. T.W. Sidle, Weaknesses of Commercial Database Management Systems in Engineering Application, *Proceedings of the 17 Design Automation Conference, Mineapolis, MN*, June 1980, 57-61.

54. J. Smith and D. Smith, Database Abstractions: Aggregations and Generalizations, *ACM Transactions on Database Systems 2*, 2 (September 1977), 105-133.

55. David C. Smith and Barry S. Wagner, A Low Cost, Transportable, Data Management System for LSI/VLSI Design, *Proceedings of the 19th Design Automation Conference*, 1982, 283-290.

56. M. R. Stonebraker, The Design and Implementation of INGRES, *ACM Transactions on Database Systems 1*, 3 (September 1976).

57. M.R. Stonebraker, B. Rubenstein and A. Guttman, Application of Abstract Data Types and Abstract Indices to CAD Data Bases, *Engineering Design Applications Proceedings of Annual Meeting, Sponsored By IEEE and ACM*, 1983, 107-113.

58. G.M. Weinberg, in *The Psychology of Computer Programming*, Van Nostrand-Reinfold, 1971.

59. R. Wiener and R. Sincovec, in *Software Engineering with Modula-2 and Ada*, John Wiley & Sons, 1984.

60. N. Wirth, in *Programming in Modula-2*, Springer-Verlag, 1982.

61. H.R.T. Wong, Design and Verification of Interactive Information Systems using TAXIS, *Technical Report CSRG-129, Department of Computer Science, University of Toronto*, April 1981.

Appendix A1

Formal Definition of DCSL

In this appendix we give the formal definition of our data definition language DCSL. To keep the definition reasonably small and readable we have omitted the lexical level (terminal) definitions and have described them in comments.

The formal definition is written in the terminology of Yacc[32]. A grammar rule has the form:

Nonter : anynames;

Nonter represents a nonterminal name, and anynames represents a sequence of zero or more names and literals. A name is either a nonterminal or a terminal (token) symbol. The colon and the semicolon are the punctuations. If there are several rules with the same left hand side, the vertical bar "|" is used to avoid rewriting the left hand side. Comments are provided between the delimiters "/*" and "*/". We have followed the convention of writing the tokens (terminal symbols) in capital letters.

FORMAL DEFINITION OF DCSL

```
dcsl              : objdef dcsl | /* empty right hand side */ ;
objdef            : object_class_defn | object_defn;
object_class_defn : OBJECT CLASS /* OBJECT and CLASS  are key words*/
                    detailed_obj_defn;

object_defn       : OBJECT /* OBJECT is a key word */
                    detailed_obj_defn;

detailed_obj_defn : IDENTIFIER property_defn END;
        /* IDENTIFIER starts with a letter followed by a sequence
           of 0 or more letters, digits or "_". END is a key word */

property_defn     : a_property_defn property_defn | ;
```

```
a_property_defn   : IDENTIFIER ':' propertyspecs | spl_propertydefn;

propertyspecs     : primaryspecs  constraintdefn| derivedspecs constraintdefn;

derivedspecs      : DERIVED derived_type_value_specs
                  |DERIVED STATIC derived_type_value_specs ;

derived_type_value_specs :  typespecs '%'[ C_CODE '%'];
                  /* C_CODE is C language statements */

typespecs         : INTEGER | REAL | TEXT
                /* INTEGER, REAL and TEXT are key words */
                |setdefn | objectdefn;

setdefn           : SET setdataspecs | SET OF IDENTIFIER setdataspecs ;
                /* SET and OF are key words */

setdataspecs      : SETDATA OF IDENTIFIER | ; /* SETDATA is a key word*/

objectdefn        : OBJECT | OBJECT OF IDENTIFIER ;

primaryspecs      : INTEGER INTEGERVALUE | REAL REALVALUE
                |TEXT TEXTVALUE | setdefn SETVALUE
                |objectdefn OBJECTVALUE ;
   /* INTEGERVALUE - an integer with an optional sign
      REALVALUE    - a real number with an optional sign, a string of
                     numbers possibly containing a decimal point, and an
                     optional exponent field  containing an 'E' or 'e'
                     followed by a possibly signed integer
      TEXTVALUE    - a string enclosed in quotes
      SETVALUE     - one or more object names (IDENTIFIERs) separated by
                     comma or white spaces
      OBJECTVALUE  - a name of an object (IDENTIFIER)  */

constraintdefn    : ifnullspecs whenmodifiedspecs whendeletedspecs;

ifnullspecs       : IFNULL constraintspecs |;

whenmodifiedspecs : WHENMODIFIED constraintspecs |;

whendeletedspecs  : WHENDELETED constraintspecs |;
    /* IFNULL, WHENMODIFIED and WHENDELETED are key words */

constraintspecs   : '%'[C_CODE,'%'];
    /* C_CODE is C language statements */

spl_propertydefn  : spl_propertyname ':' spl_propertyvalue
                  | CLASS ':' IDENTIFIER
                  | REMOVE ~~':' removespecs~~; /* REMOVE is a key word */

spl_propertyname  : WHEN_DATABASE_DESTROYED
                  | WHEN_OBJECT_CLASS_CREATED
                | WHEN_OBJECT_CLASS_DELETED
                | WHEN_OBJECT_ADDED
                | WHEN_OBJECT_DELETED
                | WHEN_PROPERTY_ADDED;

spl_propertyvalue  : '%''[' C_CODE '%'']';

removespecs        : '%''[' PROPERTYNAMES '%'']';
    /* PROPERTYNAMES is a sequence of one or more property
       names (IDENTIFIERs) separated by comma or white spaces. */
```

Appendix A2
Declarations of Functions

In this appendix, we first explain how to run the program, called "dcsl", which processes the DCSL specifications. Then we provide the header declarations of the functions provided by the system, in the syntax of the programming language C.

1. To Run dcsl

The command to run the "dcsl" program is:

```
dcsl databasename dcslfilename
```

where databasename is the name of the database, to be given by the user, and dcslfilename is the file name containing the DCSL specifications. "dcsl" will create a database with the given name and build the initial structure of the database as specified in DCSL.

2. Header declarations of functions

The structure de larations that are relevant to the functions provided by the system are given in apter 5. Another important structure which is used by these functions is the data e structure "DATABASE". The various structure declarations and the various constant declarations like "OBJ", "OBJCLASS", "YES", "NO" etc., are in the file "DBMS.h" and this file should be included by all the application programs.

```
CreateDatabase(dbstruct,dbname)
DATABASE *dbstruct;
char *dbname;
```

CreateDatabase creates a database with the name "dbname" and initializes the database structure variable "dbstruct". This variable is used by the other functions to refer to the database.

```
DestroyDatabase(dbstruct)
DATABASE dbstruct;
```

DestroyDatabase destroys the database "dbstruct".

```
OBJECT NewObject(dbstruct,flag,objectclass)
DATABASE dbstruct;
int flag;
OBJECT objectclass;
```

This function creates a new object class or a new object. It returns the name of the object or object class created. It returns -1 if it can not be created. If the flag is "OBJ" then a new object is created and if the flag is "OBJCLASS" then a new object class is created. The properties, their initial values, and constraint specifications of the object class "objectclass" are inherited by the new object or object class.

```
DeleteObject(dbstruct,object)
DATABASE dbstruct;
OBJECT  object;
```

This function deletes an object or object class. It returns -1 if it can not be deleted. The parameter "object" is the name of the object or object class.

```
OBJECT GetObjectClass(dbstruct,object)
DATABASE dbstruct;
OBJECT object;
```

This function returns the object class of an object. The parameter "object" is the name of the object. If "object" refers to an object class then GetObjectClass returns -1.

```
SetObjectClass(dbstruct,object,objectclass)
DATABASE dbstruct;
OBJECT object,objectclass;
```

This function adds an object to an object class. The parameter "object" is the name of the object and the parameter "objectclass" is the name of the object class.

```
AddProperty(dbstruct,object,propertyname,propertytype)
DATABASE dbstruct;
OBJECT object;
char *propertyname;
int propertytype;
```

This function adds a (primary or derived) property to an object or object class. The parameter "object" is the name of the object or object class, the parameter "propertyname" is the name of the property and the parameter "propertytype" is the type of the property. The type is one of the following constants: INTEGER, REAL, TEXT, OBJECT, and SET for primary properties, and DER_INTEGER, DER_REAL, DER_TEXT, DER_OBJECT and DER_SET for derived properties. This function returns -1 if the property can not be added.

```
DeleteProperty(dbstruct,object,propertyname)
DATABASE dbstruct;
OBJECT object;
char *propertyname;
```

This function deletes a property from an object or object class. The parameter "object" is the name of the object or object class and the parameter "propertyname" is the name of the property.

```
SetValue(dbstruct,object,propertyname,value)
DATABASE dbstruct;
OBJECT object;
char *propertyname;
union value_header *value;
```

This function stores the value of a property. The parameter "object" is the name of the object or object class, the parameter "propertyname" is the name of the property and the parameter "value" is the new value of the property. For derived properties, the new value is stored as the "static" value of the property.

```
GetValue(dbstruct,object,propertyname,value)
DATABASE dbstruct;
OBJECT object,
char *propertyname;
union value_header *value;
```

This function retrieves the value of a property. The parameter "object" is the name of the object or object class, the parameter "propertyname" is the name of the property and the parameter "value" is the new value of the property.

```
SetStatic(dbstruct,object,propertyname,flag)
DATABASE dbstruct;
OBJECT object;
char *propertyname;
int flag;
```

This function is used with derived properties. This function sets the "static" status of a derived property. If the value of the parameter "flag" is 1 then the derived property is set to "static", otherwise it is set to "non static".

```
IsitStatic(dbstruct,object,propertyname)
DATABASE dbstruct;
OBJECT object;
char *propertyname;
```

This function is used with derived properties. This function returns 1 if the derived property is static; otherwise it returns 0.

```
GetType(dbstruct,object,propertyname)
DATABASE dbstruct;
OBJECT object;
char *propertyname;
```

This function returns the type of a property. The parameter "object" is the name of the object or object class of the property, the parameter "propertyname" is the name of the property. The type name returned is one of the following constants: INTEGER, REAL, TEXT, OBJECT and SET for primary properties, and DER_INTEGER, DER_REAL, DER_TEXT, DER_OBJECT and DER_SET for derived properties.

```
OBJECT GetObjectClass_PROPERTY(dbstruct,object,propertyname)
DATABASE dbstruct,
OBJECT object;
char *propertyname;
```

This function returns the name of the object class that is specified for the properties of type SET and OBJECT. The parameter "object" is the name of the object and the parameter "propertyname" is the name of the property of type SET or OBJECT.

```
SetObjectClass_PROPERTY(dbstruct,object,propertyname,objectclass)
DATABASE dbstruct;
OBJECT object;
char *propertyname;
OBJECT objectclass;
```

This function stores the name of the object class for the properties of type SET and OBJECT. The parameter "object" is the name of the object and the parameter "propertyname" is the name of the property and the parameter "objectclass" is the name of the object class.

```
VisitObjects(dbstruct,objectclass,func)
DATABASE dbstruct;
OBJECT objectclass;
int (*func)();
```

The parameter "objectclass" is the name of the object class and the parameter "func" is a pointer to a function returning an integer (the actual parameter will be the name of a function returning an integer). VisitObjects visits the objects of an object class and executes the function pointed to by "func". VisitObjects passes the name of the object as parameter to this function. If the function pointed to by "func" returns -1 for an object then VisitObjects quits visiting the rest of the objects in the object class

```
VisitObjectClasses(dbstruct,func)
DATABASE dbstruct;
int (*func)();
```

The parameter "func" is a pointer to a function returning an integer. VisitObjectClasses visits each object class in the database and calls the function pointed to by "func" with the name of the object class as parameter. If this function returns -1 for any object class then VisitObjectClasses quits visiting the rest of the object classes in the database.

The following functions are specific to a property of type SET. The value of a property of type SET is a set of objects. In the following functions, the parameter "owner" is the name of the owner of the set, the parameter "tag" is the tag of the set (i.e the name of the property of type SET), the parameter "member" is the name of a member of the set and the parameter "func" is a pointer to a function returning an integer

```
AddMember(dbstruct,owner,tag,member)
DATABASE dbstruct;
OBJECT owner;
char *tag;
OBJECT member;
```

This function adds a member to a set.

```
DeleteMember(dbstruct,owner,tag,member)
DATABASE dbstruct;
OBJECT owner;
char *tag;
OBJECT member;
```

This function deletes a member from a set.

```
OBJECT GetSETDATA(dbstruct,owner,tag,member)
DATABASE dbstruct;
OBJECT owner;
char *tag;
OBJECT member;
```

This function returns the value of SETDATA, i.e the relationship information between the owner and the member of a set

```
OBJECT SetSETDATA(dbstruct,owner,tag,member,value)
DATABASE dbstruct;
OBJECT owner;
char *tag;
OBJECT member;
OBJECT value;
```

It stores the value of SETDATA, i.e the relationship information between the owner and the member of a set. The parameter "value" is the value of SETDATA.

```
VisitMembers(dbstruct,owner,tag,func)
DATABASE dbstruct;
OBJECT owner;
char *tag;
int (*func) ();
```

VisitMembers visits each member of the set and calls the function pointed to by "func" with the owner, member and tag as parameters. If this function returns -1 for a member then VisitMembers quits visiting the rest of the members.

```
VisitOwners(dbstruct,member,tag,func)
DATABASE dbstruct;
OBJECT member;
char *tag;
int (*func) ();
```

If the value of the parameter "tag" is the string "_ALL" then VisitOwners visits all owners of the member and calls the function pointed to by "func" with the owner, member and the tag as parameters. Otherwise, it visits each of the owner object with which the member is in "SIMILAR RELATIONSHIP" (through the tag) and calls the function pointed to by "func" with the owner, member and the tag as parameters. If this function returns -1 for an owner object then VisitOwners quits visiting the rest of the owners.

```
TraverseComplexObject(dbstruct,object,func)
DATABASE dbstruct;
OBJECT object;
int (*func) ();
```

The parameter "object" is the name of the root object of a complex object. It visits systematically all the objects of the complex object and calls the function pointed to by "func" with the object name as parameter. If this function returns -1 for an object then TraverseComplexObject does not visit the members of the relationship properties of that object. For each object it then calls a user-written function "Provide_Relationship_Properties" to get those relationship properties of the object which participate in the traversal. A sample header declaration of this function is given below.

```
Provide_Relationship_Properties(dbstruct,object,list)
DATABASE dbstruct;
OBJECT object;
struct name_list_header *list;
```

The declarations of "name_list_header" is given in Chapter 5. If the user wants all the relationship properties of the object to participate in the traversal then he can use the function "retrieve_all_rel_properties" which is provided by the system. The parameters to this function are the same as Provide_Relationship_Properties.

3. Parameters to the Deriving and Constraint functions

We have mentioned in Chapter 5 that an action is formed into a C function and the function is given a unique name. We summarize here the names of the formal parameters to these functions. In the following declarations this unique name of a function is refered to by the name "uniquename".

3.1. The Deriving Functions

```
uniquename(dbstruct,object,property,derivedvalue)
DATABASE dbstruct;
OBJECT object;
char *property;
union value_header *derivedvalue;
```

The parameter "object" is the name of the object containing the derived property, the parameter "property" is the name of the derived property and the parameter "derivedvalue" is for returning the derived value of the property. This function should return -1 (using the "return" statement) if the value can not be derived, to indicate to GetValue that the value is NULL.

3.2. The Constraint Functions

3.2.1. IFNULL

```
uniquename(dbstruct,object,property,defaultvalue,defaultset)
DATABASE dbstruct;
OBJECT object;
char *property;
union value_header *defaultvalue;
int *defaultset;
```

The parameter "object" is the name of the object containing the property, the parameter "property" is the name of the property, the parameter "defaultvalue" is for returning the default value. This function should assign YES to the parameter "defaultset" to indicate if a default value has been provided and should assign NO otherwise. The interpretation of the NULL value is returned using the "return" statement.

3.2.2. WHENMODIFIED

```
uniquename(dbstruct,object,property,value)
DATABASE dbstruct;
OBJECT object;
char *property;
union value_header *value;
```

The parameter "object" is the name of the object, the parameter "property" is the name of the property and the parameter "value" is the new value for the property.

3.2.3. WHENDELETED

```
uniquename(dbstruct,object,property)
DATABASE dbstruct;
OBJECT object;
char *property;
```

The parameter "object" is the name of the object and the parameter "property" is the name of the property

3.2.4. WHEN_PROPERTY_ADDED

```
uniquename(dbstruct,object,property,type)
DATABASE dbstruct;
OBJECT object;
char *property;
int type;
```

The parameter "object" is the name of the object or object class to which the property is added, the parameter "property" is the name of the property and the parameter "type" is the type of the property.

3.2.5. WHEN_OBJECT_DELETED

```
uniquename(dbstruct,object)
DATABASE dbstruct;
OBJECT object;
```

The parameter "object" is the name of the object which is deleted

3.2.6. WHEN_OBJECT_ADDED

```
uniquename(dbstruct,object,objectclass)
DATABASE dbstruct;
OBJECT object;
OBJECT objectclass;
```

The parameter "object" is the name of the object and the parameter "objectclass" is the name of the object class to which the object is added. If the object is a new object then the value of the parameter "object" is 0.

3.2.7. WHEN_OBJECT_CLASS_DELETED

```
uniquename(dbstruct,objectclass)
DATABASE dbstruct;
OBJECT objectclass;
```

The parameter "objectclass" is the name of the object class which is deleted.

3.2.8. WHEN_OBJECT_CLASS_CREATED

```
uniquename(dbstruct)
DATABASE dbstruct;
```

3.2.9. WHEN_DATABASE_DESTROYED

```
uniquename(dbstruct)
DATABASE dbstruct;
```

Appendix A3

Sample Constraint Functions

In this appendix we provide the C functions "checktransistortype()" and "checklocation()" that are used in the WHENMODIFIED constraint functions in Chapter 6. "checklocation()" is used for the property "location" of the object classes "cut_cell" and "extern_cell", and "checktransistortype()" is used for the property "transistortype" of the object class "transistor". With the comments provided in the body of the functions, they should be fairly self-explanatory.

```
checktransistortype(transistor_type)
char *transistor_type;
{

/* If the transistor type is NTYPE or PTYPE return YES
  else return NO. */

 if (strcmp(transistor_type,"NTYPE") == 0 ||
    strcmp(transistor_type,"PTYPE") == 0)
  return(YES) else return(NO);
}
```

```
checklocation(owner,object,property,newvalue)
OBJECT owner,object;
char *property;
OBJECT newvalue;
{
  union value_header oldvalue;
  union value_header oldXvalue;
  union value_header oldYvalue;
  union value_header newXvalue;
  union value_header newYvalue;
  union value_header ownervalue;
  int objectclass;
  float length, width;

 /* Let us assume that there is a general restriction that the new
   coordinates should be within 2 units of the
   old coordinates */

 /* Get the old coordinates */
```

```
  GetValue(object,property,&oldvalue);

  GetValue(oldvalue.object_value,"Xcoordinate",&oldXvalue);
  GetValue(oldvalue.object_value,"Ycoordinate",&oldYvalue);
  GetValue(newvalue,"Xcoordinate",&newXvalue);
  GetValue(newvalue,"Ycoordinate",&newYvalue);

 /* check for general restriction */

  if( fabs(oldXvalue.real_value - newXvalue.real_value) > 2 ||
       (oldYvalue.real_value - newYvalue.real_value) > 2)
   return(NO);

 /* Now check for specific restrictions */

 /* Get the dimensions of the surrounding cell */

  GetValue(owner,"BoundingBox",&ownervalue);
  GetValue(ownervalue.int_value,"length",&length);
  GetValue(ownervalue.int_value,"width",&width);

 /* Assume that the coordinates are with respect to the
   surrounding cell. For both cut and extern, their
   location can not be outside the surrounding cell */

 if (NewXvalue.real_value < 0.0 || NewXvalue.real_value > length)
   return(NO);
 if (NewYvalue.real_value < 0.0  || NewYvalue.real_value > width)
   return(NO);

 objectclass = GetObjectClass(object);
 switch (objectclass) {

/* for cut, only restriction is that its location should be
  inside the surrounding cell.      */

   case(cut_cell) :  return(YES); break;

 /* for extern, its location should be  on the sides of
   the surrounding  cell */

   case(extern_cell) :
 /* Return YES if it is on anyone of the sides else return NO */

   if (NewXvalue.real_value == 0.0 ||
      NewYvalue.real_value == 0.0 ||
      NewXvalue.real_value == width ||
      NewYvalue.real_value == length) return(YES);
   else return(NO); break;
  }
}
```