Design and Implementation of an OODBMS for VLSI Interconnect Parasitic Analysis

A Thesis

Submitted for the Degree of

Master of Science (Engineering)

in the Faculty of Engineering

by

Arun N. S.

Department of Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

July 1996

Contents

Acknowledgements **v**

Abstract **vi**

1 Introduction **1**
1.1 Motivation 1
1.2 Current data management solutions 3
1.3 Database Management Systems 4
1.4 The DIAS System 6
1.5 Earlier Work 7
1.6 Organization 8

2 Database Modeling Approaches **9**
2.1 Application Requirements 10
2.2 Drawbacks of traditional database systems 12
2.3 The Object Oriented data model 13
2.4 Object Oriented Database Management Systems 15
2.5 Choice for DIAS 16

3 OODBMS Platforms **18**
3.1 Object Oriented database programming languages 21
3.1.1 ODE 21
3.1.2 ObjectStore 22
3.1.3 GemStone 24
3.1.4 O_2 25
3.1.5 ONTOS 27
3.2 Extended relational database systems 28
3.2.1 POSTGRES 28
3.2.2 Illustra 30
3.2.3 STARBURST 31
3.3 Database System Generators 32
3.3.1 EXODUS 32
3.3.2 SHORE 34
3.3.3 GENESIS 35

3.4 Object Managers . . . 37
3.4.1 Mneme . . . 37
3.4.2 POMS . . . 38
3.5 Choice for DIAS . . . 39

4 SHORE 40
4.1 Introduction . . . 40
4.2 Basic Shore Concepts . . . 41
4.3 Shore Object Basics . . . 41
4.4 File System Features . . . 42
4.4.1 Shore Object namespace . . . 42
4.4.2 Legacy Unix tool support . . . 43
4.5 Object Oriented Database System features . . . 43
4.5.1 The Shore Type System . . . 43
4.5.2 Shore Language Bindings . . . 44
4.5.3 Other OODB-Like services . . . 44
4.6 The Shore Architecture . . . 45
4.7 Shore Software Components . . . 45
4.7.1 The Language Independent Library . . . 45
4.7.2 The Shore server . . . 46
4.8 Conclusion . . . 48

5 Object Oriented Methodologies 49
5.1 Introduction . . . 49
5.2 Traditional Approaches . . . 49
5.3 The Object Modeling Technique . . . 50
5.4 The Booch Method . . . 51
5.5 The Fusion Method . . . 53
5.6 Other Object Oriented Approaches . . . 54
5.7 Comparison of OO Methodologies . . . 54
5.8 Choice for DIAS . . . 55

6 Analysis 60
6.1 Problem Statement . . . 61
6.2 Identifying Classes . . . 62
6.3 Relationships . . . 65
6.4 Link Attributes . . . 70
6.5 Other Relationships . . . 72
6.6 Attributes and Operations of Classes . . . 72
6.7 Inheritance . . . 72
6.8 Normalization of the Object Model . . . 74

7 Design and Optimization 79
7.1 DIAS System Architecture . . . 79
7.2 Object Design . . . 82
7.2.1 Obtaining operations on classes . . . 82
7.2.2 Designing algorithms to implement operations . . . 82
7.2.3 Design of associations . . . 82
7.3 Additional classes . . . 83
7.4 Further optimization . . . 85
7.5 Access methods . . . 85
7.6 Index Organizations for Aggregation Hierarchies . . . 87
7.6.1 The Nested index . . . 88
7.6.2 The Path index . . . 90
7.7 Index Organizations for Association Paths . . . 91
7.8 Index Organizations for Inheritance Hierarchies . . . 91
7.8.1 The CH tree . . . 93
7.8.2 The hcC tree . . . 94
7.9 Unified Index Organizations for Composite Hierarchies . . . 96
7.10 Efficient execution of queries . . . 99

8 DIAS Kernel Implementation 102
8.1 Object Model implementation . . . 102
8.2 Data Manipulation . . . 106
8.3 Access Methods . . . 109
8.3.1 B^+ tree . . . 109
8.3.2 Nested Index . . . 109
8.3.3 Path Index . . . 110
8.3.4 CH tree . . . 110
8.3.5 hcC tree . . . 111
8.4 Other implementation details . . . 112

9 Interfaces 113
9.1 Tool Interface . . . 113
9.2 Programming Interface . . . 114
9.3 Query Interface . . . 117

10 The Query Processing System 120
10.1 Schema Manager . . . 121
10.2 Query Processor . . . 122
10.2.1 Interface Server . . . 123
10.2.2 Query Evaluator . . . 123
10.2.3 Query Executor . . . 125

11 Performance **127**
11.1 Bulk transfer of legacy data . . . 128
11.2 Access of specific data . . . 129
11.3 Impact of access methods on query processing . . . 129
11.3.1 Simple value based queries . . . 130
11.3.2 Queries targeted on hierarchies . . . 131

12 Summary and Future Work **135**
12.1 Our contributions . . . 136
12.2 Future Work . . . 138
12.3 Final Remarks . . . 139

Bibliography **139**

A SDL Declarations **147**

B The Rumbaugh Notation **160**

C DIAS Object Model **162**

Acknowledgements

I would like to thank my research supervisor, *Dr. Jayant Haritsa*, who has been a constant source of encouragement throughout the course of the work. I am grateful to him for his readiness to discuss at all times and his continuous support. His professionalism and attention to detail have been a great source of inspiration. The Database Systems Lab, with its abundant facilities and wonderful environment, has contributed a large extent to the final shape of the work.

My sincere thanks go to Krishna, for all his suggestions and ideas. His patience and understanding have helped me a lot. I would like to thank Bhuvana, Binto and Ramesh for all their help. Their presence created a co-operative atmosphere in the Lab. Thanks to the Parasitic Work Bench team of Texas Instruments, India for their technical inputs.

Finally, I would like to thank my parents for all the patience and encouragement that they have shown towards me.

Abstract

Right from the inception of integrated circuit fabrication in the early sixties, efforts have been made to improve chip performance, in terms of speed, density and power, by reducing the feature size. In fact, recent press reports mention new chip manufacturing processes that have less than 10 nanometre resolution. Due to the increased complexity, *interconnects* have begun to occupy a considerable portion of the layout in modern VLSI chips. A related consequence is that *interconnect parasitics*, which are unintended harmful sideeffects that arise as a consequence of the layout, can potentially have a significant adverse impact on chip performance. Therefore, the electrical characteristics of these parasitics have to be taken into account in IC chip design.

The study of interconnect parasitics is a four-stage process involving extraction of parasitic information from the circuit description, modeling of the parasitics, incorporation of the parasitics in the netlist and, finally, analysis of performance parameters. The entire process involves processing data that is both large in size and complex in nature. Current CAD systems typically use file-based or text-based approaches to parasitic data management and this has proved to be unsatisfactory in terms of modeling power, modularity, extensibility, functionality, querying, and speed of data processing. Therefore, the current interconnect analysis solutions form a serious bottleneck in the chip design process.

In this thesis, we have addressed the above problem of parasitic data management by using the recently developed object oriented database technology. We present a detailed object model for representing interconnect parasitics. We also describe the design and implementation of DIAS, a database system that implements the object model. DIAS has been developed following the OMT (Object Modeling Technique) methodology and uses the SHORE extensible system as the backend database engine for storage management and transaction support. For efficient object access, several access methods such as the CH-tree and the hcC-tree for inheritance hierarchies, the Path Index and the Nested Index for aggregation hierarchies, and the B^+-tree for simple attribute values are provided.

Users of DIAS can interact with the system through a variety of interfaces. The Programming Interface consists of a library of routines that can be used to manipulate the database. These routines can be used for developing new design tools that directly use the database system. A Query Interface to dynamically formulate queries is also provided. DIAS supports existing file-based design tools (legacy tools) through a Tool Interface using SHORE's file-system features. Thus, DIAS forms part of an information framework that extends chip design database systems to include support for analysis of interconnect parasitics. To the best of our knowledge, this system represents the first object oriented database-related work in the area of interconnect parasitic analysis. DIAS has been developed in close co-operation with the designers of Texas Instruments, India, where it is currently operational.

Chapter 1

Introduction

1.1 Motivation

Right from the inception of integrated circuit fabrication in the early sixties, efforts have been made to improve chip performance, in terms of speed, density and power, by reducing the feature size. In fact, recent reports mention new chip manufacturing processes that have less than 10 nanometre resolution [IEEE96]. The increase in circuit complexity has led to the use of interconnections spread over several metal layers (sometimes as many as five to seven metal layers). As a result, these interconnects have begun to occupy a considerable portion of the layout in modern VLSI chips. A related consequence is that *interconnect parasitics*, which are unintended harmful side-effects that arise as a consequence of the layout, can have a potentially significant adverse impact on chip performance. Therefore, the electrical characteristics of these parasitics need to be taken into account in the design of modern IC chips. This requirement is borne out by recent studies which have found that the delay due to interconnects may cause upto 70 percent of the total signal delay [RaPi94, Khan91] thereby forming the primary bottleneck in chip performance.

In the chip design process, taking interconnect parasitics into account involves a four-stage process, as shown in Figure 1.1. In the first step, parasitic information is extracted from the original circuit design. In the next phase, these parasitics are represented using modeling techniques such as, for example, RC trees or transmission lines. This data is then incorporated into the circuit netlist. An example circuit is shown in Figure 1.2 where the interconnect parasitics are modeled as resistances and capacitances. Finally, the circuit is analysed for electrical parameters such as timing, power dissipation, etc. Based on these results, the circuit design is revised and the interconnect analysis is repeated. This feedback process continues until a circuit with satisfactory electrical characteristics is realized.

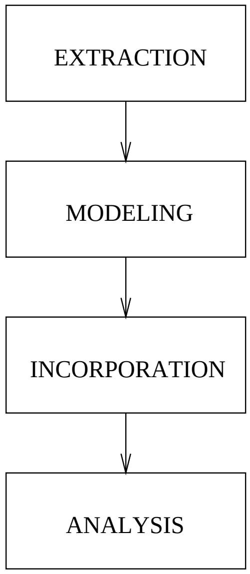


Figure 1.1: The Phases of Interconnect Study

All of the above stages of interconnect study involve processing data that is both large in size and complex in nature. For example, the parasitic data is in the order of Megabytes even for simple chip designs. Moreover, there are rich semantic relationships in the data such as "part-of" (containment), "is-a" (inheritance), etc. For example, a Cell contains ports, a Cell is a Library Cell or a Design Cell, etc. The above-mentioned features of parasitic data indicate a clear need for providing data management support tools.

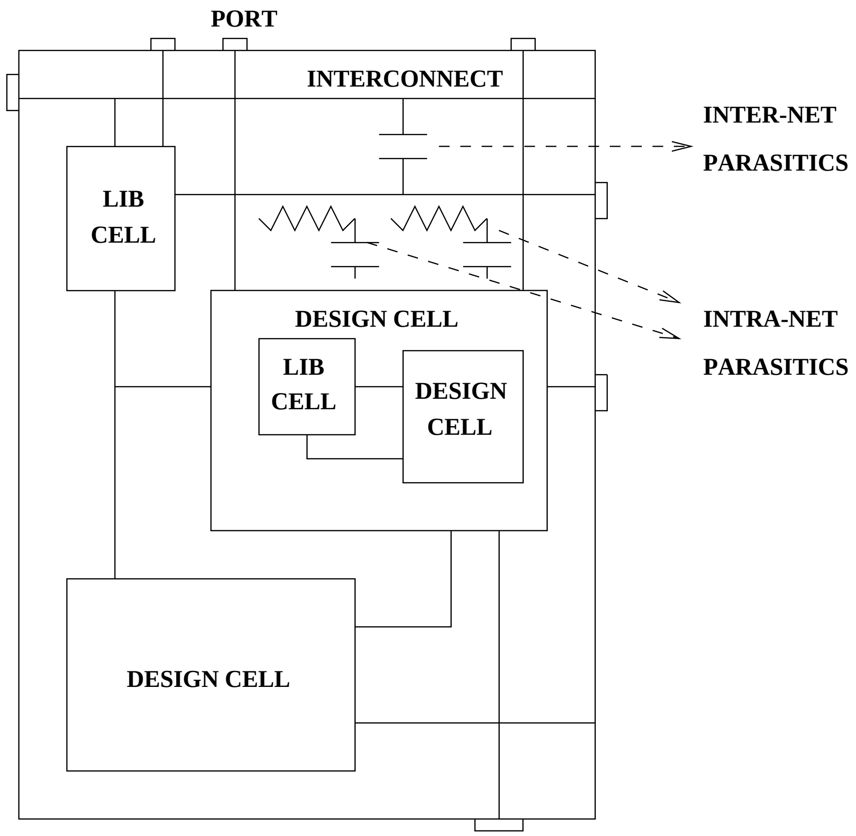


Figure 1.2: A circuit showing interconnect parasitics

1.2 Current data management solutions

Current chip design systems are typically managing their interconnect data using a custom-built *file-based* approach. This approach is turning out to be unsatisfactory for the following reasons:

- The system does not provide a natural way of modeling complex objects.
- The whole file has to be scanned even if the relevant data constitutes only a small portion of it.
- Any changes to the file structure require rewriting of all applications.
- Extending the system to handle additional parameters is difficult.

- There are no inbuilt mechanisms either for maintaining database consistency during concurrent access or for providing automatic crash recovery.

Some current chip design systems are not only file-based but are also *text-based* in that all data is stored in textual form. For these systems, the problems of the file-based approach are further exacerbated because of the following reasons:

- Data processing is slow due to textual parsing.
- The database is unnecessarily large since all data types are stored in textual format.
- The entire database has to be parsed and stored in memory before data processing can be initiated.

From the above discussion, it is evident that current interconnect data management solutions form a serious bottleneck in the chip design process in terms of both functionality and performance. Since chip complexity usually doubles every two years, the problem will only worsen with the course of time.

1.3 Database Management Systems

To effectively manage the ever-increasing vastness and complexity of these data, an alternative to the file-based approach is to employ a *database technology based* approach. In this scenario, various tools interact with the Database Management System (DBMS) in each of the stages providing inputs as well as retrieving data to feed the next stage, as shown in Figure 1.3.

Using a database system provides the following benefits:

- It is designed to handle data of arbitrary size and complexity.
- It ensures that only the relevant parts of the database are accessed.
- It serves as a single, central repository eliminating redundancy of multiple copies of the same data.

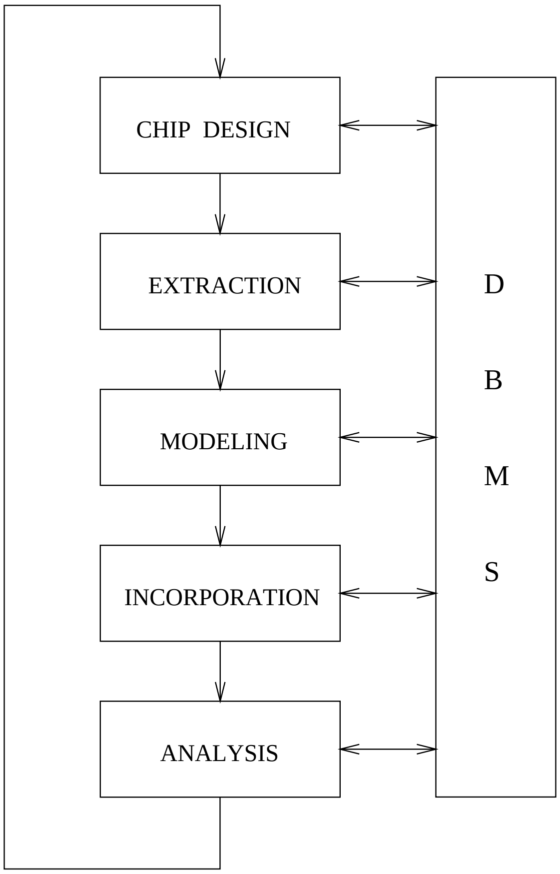


Figure 1.3: DBMS solution for interconnect parasitic data management

- It stores the data in an application independent form.
- It provides an interface to the stored data which can be used to write specific applications irrespective of the form in which the underlying data is stored.
- Desirable features such as automatic recovery of data after failures and concurrency control are now the responsibility of the DBMS rather than that of application programs.
- Special access methods can be incorporated to query the data efficiently.

Most of the commercial database systems in use today (e.g., Oracle, Ingres, etc.) are based on the *relational* data model, where all data is represented in the form of tables. However, this model has been found to be inadequate to naturally support complex applications such as VLSI CAD. A solution to the problem has recently emerged with the development of database systems based on the *object oriented* modeling paradigm, which provides a powerful and natural vehicle for representing domains that are rich in semantic relationships.

1.4 The DIAS System

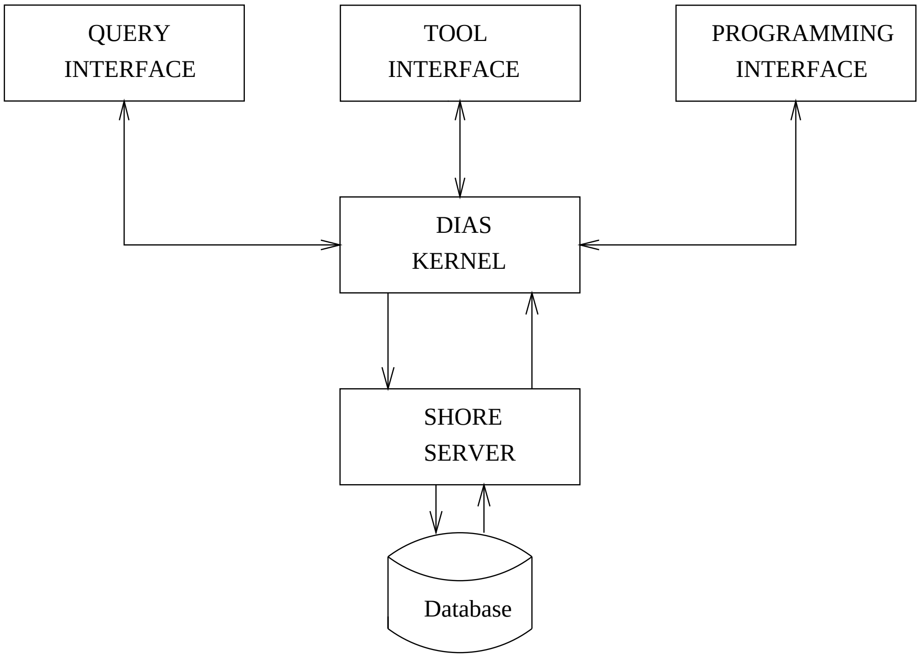


Figure 1.4: DIAS System Overview

In this thesis, we present the design and implementation of **DIAS** (Database System for Interconnect Analysis), a system that is tuned for supporting interconnect parasitics data. In particular, we first present a comprehensive *object oriented* model for capturing the complex semantics of the domain. The model represents in detail the variety of data relating to design and analysis of a VLSI circuit. The design data consists of data regarding the various components of a circuit design and the complex interrelationships among them. The analysis data deals with the data introduced during the interconnect study of the designed circuit. In addition, the model also represents layout and characterization information.

We then describe the design and implementation of a database system that implements the model and supports efficient data processing. DIAS has been developed for use in the chip development process of Texas Instruments, Inc., but its design is broadly applicable to other environments as well. DIAS is an object oriented database solution developed using Rumbaugh's OMT methodology of software development [Ru+91]. DIAS uses the SHORE system [Zwi+94] as the backend database engine, and has been implemented using a combination of the Shore Data Language (SDL), a language-neutral notation for describing types of all persistent data, and C++, used for writing application code. The current system runs to approximately 40,000 lines of code. Users of DIAS can interact with the system through either a Query Interface or a Tool Interface. The Query Interface is used for dynamically formulating queries while the Tool Interface is used for supporting existing (legacy) design tools that require a file system interface, rather than a database system interface. This is facilitated by the fact that Shore supports both database features and file system features. DIAS also provides a Programming Interface in the form of a library of routines that can be used to manipulate the database. These routines can be used for developing new design tools that directly use the database interface. For efficient query processing, a complete query processing system is implemented. Specialised access methods are incorporated for fast access of data. Issues related to object management and transaction processing are handled by the Shore server. The system overview of DIAS is as shown in Figure 1.4.

In summary, DIAS forms part of an information framework that extends chip design database systems that deal primarily with management of *design* data to include support for analysis of interconnect parasitics. DIAS is a working prototype and is being field-tested at Texas Instruments, Inc.

1.5 Earlier Work

The issue of data management for VLSI CAD has seen considerable research activity. The emphasis on making tools and data more accessible to designers has led to the *data oriented* design systems. Examples of VLSI CAD design database systems are

OCT [HMSN86], Cbase [Breu+88], 3DIS [AKMP85], etc. Also, *tool oriented* systems such as Cadweld [DaDi91] emphasizing tool integration have been developed.

Data representation and data management are major components of an electronic CAD *framework* [HNSB90, Gup+89]. Framework standards have been developed by EDIF [EDIF93] and CFI [CFI94]. These standards provide an information model and a programming interface. CAD databases such as DDB [SAL93, SPDL89] are based on the information model.

The systems considered above deal primarily with management of *design* data. In contrast, the focus of our work is on the complementary aspect of modeling and processing of *analysis* data. While the issues in design data management continue to be relevant since the basic entities are the same, other issues that are specific to analysis have to be dealt with. Our work involves a database system mainly focused on meeting the needs of data management of interconnect parasitics related data. To the best of our knowledge, this system represents the first database-related work in the area of interconnect analysis.

1.6 Organization

The rest of the thesis is organized as follows: In Chapter 2, we consider the requirements of our application and choose to adopt the object oriented data model for our database system. We briefly survey several object oriented database systems and choose a particular database engine to serve as the backend platform in Chapter 3. In Chapter 4, we provide the details of the backend database engine. Chapter 5 deals with several object oriented methodologies for software development and outlines the broad stages of the development of DIAS. We then discuss the object model that is developed for representing interconnect parasitics data in Chapter 6. The DIAS system architecture, the design and optimization stages are described in Chapter 7. We focus on the implementation of the DIAS kernel in Chapter 8. The interfaces that DIAS provides form the topic of Chapter 9. The Query Processing System is covered in Chapter 10. Chapter 11 describes the results of some initial tests to study the performance of DIAS. Finally, in Chapter 12, we present the conclusions of our study and identify future research avenues.

Chapter 2

Database Modeling Approaches

The domain of an application consists of a number of entities and the interrelationships among them. Databases are used to store information regarding these entities. The information is represented by means of the *data* and the *schema*. The data constitutes the bulk of the database while the schema is metadata describing the structure of the data. The data model of a database system defines the type system for specifying data and the operations on the data and the relationships among them. A Database Management System uses a particular data model for its implementation. Popular data models are the network model [BaWi64], the hierarchical model [McGee77], the relational model [Codd70], and recently, the object oriented data model. Frequently, DBMSs provide query languages for manipulating the data.

We now discuss some of the main requirements of a database system for VLSI CAD, specifically for managing interconnect parasitics data. Then, we discuss some of the disadvantages of using conventional database systems for this application. We then consider the main features of the object oriented data model which make it suitable for our application.

2.1 Application Requirements

We now consider the requirements of a VLSI CAD application. The subsequent design of the system will have to target these requirements.

1. Complex objects

   Applications such as VLSI CAD, multimedia and AI require a database processing model that is different from conventional ones such as banking, inventory control, etc. The idea is to have a processing model that supports the actual processing "context" or "working set" of the application rather than individual data items. Such a context defines the amount of data needed to perform a specific application task. VLSI design is usually done one step at a time using different tools: a synthesis tool is used to transform the logic description of a chip into chip structure data, which is then used by the planning tool to design the floorplan that, in turn, is fed to the chip assembly tool in order to derive the mask layout that is needed for chip manufacturing. The process model adopted by each tool is characterised by first reading its input data, then working in this context, and finally feeding the output data as the input to the subsequent tool.

   For example, the context for the chip planning tool describes the structure of the cell which is being designed. It consists of components defining the sub-cells together with their area estimations, and components giving the net data as well as the pin data that define the connections between the sub-cells relating the sub-cell components through pins on the same net. Thus, we can characterise a context as a composite or complex object consisting of several components (possibly of different types) with relationships in between. The provision of such a context requires efficient delivery of that component and the relationship data from the underlying DBMS. To satisfy this requirement, the DBMS should support direct representation of complex objects.

The interconnect parasitics study results in incorporation of large amounts of data to these contexts. This data constitutes new subcomponents and additional relationships to the existing complex design object. The role of the database system thus becomes all the more important in that in addition to supporting design entities, it has to effectively deal with the vast and complex parasitics data.

2. Hierarchical nature of design

   Usually, a complex design is composed of several subdesigns which in turn may be made of smaller components and so on till a primitive element is reached. To support this in a direct manner, the system should be capable of supporting hierarchies of arbitrary levels of nesting.

3. Uniform Representation

   A circuit is often designed and analysed from several perspectives, such as netlist, layout, etc. Individual CAD applications frequently deal with data from a single viewpoint. This requires translation of data from one format to another to facilitate intertool communication.

4. Fast storage and access

   The capability to access and modify small sections of the vast data is an important functionality. Frequently, users would want to deal with logical subunits of a complex design for ease of manipulation and speed of processing. The database system should support fast access of relevant data through efficient query mechanisms.

5. Support for schema evolution

   The underlying structure of the data or the schema may be subject to changes over a period of time. For example, the format of characterization data may change with a new vendor. The database system should support easy changes to the schema.

6. Long transactions

   The duration of CAD transactions is usually much longer than those of other database applications. The conventional techniques for concurrency control and recovery are based on locking and logging respectively. A CAD transaction acquires a lock on a data item which is released after a long time during which no other transaction can access the data item. Also, if the transaction is not successfully committed, to preserve database consistency, the database system is restored to the state at the start of the transaction. This results in substantial undoing of past work. The solution requires adoption of non-conventional transaction models.

7. Support for legacy applications

   With the development of a new system, an immediate concern is the handling of existing data and the application programs that deal with them. For example, CAD simulation tools are file based with specific formats. Very often, either the applications have to be rewritten or the data left in their original files and new applications continue to access this data from files.

2.2 Drawbacks of traditional database systems

Most current database systems are based on the relational data model, where all data is represented in the form of tables. Though these systems have a proven track record for commercial applications, they fail to satisfy the needs of specialised scientific and engineering applications such as CAD. We next highlight some of these shortcomings which are especially relevant to our application domain.

- No support for complex objects

  Most objects that are dealt with in VLSI CAD applications are complex. The inherent support for a direct and natural representation is required for performance reasons. The inability to represent naturally leads to fragmentation of a single object into numerous records. Retrieving information about this complex object as a whole needs expensive "join" operations.

- Schema evolution difficult

  The underlying structure (schema) of the data changes over time. In relational databases, any changes to the structure of the data may involve making changes to a number of relational schemes as all logically related data may not be grouped together in one single table.

- No support for hierarchies

  Hierarchies are an inherent feature in the VLSI CAD design domain. These should be represented in a natural manner. However, the relational model does not allow for an easy representation of such data as the hierarchy has to be flattened down into a set of tables.

- Lack of efficient handling of specialised queries

  Though conventional systems have a well defined query language, they do not efficiently handle some queries that are specific to our VLSI CAD application. For example, consider a typical query on the signals of a design: *"Fetch all signals and their source terminals that suffer slew greater than 5 ns."* Answering this query efficiently requires access methods that take into account the hierarchical relationships among the data.

2.3 The Object Oriented data model

Recent research in the field of databases has resulted in the emergence of new approaches such as the object oriented data model, which overcome the limitations of earlier models. The object oriented data model is also particularly suited for our CAD application as it satisfies most of the requirements discussed earlier. We next briefly describe the basic concepts of the object oriented data model.

1. Object: The basic modeling primitive is the *object*. Each object has a unique identity, which does not change with time. The system assigns a unique Object Identifier (OID) to preserve this invariant property. The *state* of objects is defined by the values they carry for a set of *properties*. These properties may be either *attributes* of the object itself or *relationships* between the object and one or more

other objects. The behaviour of objects is defined by the set of *operations* (or *methods*) that can be executed on an object of that type.

2. Class: Objects can be categorized into *classes* or user-defined types. A class serves as an abstraction for grouping objects that share the same structure and behaviour and a common range of states called its *extent.* An object is an *instance* of a particular class.

3. Aggregation: Aggregation supports the construction of objects which are themselves composed of other objects. This essentially captures the PART-OF relationship between the constituent objects and the complex object. This powerful feature supports arbitrary levels of nesting in the form of a hierarchy, called the *aggregation hierarchy.* An object at one level is composed of objects of lower levels of the hierarchy.

4. Inheritance: The inheritance feature allows new classes to be *derived* [Alba83] from existing ones. The derived classes inherit the attributes and methods of the parent class. They may also refine the methods of the parent class and add new methods. Inheritance is used to denote the IS-A relationship between classes. The parent class is a generalization of the child classes and the child classes are specializations of the parent class. For example, a Car is a specialisation of the class Vehicle, while Vehicle is a generalisation over Car, Bus, etc. Inheritance gives rise to a class hierarchy.

5. Encapsulation: The state of an object as defined by its attributes can be "hidden" and made accessible only through the methods of its class. This is done by declaring the attributes to be "private" and the methods to be "public". This is called *encapsulation* [Ga+87], which is borrowed from the concept of abstract data types in programming languages. In this view, an object has an interface part and an implementation part. The database translation of this principle is that an object has both data and operations to perform on the object. Thus, encapsulation provides a form of logical data independence.

6. Overloading and Overriding: A method may have the same name but different implementations. This is called *overloading.* The redefinition of the operation for each type is called *overriding.* The actual method invoked may be decided at runtime based on the type of the object, called *late binding.* Late binding is a form of *polymorphism* [CaWe85], where different implementations are attached to the same name so that they can be applied on objects of different types.

2.4 Object Oriented Database Management Systems

The object oriented data model is the basis on which Object Oriented Database Management Systems (OODBMS) are built. In addition to implementing the data model, an OODBMS supports the basic features of database management systems such as transactions, concurrency control and recovery. We shall next look at some of the important features of OODBMSs which are particularly important from our application's point of view and provide relevant examples from our application.

- Modeling Power: Since an OODBMS uses an object oriented data model, the expressive power available in object oriented programming is readily available in OODBMSs. Hierarchies which occur naturally in the domain are capable of being mapped directly. An example of this in our application is the following: A Cell could be a Design Cell or a Library Cell and a Design Cell is composed of other Design Cells and so on. This is represented using a combination of aggregation and inheritance hierarchies.

- Persistent Programming: Most of the concepts such as classes, aggregation, inheritance, etc. discussed earlier are similar to the corresponding concepts in object oriented languages. Many OODBMS are based on extending object oriented programming languages to include support for *persistence*. Persistence is the property by which data survives beyond the execution of the program that created them.

- No "impedance mismatch": In relational database systems, the computational power of the query language is considerably lesser than that of a programming language. This poses a restriction on the processing of the data stored in the database. This is frequently referred to as the *impedance mismatch* problem [BlZd87].
  OODBMSs resolve this problem by providing a single *database language* that can be used for both programming and querying.

- Schema evolution: The changes to the schema of the database get mapped to the following:

  1. changes to the attributes, methods of a class
  2. changes to the inheritance graphs
  3. changes to the types themselves

  These changes are easily applied to the existing class definitions. However, the existing instances are converted to a new class by defining transformation methods for the new classes with the old instances as arguments and then creating instances of the new classes and finally deleting the old instances.

  In our application, the structure of the characterisation data changes over time. So, easy schema evolution helps in migration to newer formats.

2.5 Choice for DIAS

Considering the above features, it appears that the object oriented data model is a suitable choice for DIAS. The requirements discussed earlier are targeted in DIAS by providing a database system solution built on the object oriented data model that captures the modeling requirements such as complex objects, hierarchies, etc. It also supports for easy schema evolution. The use of a single data store in the form of an object oriented database system has the following advantages. It stores the data in a format independent manner, thereby saving the high runtime cost incurred in translation of data between different formats. All applications see a single consistent view of the data. Also, modifications to the data are now restricted to a single location.

Efficient manipulation of the stored data is handled by providing support for querying and fast retrieval. DIAS implements a transaction model in which a long transaction is broken down into a sequence of chained transactions. Legacy data is supported by providing an interface for existing applications to interact in a seamless manner with the new system. Applications can then be developed to incrementally move these data out of the files and into the database system as if they were new data.

Chapter 3

OODBMS Platforms

In this chapter, we briefly survey several Object Oriented Database Management Systems (OODBMS) that could serve as the platform on which to build the database system for storing VLSI interconnect parasitic data. We finally choose a database engine that is particularly suited for our application. We start with a description of the features that a system should possess in order to be called an object oriented database system. While Codd's original paper [Codd70] gave a clear specification of a relational data model, no common definition for "the" object oriented data model has as yet emerged. The reason might be the lack of a consensus on a formal specification of the object oriented data model. However, there is general agreement over what features are expected of an object oriented database system. We next consider some of the descriptions of object oriented database systems that are found in the literature.

The Object Oriented Database System Manifesto [Atk+89] is an early attempt to define an object oriented database system. It describes the features and characteristics of an object database system. It classifies the characteristics of an OODB system into three groups:

1. Mandatory Features: Mandatory features are those that a database system must provide to be called object oriented. These include the support for complex objects, object identity, encapsulation, types or classes, inheritance, overriding with dynamic binding, extensibility, computational completeness, persistence, secondary storage

management, concurrency, recovery, and ad-hoc query facility.

2. Optional Features: These features would be desirable but are not mandatory to qualify the system as an OODBMS. These are multiple inheritance, type checking and inference, distribution, long transactions and versions.

3. Open Features: These are points open to the designers' choice. These include the programming paradigm, the representation system, the type system and the degree of uniformity in the treatment of types, objects, and methods.

A data model that is similar to that described in the Object Oriented Database System Manifesto is proposed by [Kim90]. The requirements for an OODBMS that are mentioned include support for concepts of object and object identifier, attributes and methods, class, and class hierarchy, on top of the database features required by a DBMS in the traditional sense. This view is also shared by [ZdMa90]. They maintain that an OODBMS must first provide the basic features and functionality that are provided by conventional current database systems. These features are divided into essential and frequent features:

- Essential Features, such as a data model and language, representation of relationships among entities, permanence, sharing of data and capability of supporting databases of arbitrarily large sizes.

- Frequent Features, that include integrity constraints, authorization, query language and views.

After defining the above features, [ZdMa90] propose the *threshold* and the *reference* object oriented models. An object oriented database system, in addition to satisfying the essential features, must atleast satisfy the requirements of the threshold model: object identity, encapsulation, and complex objects. A powerful OODBMS may adopt the reference model, that adds the following features to the threshold model: typing of objects and variables, hierarchies, polymorphism, collections, namespaces, queries and indexes, relationships and versions.

The above specifications try to encompass almost all the features of an OODBMS. Object oriented database systems use different approaches to provide these features. We shall next attempt to survey representative systems based on each of these approaches.

Based on the approaches to object data management, a broad classification is as below:

- Object Oriented database programming languages: Object oriented database programming languages add database capabilities to existing object oriented programming languages. The database system augments the programming language by providing persistence, concurrency control, a query language and other DBMS capabilities. Examples are ODE [AgGe89], ObjectStore [Lamb+90], GemStone [Bret+88], O_2 [Deux+90], and Ontos [AHKD89].

- Extended Relational Database Systems: Extended Relational Database Systems augment the relational model with the ability to define new data types, invoke programming language procedures from the query language, and to write procedures in the query language itself. Other features that are incorporated are a hierarchy of relation types with inheritance of attributes, query language extensions for transitive closure, and rules. POSTGRES [RoSt90], Illustra [IlUG95], and Starburst [Sch+90] are examples of such systems.

- Database System Generators: These are systems allowing the user to build an OODBMS tailored to particular needs, typically with a custom data model and a database language. There are two approaches here: the *toolkit* approach and the *constructor* approach. In the toolkit approach, a powerful language is provided for writing a DBMS together with prewritten packages for the application independent capabilities such as the query parser and optimizer and a storage manager. In the constructor approach, the database system generator provides a formalization of database architecture, allowing a DBMS to be generated by linking together the existing modules in different ways. EXODUS [Car+90] follows the toolkit approach while GENESIS [Bat+90] follows the constructor approach.

- Object Managers: Object Managers usually have less functionality than the ones considered so far. They minimally provide a persistent object store with concurrency control but do not have a query or programming language. Examples are Mneme [MoSi88] and POMS [Coc+89].

We shall now study briefly some of the above systems, concentrating mainly on the following features: persistence, programming interface, query language, storage management and transaction management and other options such as constraint enforcement, triggers, versioning, etc.

3.1 Object Oriented database programming languages

3.1.1 ODE

ODE (Object Database and Environment) [AgGe89] offers an integrated data model for both database manipulation and general purpose programming. The database programming language is O++, which is based on C++. O++ borrows and extends the class-based object definition facility of C++ by providing persistence. Other facilities include versions, sets and clusters of persistent objects and iterators for these objects. It also has provisions for associating constraints and triggers with objects.

O++ and the data model

In addition to the programming capabilities provided by C++, persistent objects can be declared using the keyword *persistent*. Persistent storage operators, *pnew* and *pdelete*, are used instead of the C++ operators, *new* and *delete*, for creation and destruction of persistent objects respectively. Then, these objects may be referenced using pointers to persistent objects similar to transient objects. ODE also introduces the notion of *dual pointers* which can refer to both transient and persistent objects. All persistent objects of the same type are grouped together in a *cluster*, the name of the cluster is the same as that of the corresponding type; that is, clusters are type extents.

Query Processing

By providing clusters, sets and iterators, O++ provides an alternative to using persistent pointers to navigate through the database to answer a query. Also, arbitrary "joins" can be performed by iterating over the clusters to be joined with the join predicate expressed as the iteration condition.

Versioning

The versioning principles employed in ODE ensure that all persistent types can have any number of versions and that at any point, the current version or a specific version can be accessed. A pointer to a persistent object can access the current version, in which case it is called a *logical id*. In case the pointer refers to a specific version, it is called a *version pointer*. The macro *previous* when called with a persistent pointer and a non-negative integer i returns the *ith* version.

Triggers

Triggers monitor the database for specific conditions, and when the condition becomes true, the associated trigger action is executed. Triggers are specified within class definitions and are of two types: *once-only* and *perpetual*. An once-only trigger is automatically deactivated after the trigger has been "fired", and then must be reactivated explicitly if desired. On the other hand, a perpetual trigger is automatically reactivated after being fired. A trigger T_i is associated with an object whose id is *object-id*. If the trigger condition is true, T_i is activated by the call, $object - id \rightarrow T_i(arguments)$ which executes the trigger body of T_i in a separate transaction.

3.1.2 ObjectStore

ObjectStore [Lamb+90] is an OODBMS that provides a tightly integrated language interface to the traditional features of persistent storage and transaction management. It supports a client server model of computing, with user applications being clients and

ObjectStore servers providing the functionality of distributed databases. Applications access ObjectStore through DML (Data Manipulation Language) and library interfaces to C++ and C. The DML interface uses extensions to the C++ language, while the library interface works through a library of C++ functions.

Querying

In the DML interface, queries are a new kind of language expression. A query expression consists of a collection object (such as a list or a set) and a C++ Boolean-valued subexpression. The value of the query expression is the subset of elements of the collection for which the subexpression is true, with the value of the C++ *this* variable being set to each element. Queries are also supported by the library interface. The query expression is passed as a string to query construction, binding and evaluation functions. Index maintenance and optimization-strategy formulation are maintained automatically by the system; the programmer specifies which data members should have indexes, and whether they are B-tree or hash-table indexes.

Persistence

Object persistence is specified by a parameter to the C++ *new* operator, specifying in which database the object should reside, and optionally specifying clustering information. An interesting feature of ObjectStore is its implementation of object identifiers (OIDs) through the use of direct virtual memory pointers. Whenever the target of a reference is not in the client cache, the virtual memory system signals a page fault, which the operating system reflects to ObjectStore. Subsequent accesses to the data are as fast as they would be for ordinary C++ objects. It reverts to longer, structured OIDs for special purposes such as conveying OIDs between processes or to external ObjectStore repositories.

Version Control

Configurations of objects can be checked out into private "workspaces", modified, and then checked back into shared workspaces when the changes are complete. When a configuration is checked out, other users can be forbidden from checking out the same configuration, or they can be allowed to check out an alternative version. Thus, the concurrency control can be either pessimistic or optimistic. If two users change the same data, rather than forcing a long transaction to abort, ObjectStore allows the users to operate simultaneously on parallel versions of the data, and later to merge their changes. The version control mechanism can be used for simply recording history, as well as for long transactions.

The overall distributed database ObjectStore architecture supports a client/server model of computing, with user applications being clients and ObjectStore servers providing the functionality of distributed databases. The process architecture broadly consists of the ObjectStore Server, that manages data pages on behalf of client applications, the ObjectStore Directory Manager, that maps hierarchical database names to servers, and the ObjectStore Cache manager, responsible for optimization of page-lock management on behalf of client applications.

3.1.3 GemStone

GemStone [Bret+88, MaSt90] merges object oriented language concepts with those of database systems, and provides an object oriented database language called OPAL, based on Smalltalk. It has subsequently been integrated with C++; it is one of the first database programming language products having close integration with two languages.

The GemStone Data Model

The data model and the programming language, OPAL are based on the Smalltalk concepts of *object, message,* and *class*. Objects communicate with one another by passing *messages*, which are requests for the receiving object to change its state or return a result. An object responds to a message by a *method*, an OPAL procedure that is invoked when an object receives a particular message.

The GemStone Architecture

The major components of the Gemstone system are the two processes, *Gem* and *Stone.* The Stone process is the data manager, providing disk I/O, concurrency control and other transaction services. The Stone process typically resides on the server machine, accessing the disk through the operating system calls. The Gem process provides compilation of OPAL programs, user authorization and a predefined set of OPAL classes and methods for use by the user program. Another process, the PIM (Procedural Interface Module) is a set of routines to facilitate communication from programs written in other languages. For example, it supports calls from C programs to execute a sequence of OPAL statements.

Concurrency control is ensured using either optimistic or pessimistic methods. The pessimistic scheme uses traditional locking implementation. The optimistic scheme uses shadow paging.

GemStone interfaces

It supports multiple languages and tools. It provides its own implementation of a Smalltalk based query language called OPAL. It also supports mapping between C++ object data structures and OPAL data structures stored in the database, and transparently fetches and translates objects when a C++ application references them. Also, data may be fetched and stored from GemStone using procedure calls from application programs written in C or Pascal. This is done through the PIM process using procedure calls. The GemStone Object Development Environment (GEODE) provides a visual programming environment for the development of interactive object oriented software based on ObjectStore.

3.1.4 O_2

The O_2 [Deux+90] system, is a research prototype that has been transformed into a commercial product. O_2 provides object identity, encapsulation, multiple inheritance, late binding, lists, sets, and a powerful declarative query language.

System Overview

O_2 provides a complete application environment consisting of the database programming languages, CO_2 and BasicO_2, a set of user interface generation tools, LOOKS, and a programming environment, OOPE. O_2 consists of several other additional functional modules: an alphanumeric interface, the language processor, the query interpreter, the schema manager, the object manager and the disk manager.

The programming functionalities of O_2 can be realised through CO_2 and BasicO_2, which are object oriented extensions of C and Basic respectively. Procedures are stored in the O_2 database, rather than in the programming language environment. Procedures are dynamically loaded and linked at run time. This approach is in contrast to most commercial OODBMS products, which use binary files which are part of the application. This approach allows optimization of the executable code by replacing dynamic name solving with function calls where possible. However, this precludes modification to the schema. O_2 remedies this by providing two modes of application development, the initial *development* mode, when changes to the schema are allowed, and the *execution* mode, when the schema is frozen and the code optimized.

O_2 has a powerful declarative query language. The query language can be used to access attributes and execute methods when used for ad hoc queries and is restricted to operate only on methods to enforce encapsulation constraints when used as part of the programming language. It provides *filters* to select data using the SQL *select-from-where* clause and includes a complete set of operations to construct and decompose lists, sets, and tuples. The result of a query can be an object of an existing type in the database or values. Values may be simple or list, set or tuple of simple values. Apart from these, a program can also access data one object at a time in navigational fashion also.

Persistence Features

O_2 is implemented in C on top of the Wisconsin Storage Manager [CDKK85]. Structured OIDs are used, and a reference count is maintained for each object. Persistent objects are deleted automatically if they can no longer be reached from a persistent, global named object. Lists and sets are implemented as ordered trees.

Transactions

Transactions are implemented using an optimistic scheme. Pages are read-locked when they are read from the server, but pages containing modified objects are not write-locked until right before transaction commit. If the write-lock fails, the transaction is aborted.

3.1.5 ONTOS

ONTOS [AHKD89], provides persistent storage capabilities to data manipulated using C++ programs. It is integrated with the language through the introduction of new persistent classes all defined as subclasses of the base class *Entity*. Methods can be invoked through conventional compiled C++ calls. ONTOS also provides facilities to invoke methods by interpretation of named strings at run time.

ONTOS objects exist in two states, *deactivated* when stored on disk, and *activated*, when appearing as ordinary C++ objects in an application program. Objects are activated automatically when an application program first attempts to fetch them and are deactivated and written back to disk if changed at the end of a transaction.

Concurrency control is provided by conventional locking at the different granularities of individual objects, pages or entire segments. It provides caching of objects on the client workstation in its client-server architecture. It employs a variant of SQL as a query language, making attributes of the objects visible to the query language user, though it violates the encapsulation semantics of C++. Other features include automatic maintenance of inverse relationships, versioning and a *make* tool to keep application programs synchronized with schema changes.

The programming capability is embedded inside the query language. The query language may be extended to provide control constructs with the capability of a programming language, which are used to define actions to be taken when a field is fetched or modified.

Other ONTOS features include the support for multiple versions of an object and *Binary Large Objects (BLOB)*, which are too large to fit in a single page. These BLOBS are stored using a B-tree representation, similar to that used in the EXODUS system described later.

3.2 Extended relational database systems

3.2.1 POSTGRES

POSTGRES [RoSt90, StRo86], extends the conventional relational data model to offer services in two dimensions: *object management* and *knowledge management*. Object management ensures direct representation and manipulation of data belonging to non-traditional data types, while knowledge management entails storing and enforcing *rules* that are part of the semantics of an application. Rules help describe the integrity constraints of an application and also allow the derivation of data that is not directly stored in the database.

The POSTGRES Data Model

The stated goal of POSTGRES is to retain the simplicity of the relational model. Consequently, the data model adds the following constructs to the relational data model: classes, inheritance, types and functions.

The fundamental notion in POSTGRES is that of a class, which is a named collection of instances of objects. Each instance has the same collection of *attributes* and each attribute is of a specific *type*. Each instance has a unique identifier (OID). A new class can be created by specifying the class name, along with all attribute names and their types. A class can optionally *inherit* data elements from other classes.

POSTGRES contains an expressive type system and a powerful notion of functions. There are three kinds of types: base types, array of base types and composite types. It contains an *abstract data type (ADT)* facility whereby any user can construct new base types. Functions need to be specified to convert to these types from the usual base types integer, float and character strings. Arrays of these types are supported just as for base types. Composite types allow an application designer to construct *complex objects*, that is attributes which contain other instances as part or all of their value.

There are three different kinds of functions in POSTGRES: functions written in C, operators and POSTQUEL functions. A C function can be defined whose arguments are base types or composite types. This function can be called with the argument as a class name or as a new *attribute*, whose type is the return type of the function. POSTGRES supports the *operator* functions for using indexes in processing queries. Operators are functions with one or two operands and use the standard operator notation in the query language. The operator is defined by indicating the token to use in the query language as well as the function to call to evaluate the operator. The third kind of function is the POSTQUEL function, where a collection of commands in the POSTQUEL query language are packaged together and defined as a single function.

The POSTGRES query language

The query language, POSTQUEL, is a set-oriented query language that is a superset of a relational query language. The features added to a relational query language are support for nested queries, transitive closure, support for inheritance and *time travel*. Operators can also have sets of instances as their operands. The transitive closure operation allows one to explode an aggregation or inheritance hierarchy by specifying a * after the class name. The query is run until the answer fails to grow. Time travel allows the user to run historical queries such as "Find the salary of Sam at time T".

Rules

Rules are used to support all of the following functions: view management, triggers, integrity constraints, referential integrity, protection and version control. The rule consists of commands to be executed when an event such as retrieve, delete, append, replace takes place on an object.

Storage System

The storage manager employs a mechanism of "no-overwrite", where the old record remains in place whenever an update occurs and serves the purpose of a write-ahead log. Hence, the log in POSTGRES just consists of two bits per transaction indicating whether the transaction has committed, aborted or is active. As a result, crash recovery is instantaneous as no updates are undone. The previous records are readily available. The second benefit of this approach is the notion of time travel. Time travel is supported by maintaining two different physical collections of records, one for current data and another for historical data. However, these mechanisms result in the accumulation of a large amount of data. Users, who do not require historical data have to periodically run the *vacuum cleaner*, that moves the historical records from the disk containing current records to archival storage.

3.2.2 Illustra

Illustra [IlUG95] is a commercial *Object-Relational* DBMS based on the POSTGRES database system. It supports object oriented features, provides a declarative query language (based on SQL), and has a comprehensive set of library routines that can be used for developing applications.

The client-server architecture of Illustra consists of several processes. The *midaemon* is the main server process. An instance of the *miserver* process is spawned for every client process. The client processes can be the query interface or an application using the programming interface.

The object oriented features of Illustra include the support for complex objects which map to tables of a relational database. Composite attributes such as arrays, sets are stored in separate tables automatically created by Illustra. Illustra allows users to create functions written using SQL or C. These functions can be used to build iterators for traversing set-valued attributes.

For increased concurrency, Illustra supports different levels of isolation: *read uncommitted, read committed, repeatable read*, and *serializable*. Application programs can monitor the occurrence of asynchronous events, using *alerters*. Illustra provides an event-driven rules' system. A rule can be specified for an attribute of a class or the entire class. When a database event occurs, and particular conditions based on the rules are satisfied, it takes specific actions. Users can request data current at a particular point in time or between a time interval. This is similar to the concept of *time travel* supported in POSTGRES.

3.2.3 STARBURST

Another example of an extended relational database system is Starburst [Sch+90]. In addition to providing built-in extensions to the relational model, it allows the advanced user to augment the functionality of the DBMS.

The Starburst query language extends the relational algebra, and supports pre-defined extensions to query analysis, optimization, execution, and access methods. The user may also define new literal data types with associated procedures that encapsulate type semantics. It also supports recursive queries and structured-result queries which are similar to POSTGRES. It also provides the capability for triggers.

There are two different type systems and languages, one for the DBMS and one for the application program. Starburst parses queries into a semantic net representation called the Query Graph Model (QGM). The query analysis phase is driven by rewrite rules on the QGM, traversing and translating the query graph into equivalent, more optimal representations. The query optimizer translates the QGM representation into an execution plan of LOw LEvel Plan OPerators, or LOLEPOPs. Examples of LOLEPOPs include access method scans, joins and sorts. The sophisticated user can extend the Starburst query language with *table functions*, which are pseudotables that are not stored in the database but are computed on the fly.

3.3 Database System Generators

The approaches to object data management discussed so far have been designed to provide extensibility in the implementation and definition of types and operations provided by the DBMS. However, database system generators, such as EXODUS and GENESIS, go further in this regard allowing for extensions in the data model.

There is no single data model for a database system generator. The data model is defined by a special user called the *database implementor* (DBI). Both the logical and the physical data model need to be described. The logical data model specifies the query and representation capabilities of the system such as keys, relationships and query syntax. The physical data model specifies the access methods and the ways in which logical data schemas can be mapped onto them.

There are two variations of database system generators, according to the architecture chosen:

1. Toolkit approach: In the database toolkit approach, the database system generator provides a powerful language for writing a DBMS together with prewritten packages for the most commonly used capabilities required in a DBMS, such as a B-tree package, query optimizer and storage manager.

2. Constructors: In the database constructor approach, the database system generator provides a formalization of database architecture, allowing a DBMS to be generated by linking together existing modules in different ways and using tools and high-level languages to generate DBMS modules automatically.

We shall consider three systems which are database system generators. Of these, EXODUS and SHORE are examples of the database toolkit approach while GENESIS is an example of the constructor approach.

3.3.1 EXODUS

In the EXODUS toolkit [Car+90], the major facilities provided to aid the DBE (Database Engineer) in the task of generating an application specific database system are:

- The Storage Manager
- The E programming language compiler
- A library of type independent access and operator methods
- A rule-based Query Optimizer generator
- Tools for constructing query language front-ends

A DBMS generated by EXODUS consists of a parser and query optimizer, both generated automatically from a description of the target query language and operators, the E compiler, required because the output of the query optimizer is an E program, a catalog (data-schema) manager, partially or wholly written by the database implementor in E, and the database engine itself. The database engine consists of three layers: the operator methods (implementation of language operations), access methods and the storage manager. The database implementor may choose the operator and access methods from the libraries provided by E, or may implement his own using E. The storage manager can also be modified by the database implementor, but is normally left unchanged.

The basic representation of data in the storage manager is a variable length uninterpreted sequence of bytes. In the simplest case, these storage objects are implemented as a contiguous sequence of bytes. As the objects grow, they are represented using a B-tree of leaf blocks, each containing a portion of the sequence. Objects are referenced using structured OIDs. On these basic storage objects, the storage manager performs buffer management, concurrency control, recovery and a versioning mechanism that can be used to provide a variety of application-specific versioning schemes. Transactions are implemented using a shadowing and logging technique. The database implementor's language E is a superset of C++, providing generic classes, iterators and support for persistent object types. References to persistent objects look much like those for ordinary C++ objects. The E compiler generates calls to the storage manager for buffer management. As a trial run, the EXODUS system was used to develop an object oriented DBMS, consisting of the data model, EXTRA, and the query language, EXCESS. EXTRA, includes support for persistent objects, type inheritance, automatic maintenance of inverse attributes for relationships and bulk types such as lists, sets and arrays. EXCESS, is designed to provide a uniform query interface to sets, arrays and tuples and individual objects, all of which can be composed and nested. Subsequently, more than 350 groups have used EXODUS and it forms the storage manager for MediaDB, a commercial multimedia DBMS.

3.3.2 SHORE

SHORE (Scalable Heterogeneous Object REpository) [Zwi+94] is a recent persistent system developed at the University of Wisconsin, Madison. It is the successor to EXODUS, and is the result of a merger of file system and object database technologies.

Object Oriented Database Features

SHORE (or Shore) is intended to allow databases built by an application written in one language to then be accessed and manipulated by applications written in other object oriented languages as well. The SHORE Data Language, SDL, is the language in which SHORE types are defined. The methods associated with the types written earlier in SDL can be written using any of the languages for which a SHORE language binding exists. Apart from these features, SHORE provides regular database system services such as concurrency control and recovery. It also provides for clustering and lower levels of transaction consistency.

File System Features

SHORE provides a tree-structured UNIX-like namespace in which all persistent objects are reachable from a distinguished root directory. Each Shore object may optionally designate a range of bytes as its *Text* field. A file then becomes a Shore object with only one field, the *Text* field. Existing applications can treat these objects as if they were files.

For applications that do not tolerate even minimal changes, an NFS file server is available. An entire subtree of the SHORE name space can be mounted on an existing UNIX file system. When applications attempt to access files in this portion of the name space, the UNIX kernel generates NFS protocol requests that are handled by the NFS value added server.

We present the features of Shore in greater detail in Chapter 4.

3.3.3 GENESIS

The GENESIS database system generator [Bat+90] has concentrated on the development of an encompassing and practical theory of DBMS implementation. The theory provides a framework in which the basic components of DBMS software are modules that realize simple files (file structures), linksets (record linking structures), and elementary transformations (conceptual-to-internal mappings). The storage architectures of commercial DBMSs are explained by compositions of these building blocks.

GENESIS is based on providing high level tools for constructing a DBMS from a library of existing modules. Once the storage architecture for the new system is designed, only the modules that are not present must be written.

Files and records are fundamental concepts in databases. A *file* is a set of records that are instances of a single record type. A relationship between two or more files is a *link*. Files and links are logical concepts; their implementation is still unspecified. To explain their implementations, two models, the *Transformation Model* and the *Unifying Model* are used. The TM formalizes the notion of conceptual to internal mappings and the UM codifies file structures and record linking mechanisms.

Overview

The Database Implementor (DBI) is responsible for designing a storage architecture for the target DBMS. The DBI specifies this architecture by writing an *architecture program*, which compiles conceptual schemas using the DDL *compiler* and maps the data definitions of conceptual files and links to their internal counterparts. The mappings are accomplished by prewritten procedures called *transformers* which realize the abstract-to-concrete data definition mappings of elementary transformations. Different storage architectures are realized by different architecture programs. The *Database Administrator* (DBA) is responsible for database design. The DBA develops conceptual schemas in terms of the GENESIS data definition language and runs the architecture program to convert these schemas to an internal representation called *storage architecture* tables. Database users write transactions to process database retrievals and updates. The host language is C. The record types that can be defined in GENESIS are more general than those supported in C, with routines to read and manipulate buffer-resident GENESIS records. These are the routines of the *trace manager*. Records are transferred between main memory buffers and secondary storage by file operations which are accessed via the *Grand Central* module. Modules called *expanders* define the abstract to concrete mapping for each transformation. Abstract operations are eventually mapped to operations on internal files, which are processed by Jupiter, the file management system of GENESIS.

A key feature of the GENESIS work is the formalization of database architecture on which it is based. The formalization is made possible by "standardization" and "layering". Layering refers to dividing an architecture into interacting portions that implement mappings between successive levels of data representation. It must be possible to design and implement the different layers independently. Standardization refers to finding a "simplest common interface" between layers that permits many different implementations of each layer's functionality. For example, a common interface may be designed that allows different alternatives for transaction recovery such as shadowing, logging etc. GENESIS allows a mixture of record structuring capabilities, including fixed-length and variable length fields, scalar and set-valued attributes, nested records and arrays. It also incorporates transactions and a variety of physical access methods including several variations of indexing. A preconstructed library of the common functions required in a DBMS is assembled by GENESIS's graphical database implementor tool known as the Database Type Editor, DaTE.

3.4 Object Managers

The last category of OODBMS we consider is *object managers.* These systems generally have less functionality as compared to the others we have discussed so far. They minimally provide a persistent object store with concurrency control and they generally do not provide a query language. An object manager typically provides a simple data model, sometimes patterned after a programming language for which it might serve as an object store. They also provide concurrency control and disk management capabilities. We shall consider two examples of object managers, Mneme and POMS. The first was built to serve as a storage manager for any application language or DBMS; the second was designed for the persistent language PS-ALGOL.

3.4.1 Mneme

Mneme [MoSi88], developed at the University of Massachusetts, provides object identifiers, persistence, concurrency control and recovery. The client interface to Mneme is a set of procedures. The goal of the system is to achieve very high performance with this minimal functionality. The Mneme data store is simply a heap of untyped objects. It is not aware of the internal structure of objects, except for attributes containing OIDs. An object consists of an OID, an array of data bytes and an array of references. Objects are grouped into *pools* that can have different storage management policies. The policy (for example, for caching and clustering) is defined by the database administrator writing a *policy module*. One or more pools are stored in one operating system *file*. OIDs occur in two forms, called client IDs (CIDs) and persistent IDs (PIDs). PIDs are the form actually stored in reference arrays, and are valid only within the file where they are stored. To reference an object outside the file, an artificial *forwarding* object is created within the file, which identifies the file and remote PID in that file. Thus, local references are short and fast, but remote references require a level of indirection. CIDS are valid only for the duration of a database session, but they may reference an object in any open file.

Mneme, thus, is a good example of the object manager approach. It provides the minimum requirements of an object store with the thrust being on high performance.

3.4.2 POMS

The Persistent Object Management System [Coc+89], or POMS built for the persistent Algol project at the University of Edinburgh in UK, is an example of an object manager motivated by a particular programming language. It provides persistent objects, object identifiers, transactions for concurrency control and recovery, translation of persistent Algol objects to and from a disk representation, and general storage management and allocation. The kernel of the POMS system is a process for translating OIDs to memory addresses when a reference is followed to an object not in memory, and back again when an object is flushed to disk. This is achieved with a two way hash table that allows lookups by OIDs or by memory addresses. Transactions are implemented using shadow pages, using a rewrite of the logical-to-physical page map as a means of atomicity. Special optimizations are made in the use of storage allocation on disk. An effort was made to store sequential elements of lists in the same disk page. Record types (structures) are also grouped on disk pages according to type.

In POMS, both program object code and databases carry detailed descriptions of the classes declared or stored within them. This is done by tagging a *header* in the front of each object. A POMS database contains a named collection of PS-algol objects: strings, structures and vectors. At an implementation level, a database effects mappings between disk block based address space to a high level language object address space and between relative disk block numbers to physical disk block numbers so that a set of non-contiguous disk blocks may be made to look like a single contiguous address space.

In summary, all the systems we have considered are representative object oriented database systems. These provide additional features compared to relational database systems and are especially suited to specific application domains such as CAD, CASE, AI, office information systems, etc. where relational database systems are inadequate.

3.5 Choice for DIAS

The commercial object oriented database systems, though robust, do not offer much scope for customization for a specific domain. For example, they usually do not allow for easy incorporation of new access methods. The extended relational systems are suitable for adding additional functionality to existing applications developed on relational systems. They do not completely overcome the drawbacks of relational systems as the data is finally stored in the form of relations. The object managers do not provide all the features of an OODBMS. In contrast to all these systems, database system generators are particularly attractive as they allow the development of a customized system tuned to the needs of a specific domain. Of these, EXODUS is a robust, powerful system that has been tried and tested the world over and also forms the storage mechanism for some commercial DBMSs. The SHORE system is the successor to EXODUS, and continues to support the features of extensibility using the toolkit approach. Moreover, SHORE provides file system features in addition to database system features. This is specifically useful in the VLSI CAD parasitic domain since a number of legacy file based tools exist that interact with the stored data. In view of all these features, we have selected SHORE as the backend database platform on which to build our DBMS for management of VLSI CAD parasitic analysis information.

Chapter 4

SHORE

The SHORE (Scalable Heterogeneous Object REpository) system represents a merger of file system and object oriented database technologies. In this chapter, we describe features of the SHORE (or Shore) system, which forms the backend database server of the database system for storing VLSI CAD analysis data and parasitics related information.

4.1 Introduction

While the past few years have seen significant progress in the OO databases area, most applications have not chosen to leave file systems behind in favour of OODBMSs. Some of the reasons are:

1. Many current OODBMSs are restricted to a single language while large scale applications often require multilingual data access.

2. With most current OODBMSs, application programmers face an either/or decision - either they put their data in the OODBMS, in which case all their file based applications must be rewritten, or they leave their data unchanged in files.

3. Most current OODBMSs have strong client server architectures, and are thus inappropriate for execution in peer-to-peer distributed systems.

The Shore system addresses these issues. The Shore Data Language (SDL) provides a single language neutral notation for describing the types of all persistent data. Shore has a symmetric, peer-to-peer structure. The design is scalable; it can be run on a single processor, a network of workstations or a large parallel processor. It also provides a Unix-compatible interface for legacy software tools. Apart from these features, it supports the traditional database services such as associative data access, indexing and clustering.

4.2 Basic Shore Concepts

Shore is a collection of cooperating data servers, with each data server containing typed persistent data objects. To organize this, a Unix-like namespace is provided. As in Unix, named objects can be directories, symbolic links or individual typed objects (the counterpart of Unix files). Shore allows each object to be accessed by a globally unique Object Identifier (OID). The OID consists of an 8-byte volume identifier and an 8-byte serial number. The former is designed to be large enough to be globally unique, while the latter is large enough to avoid reuse of values under any operating conditions. Shore also introduces a few new types of object classes, including *types* and *pools*. The type system is language neutral, supporting applications in any programming language for which a language binding exists. For objects whose primary data content is textual or untyped binary data, Unix file system calls are provided to enable legacy applications to access their data content in an untyped manner.

4.3 Shore Object Basics

The Shore object model consists of *objects* and *values*. Every persistent datum is an object and an object is a container for a value. Every object has a type, and all Shore objects are tagged with a reference to a type object that captures this information.

The *core* of an object is described by its type. To support the flexibility of dynamic structures with the efficiency of logically contiguous blocks on secondary storage, Shore allows each object to be extended with a variable-sized *heap*. The heap is used by the system to store variable-sized components of its value such as strings, variable arrays and sets.

4.4 File System Features

From a file system standpoint, Shore provides two major services: First, to support object naming and space management in a world with many persistent objects, Shore provides a flexible object namespace. Second, to enable legacy Unix file based applications to continue to exist while new Shore applications are being developed, mechanisms are provided that permit Shore object data to be accessed via Unix file system calls. However, this method of access does not guarantee the database system features of concurrency control and recovery.

4.4.1 Shore Object namespace

Shore provides a tree-structured, Unix like namespace in which all persistent objects are reachable either directly or indirectly from a distinguished root directory. The namespace extends the set of familiar Unix object types (directory, symbolic links, regular files) with *cross references, pools, modules* and *type objects*. Directories and the objects they contain are called *registered* objects. Each registered object contains a superset of the Unix attributes: ownership, access permissions, and timestamps. Shore also introduces a new kind of registered object called a *pool*. Typically, the contents of one pool form a single database. Members of a pool, called *anonymous objects*, are clustered near one another. Anonymous objects do not have path names, but they can be accessed by OID like any other object. The *registered* property is orthogonal to type: any kind of object can be created either in a pool (as an anonymous object) or in a directory (as a registered object). In a typical SHORE database, the vast majority of objects will be anonymous, with a few registered objects serving as entry points to graphs of anonymous objects. By introducing these object types, Shore gives users a framework in which to register both individual persistent objects as well as the roots of large persistent data structures. Thus, Shore provides a much richer naming environment than the single-level "persistent root" directory found in current OODBMSs.

Shore introduces three more fundamental kinds of objects, *modules*, *type objects* and *cross references*. Modules and type objects are similar to pools and anonymous objects, respectively, but have different deletion semantics to preserve the existence dependency from objects to their types. Cross references are similar to symbolic links in that they provide a way to insert an alias for an object into the directory name space. While a symbolic link contains a path name for a registered object, a cross reference contains the OID of an arbitrary object.

4.4.2 Legacy Unix tool support

While Shore provides a richer environment than traditional file systems, there are many situations where tools designed to be used on files need to be invoked on database objects. The solution followed in Shore is to provide a special *Unix compatibility* feature. The Unix compatibility feature may be utilised by designating a range of bytes as its *text* field. A compatibility library provides versions of Unix file system calls.

For applications that cannot even be relinked, Shore provides an NFS file server. An entire subtree of the Shore namespace can be "mounted" on an existing Unix file system. When applications attempt to access files in this portion of the name space, the Unix kernel generates NFS protocol requests that are handled by the Shore NFS value added server.

4.5 Object Oriented Database System features

4.5.1 The Shore Type System

The Shore type system is embodied by the Shore Data Language, SDL, the language in which Shore types are defined. SDL is quite similar to the Object Definition Language (ODL) proposal [Catt94] from the ODMG consortium, which is descended from OMG's

Interface Description Language (IDL), a dialect of the RPC interface language used in OSF's Distributed Computing Environment.

All objects are instances of *interface types*, types constructed with the *interface* type constructor. Interface types can have methods, attributes and relationships. The attributes of an interface type can be one of the primitive types (e.g., integer, character, real) or they can be of constructed types. Shore provides the usual set of type constructors: enumerations, structures, arrays, and references (which are used to define relationships). In addition, Shore provides a variety of *bulk types*, including sets, lists and sequences, that enable a Shore object to contain a collection of references to other objects. Finally, Shore provides the notion of *modules*, to enable related types to be grouped together for name scoping and type management purposes.

4.5.2 Shore Language Bindings

Shore is intended to allow databases built by an application written in one language (e.g., C++) to then be accessed and manipulated by applications written in other object-oriented languages as well (e.g., CLOS). This capability will be important for large scale applications such as VLSI CAD; C++ might be used for efficiency in simulating large chips, while CLOS (or perhaps Smalltalk) might be more convenient for writing the associated design rule checking and user interface code. In Shore, the methods associated with SDL interfaces can therefore be written using any of the languages for which a Shore language binding exists. Currently, only the C++ binding is operational.

4.5.3 Other OODB-Like services

Shore provides support for concurrency control via locking and crash recovery through logging. Shore provides users also with a choice of lower levels of consistency and recovery.

4.6 The Shore Architecture

Shore executes as a group of communicating processes called Shore servers. Shore servers constitute exclusively of *trusted* code, including those parts of the system that are provided as part of the standard Shore release, as well as code for Value Added Servers (VASs) that can be added by sophisticated users to implement specialized facilities (e.g., a query shipping SQL server). Application processes manipulate *objects*, while servers deal primarily with fixed-length *pages* allocated from disk *volumes*, each of which is managed by a single server.

The Shore server plays several roles. First, it is the page-cache manager. Second, the server acts as an agent for local application processes. When an application needs an object, it sends an RPC request to the local server, which fetches the necessary pages and returns the object. Finally, the Shore server is responsible for concurrency control and recovery. A server obtains and caches locks on behalf of its local clients. The owner of each page is responsible for arbitrating lock requests for its objects as well as logging and committing changes to the page.

4.7 Shore Software Components

The main software components of Shore (Figure 4.1) constitute the Shore server and the Language Independent Library. We next discuss these components in greater detail.

4.7.1 The Language Independent Library

The process of converting object references on disk to main memory addresses is called *swizzling*. When an application attempts to dereference an "unswizzled" pointer, the language binding generates a call to the object-cache manager in the *language independent library* (LIL). If the desired object is not present, the LIL sends an RPC request to the local server, which fetches the necessary pages by reading from the local disk.

To reduce paging, the object cache manager locks the cache in memory and uses LRU replacement if it grows too large. All OIDs in the cache are swizzled to point to entries in

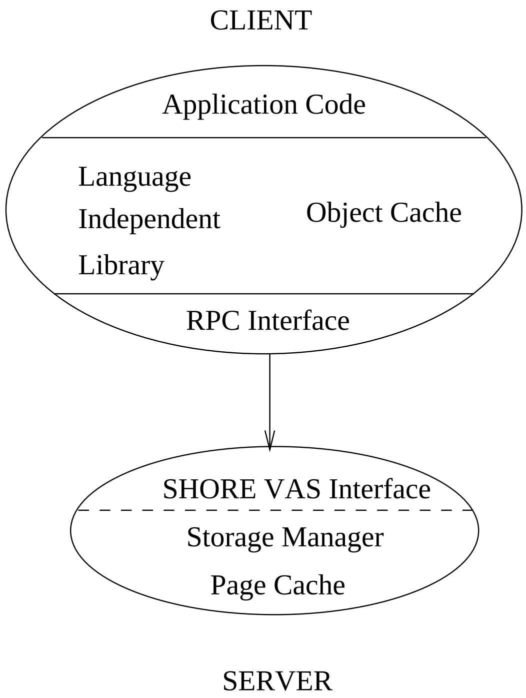


Figure 4.1: Application - Server Interface

an *object table.* This level of indirection allows objects to be removed from memory before the transaction commits, without the need to track down and unswizzle all pointers to them. Finally, the LIL is responsible for authenticating the application to the server using the Kerberos authentication system.

4.7.2 The Shore server

The Shore server (Figure 4.2) is divided into two main components: a *Server Interface*, which communicates with applications, and the *Storage Manager* (SM), which manages the persistent object store.

The Server Interface is responsible for providing access to Shore objects stored using the SM. It manages the Unix-like namespace. When an application connects to the Shore server, the server associates Unix-like process state with the connection such as user ID and a current directory name. User ID information is checked against registered objects when they are first accessed to protect against unauthorized access. As in Unix, the

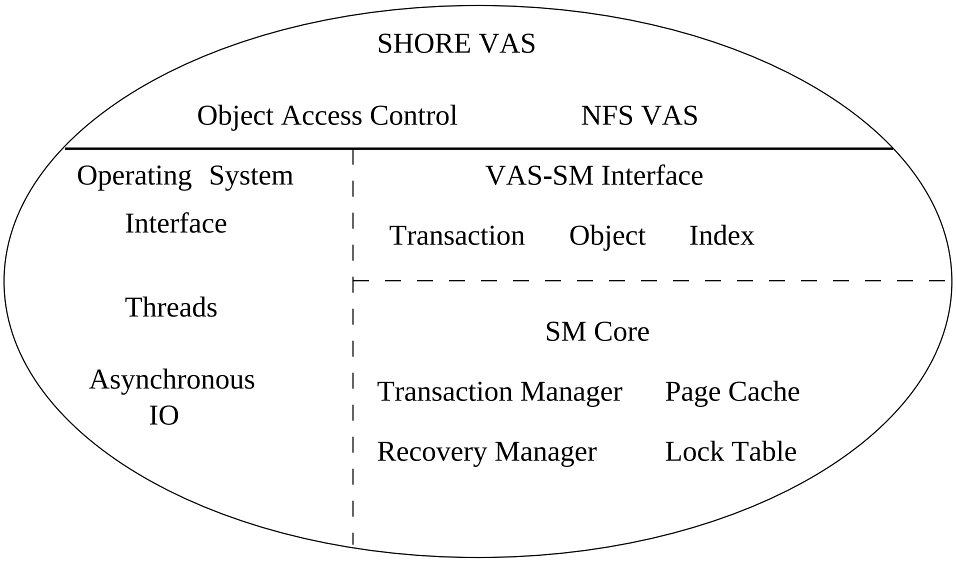


Figure 4.2: System Architecture

current directory name information provides a context for converting path names into absolute locations in the name space.

The Shore server code is modularly constructed so that users can build application-specific servers, thus supporting the notion of "value-added" servers (VAS). The Server Interface is an example of one such VAS. Another VAS is the NFS file server described earlier. Each VAS provides an alternative interface to the storage manager. They all interact with the storage manager through a common interface that is similar to the RPC interface between applications and the server. It is thus possible to write a new VAS as a client process and then migrate it into the server for added efficiency. Below the server interface lies the Storage Manager (SM). The SM can be viewed as having three sub-layers. The highest is the VAS-SM interface, which consists primarily of functions to control transactions and to access objects and indexes. The middle level comprises the core of the SM. It implements records, indexes, transactions, concurrency control and recovery. At the lowest level are extensions for distributed server capabilities. In addition to these layers, the SM contains an operating system interface that packages together multithreading, asynchronous I/O and inter-process communication.

4.8 Conclusion

Shore is an integration of file system and OODB concepts and services. From the file system world, Shore draws object naming services and an object access mechanism for use by legacy Unix file-based tools. From the OODB world, Shore draws data modeling features and support for associative access and performance acceleration features.

Chapter 5

Object Oriented Methodologies

5.1 Introduction

A *Methodology* is defined as comprising of a set of concepts, a notation(s), a process and pragmatics [Booc94]. Object oriented methodologies usually consist of Object Oriented Analysis (OOA) and Object Oriented Design (OOD) phases. Having chosen the object oriented model as the data model for our database system, the next step involves the choice of a suitable methodology to develop the system.

In this chapter, we consider some of the popular object oriented methodologies that are presently in use and compare the major ones. Finally, we choose a particular methodology suited for our system and outline the broad steps that will be followed to build the system according to the methodology.

5.2 Traditional Approaches

The structured analysis/structured design (SA/SD) approach [YoCo79, Your89] has been employed for software analysis and design for over two decades, and can be considered representative of traditional approaches to software development. The structured analysis phase has traditionally employed some of the following:

1. Data flow diagrams

2. The data dictionary

3. State transition diagrams

4. Entity-relationship (ER) diagrams

The structured design phase involves a top-down design consisting of grouping the processes from the data flow diagrams into tasks and converting the tasks into programming language functions. However, there has been a shift in this fundamental approach to software development propelled by the need to create systems that are easy to develop and maintain. This has resulted in the emergence of the more flexible and reusable object oriented approach. The object oriented paradigm relies on modeling the real-world domain naturally using the class and object as the building blocks in contrast to the structured design methods that use algorithms as fundamental elements. We next summarise the three major object oriented methodologies for software development: James Rumbaugh's Object Modeling Technique, the Booch technique of OOA and OOD, and the Fusion Method.

5.3 The Object Modeling Technique

The OMT methodology [Ru+91] consists of the following major phases: Analysis, System Design, Object Design, and Implementation. It employs three kinds of models as part of the analysis phase: the *Object Model*, describing the objects in the system and their relationships; the *Dynamic Model*, describing the interactions among the objects in the system; and the *Functional Model*, describing the data transformations of the system. These three models are considered as orthogonal views of the system.

An object model is considered as the first step in analyzing the problem. As part of the object model, classes are defined for abstractions that are relevant in the particular context. Object diagrams are created using the graphical notation to express these classes and the associations among objects. These diagrams are improved upon by adding more detail with attributes and use of inheritance.

The dynamic model is employed to specify temporal constraints. The sequences of events, the states of events, and the operations that are important are all included in the dynamic model. The dynamic model is important for applications where the user interface makes the application "event-driven". User-interaction sequences are created and their effect on objects is described in the form of state diagrams. Events that initiate interactions among objects are also identified.

The functional model is represented by data flow diagrams (DFD), where the processes correspond to activities or actions on classes, and the flows correspond to objects or attribute values in an object diagram. Thus, the DFDs are used to express the different states of objects and the state transitions taking place due to operations on the object.

The next phase, the system design phase, involves making high level decisions about the overall architecture. During system design, decisions regarding the performance characteristics to optimize, the resources to allocate, etc. are taken. The object design phase of software development that follows this phase incorporates design approaches to minimize execution time, memory and other resources. The object design phase is when the operations identified during analysis must be expressed as algorithms, and when the classes, attributes and associations from analysis must be converted to appropriate data structures. Finally, in the implementation phase, the object classes and relationships developed during object design are translated into a particular programming language.

5.4 The Booch Method

This methodology [Booc94] is one of the first of the complete object oriented methodologies to be formulated. Booch proposes different views to describe an OO system. These views are organized along two dimensions: one comprising the logical/physical view and the other the static/dynamic view. For a given problem, the products of analysis and design are expressed through logical, physical, static, and dynamic models. For each dimension, a number of diagrams are defined that denote a view of the system's models.

For the static view, the following diagrams are introduced:

- Class diagrams: The main elements of class diagrams are the classes and their relationships. During analysis, they represent the roles of the entities that provide the system behaviour while during design, they show the classes that form the system architecture.

- Object diagrams: An object diagram shows a snapshot in time of the system over a configuration of objects. It graphically shows objects and their relationships.

- Module diagrams: A module diagram shows the allocation of classes and objects in the physical design of the system through modules and dependencies.

- Process diagrams: A process diagram is used to show the allocation of processes to processors. The three elements of a process diagram are the processors, devices and the connections among them.

The first two diagrams represent the logical model while the remaining two the physical model of the system.

The dynamic view of the system is represented by:

- State Transition diagrams: The state of an object represents the cumulative results of its behaviour. At any given point in time, the state of an object encompasses all its properties (the object's attributes and relationships) and the values of each of these properties. A state transition diagram shows the state space of a given class, the events that cause the transition from one state to another, and the actions that result from a state change.

- Interaction diagrams: An interaction diagram is used to trace the execution of a scenario in the same context as an object diagram. The interaction diagram gives the sequence of messages in the scenario. It restructures the essential elements of an object diagram and shows the relative order of passing of messages for a scenario. To a large extent, it is simply another way of representing an object diagram.

5.5 The Fusion Method

The Fusion method [Cole+94] is a recent approach to object oriented software development. It has integrated and extended existing approaches to provide a direct route from a requirements definition to a programming language implementation. It consists of three main phases: Analysis, Design, and Implementation.

The goal of the analysis phase is to start from the requirements definition document containing details of the problem, probable solutions and how a system that solves the problem ought to work. At the end of analysis, we have the Object Model and the Interface Model, which consists of the Operation Model and the Life-Cycle Model.

The Object Model captures the concepts that exist in the problem domain and their interrelationships using an extended entity-relationship approach. It represents the static structure of the information in the system consisting of classes and relationships between them, attributes of the classes, aggregation and specialization/generalization. The System Object Model is a subset of the object model excluding those that belong to the environment.

The Interface Model defines the input and output communication of the system. The description is in terms of events and the change of state that they cause. A system is modeled as an active entity that interacts with other active entities called *agents*. The agents may be human users, or other hardware or software systems. An Interface Model uses two models to capture different aspects of its behaviour. The Operation Model specifies the behaviour of system operations declaratively using Operation Model Schema. It defines operation effects in terms of change of state and the events that are output. The Life-Cycle Model describes how the system communicates with its environment from its creation until its death. It consists of life-cycle expressions. A life-cycle expression defines the allowable sequences of interactions that a system may participate in during the course of its lifetime.

The Design stage involves developing an abstract model of how a system implements the behaviours specified by analysis. The designer chooses how the system operations are to be implemented and the inheritance relationships among classes. The following four design models are developed: *Object Interaction Graphs*, describing how objects interact at run-time to support the functionality specified in the operation model in the analysis phase; *Visibility Graphs* describing the object communication paths; *Class Descriptions* providing a specification of the class interface, data attributes, object reference attributes and method signatures for all the classes in the system and, *Inheritance Graphs* describing the class/subclass inheritance structures.

The Implementation phase involves the actual programming required to build the system. The methods are defined by the preconditions and postconditions of the method. An error is defined as a violation of a precondition. Error recovery takes place through exception handling. Finally, the system undergoes testing.

Fusion borrows the object model and the development process from OMT while the design stage partly follows Booch.

5.6 Other Object Oriented Approaches

Apart from the previous three methodologies, various others have been proposed. The Shlaer-Mellor approach [ShMe88] advocates the development of three models, the Information Model, the State Model and the Process Model. The Wirfs-Brock method [WiBr90] constitutes identifying classes and the relationships. The relationships are recognised as a contract between two collaborating objects. In the Object Oriented Design method of Coad and Yourdon [CoAd91], design involves four components, the Problem Domain Component, the Human Interaction Component, the Task Management Component and the Data Management Component.

5.7 Comparison of OO Methodologies

We compare the described methodologies using the following: concepts, notation, and the process. The concepts term denotes the definition of object oriented concepts as defined by the methodology and represents the "object orientedness" of the methodology. The concepts include class, object, aggregation, inheritance, and association. The notation involves the graphical representation for the models of the methodology. The process is the various steps involved in the analysis and design stages of object oriented software development.

All three models discussed are fully object oriented and hence support all object oriented concepts. The Rumbaugh method offers a concise notation. The Booch method is descriptive, and hence there is no well defined sequence of steps to follow the process in contrast to Rumbaugh's OMT. The Fusion method integrates other methods and is largely influenced by both Booch and OMT methods. The distinction between Booch's approach and the OMT approach is the emphasis the OMT places on associations. However, the similarities between the approaches are more striking than the differences and the two complement each other. Shlaer and Mellor's methodology like the OMT methodology, breaks analysis down into three phases: static modeling of objects, dynamic modeling of states and events, and functional modeling. A flaw in Shlaer and Mellor's treatment is however their excessive preoccupation with relational database tables and database keys. Researchers at Hewlett and Packard [MoPu92] have conducted a study of the differences between these OO methodologies. The study concluded that the Booch, Rumbaugh and Wirfs-Brock methodologies fully support OO concepts. The Wirfs-Brock methodology helps exploratory analysis and informal design, and might prove useful for analysis or high-end design. Fusion uses various other models to show various aspects of the system. It is not immediately obvious why one would need them all. With OMT for example, the concept of three simple models is easier to get across than a number of interconnected models. Also, OMT models are not as interconnected as Fusion and hence OMT models can be used separately.

5.8 Choice for DIAS

We choose the OMT methodology of software development for the following reasons. Our database system will be used as a static repository of data and hence the object model is the most important. In fact, dynamic modeling and functional modeling are not necessary for the development of our database system [Ru+91] and we shall not be developing them. Of the methodologies considered, OMT is the one which specifically emphasises the object model. The Fusion method, to a large extent borrows the OMT technique for its object model. A striking feature of the OMT methodology is the major role played by associations, a fact which fits in with the importance of associations among entities modeled for representation in database systems. Another advantage is the simplicity and conciseness of the notation compared to the Booch method. Moreover, the OMT technique has been successfully applied to the development of database system applications earlier [BPR88]. Over the past few years, the OMT technique has become so popular that now efforts are underway to merge the Booch method with OMT.

The usual lifecycle for database system applications includes the following steps:

- Design the application
- Select a database engine to serve as the backend platform
- Design the database
- Write programming language code
- Populate the database
- Develop application specific features for the system

Of the above tasks, the most important is the database design referred to as the *schema*. The standard architecture that is usually followed is the *three schema architecture*, consisting of the *external, conceptual and internal schemas*. The external schema is a view or abstraction of the overall conceptual schema. The conceptual schema is the database design for a particular problem domain. The internal schema consists of the actual DBMS code required to implement the conceptual schema.

The development of our object oriented database system for VLSI interconnect parasitics involves integrating the above steps with the OMT methodology to arrive at the stages of development. The Object Model is used for the external and conceptual schema design. The internal schema now falls into the Object Design and Implementation stages. We next discuss the stages that we shall be following in developing the database system.

Analysis

Analysis, the first step of OMT methodology, is concerned with devising a complete model of the application domain. First, it involves gathering all information about the problem domain. These constitute the data required to be stored, the constraints and the relationships among the various entities. The result is a *problem statement* defining the requirements of the system. From the problem statement, the *object model* is developed using the graphical notation of OMT.

The following steps are performed in constructing an object model:

- Identify objects and classes
- Identify associations and aggregations between objects
- Identify attributes of objects and links
- Organize and simplify classes using inheritance
- Verify that access paths exist for likely queries
- Iterate and refine the model
- Group classes into modules, based on the degree of coupling and how close they are related functionally

System Design

Once the analysis is done, system design is the high-level strategy for solving the problem and building a solution. The overall *system architecture* is decided at this stage. Then, interfaces to store and manipulate the database are designed. Mechanisms for interaction of existing tools with the data are developed. All these subsystems are organized into a whole system.

Object Design

Based on the analysis model, a design model is built that incorporates implementation details. The focus of object design is the data structures and algorithms needed to implement each class. The object classes from analysis are still meaningful, but they are now augmented with computer-domain data structures and algorithms chosen to optimise important performance measures. During object design, the following steps are performed.

- Obtain operations on classes designed earlier
- Design algorithms to implement operations
- Optimize access paths to data
- Adjust class structure to increase inheritance
- Design associations
- Package associations and classes into modules

The operations for classes must be formulated as algorithms. Data structures on which these algorithms work on are declared next. Frequently, these algorithms require defining new auxiliary classes and operations.

From a database viewpoint, for efficient query processing, optimization of access paths to data is important. In relational database systems, this involves declaring new storage structures such as hash tables and B^+ trees for accessing data quickly. In an object oriented database system, these access methods are not sufficient and hence specialised access methods are designed.

Classes can be rearranged to increase inheritance among groups of classes. Packaging related classes into discrete physical units such as source files improves coherence among entities and limits the scope of each class. This stage marks the end of the design phase of the development process.

Implementation

The object classes and relationships developed during object design are finally implemented using the programming language of the backend database system. The various steps involved are:

- Class definitions
- Creating objects
- Implementing operations
- Using inheritance
- Implementing associations
- Implementing access methods

The design methodology is by nature an iterative one and several passes have to be made through these phases to arrive at the final design of the database system. In the next chapter, we describe the analysis phase. Chapter 7 describes the system design and object design phases while the implementation phase is covered in Chapter 8.

Chapter 6

Analysis

In this chapter, we look at the analysis stage of the development of the database system for managing interconnect parasitics data. The analysis stage is concerned with devising a precise and correct model of the problem domain. The initial stages of building the software system consist of understanding the requirements along with the environment in which the software will be used. This leads to the formulation of a problem statement.

As mentioned in the introduction, the study of interconnect parasitics calls for effective data management using a database system. The database system forms the hub of the system for interconnect study. Various tools are used to input data into the system. The stored data must be accessed efficiently for answering queries. Circuit simulation tools must be able to extract data from the database system. Application programs which process the data must be capable of being developed quickly.

The various data can be classified into design-related data and analysis-related data. The design data consists mainly of data regarding the various components of a circuit design. The analysis data deals with the data introduced during the interconnect study of the designed circuit. Broadly, the data falls into the following major groups:

- Netlist data: A netlist shows the electrical connectivity of the circuit. The data involved are those comprising the design entities such as cells, ports, etc.

- Layout data: The layout related information regarding the design elements and the parasitic data fall under this category. Some of the data stored include layout geometries, length, width of various metal layers, etc.

- Characterization data: The library cells used in the design have characterization information. These are used for subsequent calculations for a design. The data involved include the output capacitance for the different ports of a cell, etc.

- Parasitic data: The main data that is stored in DIAS system deals with parasitics and hence this forms the bulk of the data in the database. For each signal in the circuit, details about the interconnect model are stored in the database. Also, using that interconnect model, the values obtained for different parameters of interest such as slew at the output terminal, the delay due to RC elements in the interconnect suffered by a signal while travelling from input port to output port, etc. are stored.

6.1 Problem Statement

Summarizing the above details, we have the following as the problem statement that characterizes the requirements of the database system:

1. The database system should be capable of supporting all the different kinds of data discussed earlier.

2. The system should be capable of speedily answering queries to retrieve a subset of data.

3. Existing tools should be capable of accessing and loading data without any changes to the tool itself till completely new tools are written that can directly access the database.

4. A Programming Interface should be provided to facilitate the easy development of new application programs around the database system.

The next step is to develop a comprehensive object model which captures all the information that are required to be stored in the database.

The Object Model

As seen in Chapter 5, the development of an object model involves various stages such as identifying the classes, associations, attributes and operations, simplifying the classes using inheritance and finally verifying the existence of access paths for likely queries. We discuss here each of the steps in the object model development. The model is pictorially represented using the OMT notation (Appendix B).

6.2 Identifying Classes

To model the complex data of the domain, we abstract common properties of entities in a *class*. The *Dias* class represents any entity found in the domain. The *Property* class is used to denote a (name, value) pair which can be attached to any design object. A *Text* class is used to represent design and analysis management information.

The information that is stored in our DBMS is mainly concerned with *interconnect parasitics* data. Apart from this data, we also need to store netlist related data, layout related data, and characterization data. These form the basic design data. The details regarding the netlist are needed for storing the connectivity of the design. The characterization information is necessary for calculating parametric data. The layout geometries, location and other layout information are required for calculating the interconnect parasitics and subsequently incorporating them as part of the designed circuit for analysis using simulation tools. We next describe the classes designed for adequately representing these various types of basic design data.

Basic design data

The class *Cell* is used to model the concept of a design entity. The classes *LibraryCell* and *Design* represent a leaf cell (primitive element) and a complex design, respectively. The information about the transistors comprising a *LibraryCell* is modeled using the *Transistor* class. The intrinsic delays suffered by a signal while travelling from the input port to the output port of a cell are stored using an object of the *InterPort Info* class.

A *Design* object has several data associated with it. The data is divided into the following classes: *DesignParameters*, *ProgramStamp*, *Technology* and *Units*. The *DesignParameters* class stores information such as the date of creation, the directory in which it was created and so on. The *ProgramStamp* class has similar details about the program which created it. The *Technology* class has information regarding the technology to which the design belongs. The units for the various values used in the design are stored in an object of the *Units* class.

Other basic entities such as the ports of a cell and the nets in a design directly map to the classes *Net*, *Port*, etc. A collection of cells may have certain common characteristics such as the same technology. The *Library* class groups together such a collection of cells.

A net could be physically divided into a number of disjoint interconnects. We make a distinction between two kinds of nets: the ones that are connected to the boundary ports of a cell and those which are fully internal. In case of the former, additional information regarding the port of the external cell which is driving the IO-net is stored. This distinction results in the two classes, *IONet* and *Signal* respectively.

The characterization data represent the properties of the library cells which are used in a design. The structure of the characterization data is usually vendor-specific and hence shows frequent changes. We now discuss how the characterization data for library cells is modeled.

The characterization data supplied with a library cell usually consists of the following: the information about the ports (pins) of a library cell and the intra-cell timing information (buffer characterization information). The information about the pins is captured by the *PinInfo* class. It contains a list of entries, with each entry storing details about the pin name, direction and the capacitance value observed at that pin. The buffer characterization information is composed of different *Arc* objects, with each *Arc* object storing information about a particular property for all the pins of the library cell. Each *Arc* is composed of the *Group*, *OutputDelay* and *OutputSlew* classes. The *Group* class abstracts a logical grouping of all pins with the same information in the library cell. The details stored are the two pins about which it stores the information, the timing sense (Low-High or High-Low), and the Transition Type (one combination among the Low, Zero, and High transitions). The *OutputDelay* and the *OutputSlew* classes are associated with a "look-up" table containing pairs of values of input slew and output load. The *OutputDelay* class thus has a list of corresponding loading delay values for each pair in the look-up table and similarly, the *OutputSlew* class has a list of corresponding output slew values for each pair in the look-up table.

Interconnect Parasitics Data

We shall now describe the major classes designed for parasitics related data. A net is logically divided into a number of Parasitic Nodes and Parasitic Devices. These fall into the *ParasiticNode* and *ParasiticDevice* class. Earlier, the *Design* class was broken into *Net* class and *Device* class. In a similar manner, the *Net* class is broken into *ParasiticNode* and *ParasiticDevice* class. Also, parasitic effects between nets are represented by the *Coupling* class.

Interconnect Models

The interconnect parasitics information is modeled using electrical circuit equivalents. The following are some of the major types of interconnect models with varying levels of accuracy. We discuss how each of them is represented by means of classes.

- Single Lumped Capacitance

  This is the simplest electrical equivalent. In this model, an interconnect of a net is represented by a single capacitance from the interconnect to the ground. Thus, this ignores resistive effects. This is modeled by means of a *LumpedCapacitance* class which stores the capacitance value.

- Resistance Capacitance (RC) Network

  This includes the resistive effects also and hence a net is represented by means of an RC tree. An RC tree is made up of several RC elements, each element representing a section of an interconnect. We model the RC Network for a net by treating it as a graph modeled using the *Node* and *Edge* classes. A *Node* object is connected to a set of *Edge* objects. An object belonging to the *Edge* class stores the resistance and capacitance values of a section of the interconnect. In a model of a long-branched interconnect, thousands of such *Nodes* and *Edges* are present in an RC tree. This data tends to be too vast for analysis applications leading to delay in processing. As a result, at the cost of accuracy, the number of nodes and edges is often reduced.

- Transmission Line Model

  This is more accurate than the previous form since inductive effects of the interconnect parasitics are also taken into account. Thus, a single signal is represented by means of a Transmission Line model consisting of several uniformly distributed elements. Each such element is represented using an object of the *UnitElement* class, that stores the information regarding the resistance, capacitance, and inductance values for unit length of the transmission line.

The object model can be extended easily for other interconnect modeling techniques. The *InterconnectionModel* class serves as the abstract base class which is derived by all these models.

The effects of the interconnect parasitics on the performance of a circuit are modeled by the *ParametricInfo* class. These include timing, crosstalk and power effects. The information obtained at the *Prelayout* and *Postlayout* stages of a design are separately considered. Also, the effects are found at different granularities of the circuit such as in a path of the circuit, at a physical point, etc.

6.3 Relationships

Relationships among the entities in the domain are modeled using associations. A *link* is an instance of an association. We next discuss some special relationships among the entities and the modeling of these relationships.

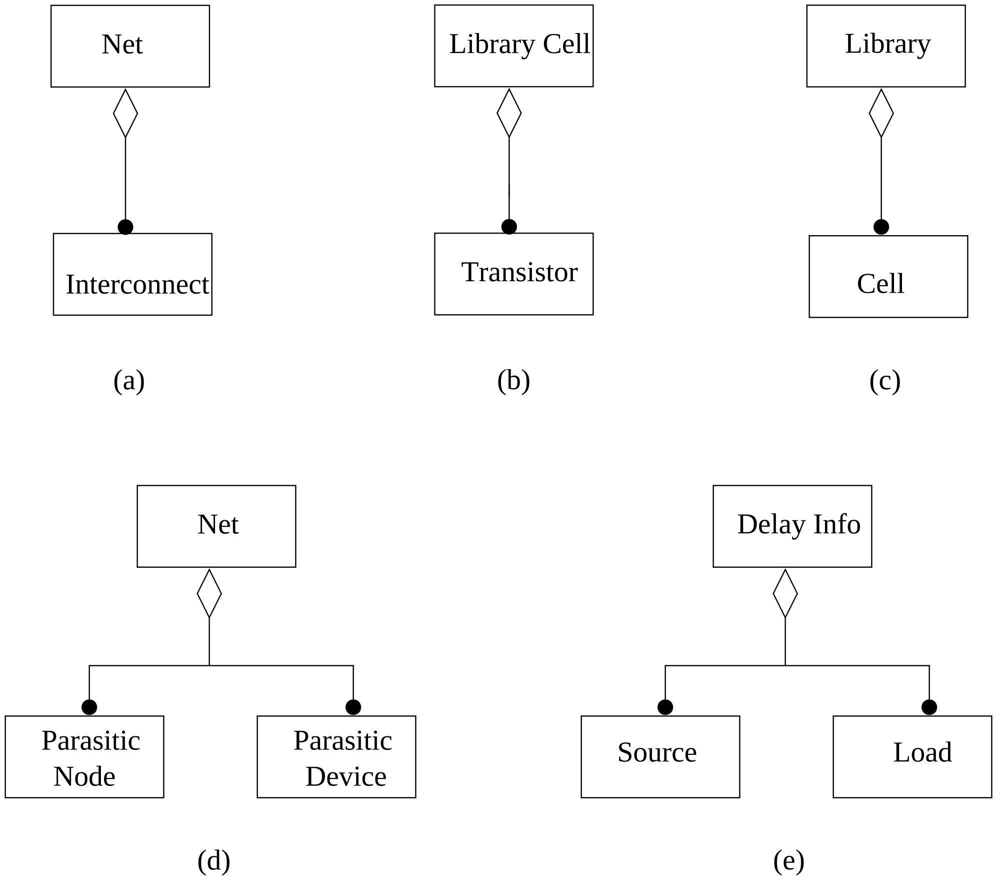


Figure 6.1: Example Aggregations in the DIAS object model

Aggregations

An aggregation represents the "is-a-part-of" relationship between objects. An aggregation hierarchy is a hierarchy of instances. We now describe the various aggregation hierarchies used to model the information in this domain.

- Net hierarchies: A *Net* figures in two aggregation hierarchies. A net is physically composed of disjoint *Interconnect* objects of the layout (Figure 6.1 (a)). From a parasitic viewpoint, the parasitic extraction can be done on the basis of breaking down a *Net* class into logical parasitic nodes belonging to the *ParasiticNode* class and logical devices belonging to the *ParasiticDevice* class (Figure 6.1 (d)). The parasitic nodes map to specific points on an interconnect while the parasitic devices map to sections of an interconnect. In case the parasitic device stores parasitic information between two interconnects, then it does not map to any physical interconnect.

- Transistors: Though the *LibraryCell* is the leaf cell in a design hierarchy, we store details of the transistors also as an aggregation using the *Transistor* class (Figure 6.1 (b)).
- Collection: A collection is a logical grouping of classes having some common features. A *Library* class is a collection of *Cell* classes which have the same characterization information (Figure 6.1 (c)).
- Delay information: The delay information regarding a signal is represented by means of the sources and loads, represented using the *Source* and *Load* classes respectively (Figure 6.1 (e)).

Other aggregations used in describing the interconnect model and the characterization data are dealt with separately in later sections.

Instantiation

Consider the following example. The design of an adder has four *copies,* called *instances* of an adder-slice circuit. Each instance is distinct, that is, each has its own separate set of inputs, outputs, and co-ordinate positions on a circuit diagram. However, each shares the same adder-slice features. Thus, each of these instances are simply reproductions of a "master", in this case, the adder-slice circuit. Instantiation, leads to an implicit inheritance of attributes from the "master", in addition to non-inherited attributes. For example, in the above case, the non-inherited attributes are co-ordinate positions, inputs and outputs specific to an instance.

We capture the instantiation concept by introducing the *Device* class. The *Device* class and the *Cell* class are related by means of the "is-instance-of" relationship. Thus, an object belonging to the class *Design* has several objects of the *Device* class, each of these being an "instance" of a *Cell*. Note, that the term "instance" is also used in the context of an object being an "instance-of" a class. To disambiguate the usage of the term in this thesis, whenever instance is used to mean the relationship between the *Cell* and *Device* classes, it is surrounded in quotes.

Recursive Aggregation

A recursive aggregate contains, directly or indirectly, an instance of the same kind of the aggregate. This leads to an aggregation hierarchy whose number of levels is potentially unlimited. For example, a computer program is an aggregation of *blocks* of statements. Those blocks may be either compound or simple statements. The compound statements recursively contain other blocks until the recursion terminates with a simple statement.

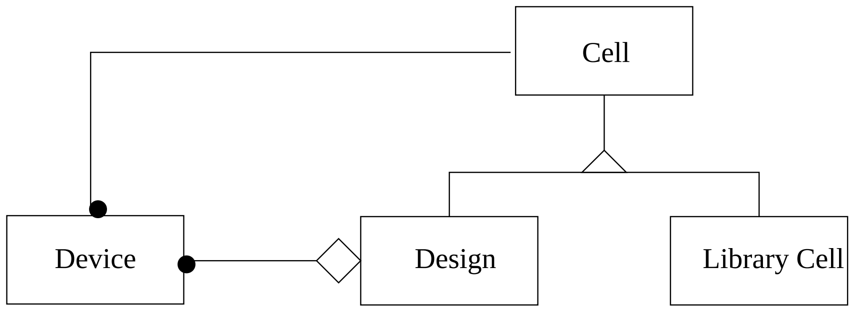
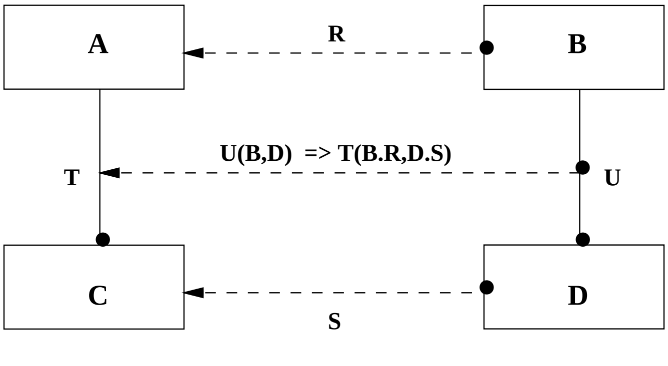


Figure 6.2: Recursive Aggregation

In our domain, recursive aggregation is exhibited by a hierarchical design consisting of several levels of nested subdesigns. The class *Cell* forms the superclass for two subclasses, *LibraryCell* and *Design*. The *LibraryCell* forms the primitive element of a design which is not subdivided further. The *Design* class is composed of *Device* class, which is an "instance" of a *Cell* class. A *Design* object is in turn composed of other *Device* objects and so on till a *LibraryCell* object. Here, the recursive aggregation is indirect, with the *Device* class forming the implicit link. The relationships are depicted in Figure 6.2.



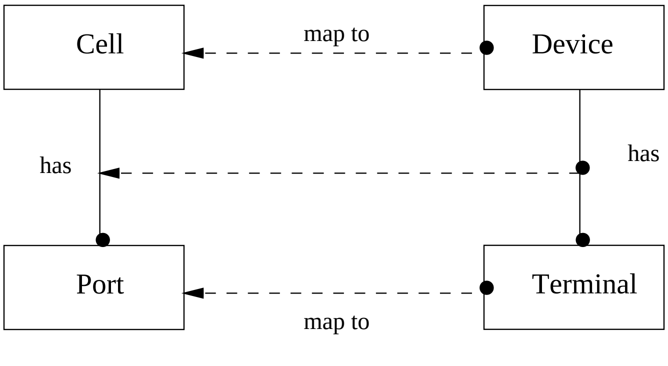


Figure 6.3: The Homomorphism relationship

Homomorphism

A *homomorphism* is a special relationship that maps between two associations. For example, in a parts catalog for an automobile, a catalog item may contain other catalog items. Each catalog item is specified by a model number that corresponds to thousands or millions of individual manufactured items, each with its own serial number. The individual items are themselves composed of subitems. Each physical item's parts' explosion tree has the same form as the catalog item's parts explosion tree. Thus, the *contains* relationship on catalog items is a homomorphism of the *contains* relationship on physical items.

In general, a homomorphism involves four relationships among four classes as shown in Figure 6.3 (a). The homomorphism maps links of one general association (U) into links of another general relationship (T) as a many-to-one mapping. R is a many-to-one relationship from class B to class A and S is a many-to-one mapping from class D to class C.

The concept of homomorphism can be used to represent the relationships between the *Cell*, *Port*, *Device*, and the *Terminal* classes. A *Cell* has a set of *Port* objects which form the electrical interface to the external circuit. A *Device* is an "instance" of a *Cell*. A *Device* object has *Terminal* objects which are "instances" of *Port* objects. Then, the links of the association between a *Device* and its *Terminal* objects is mapped by a homomorphism into links of the relationship between a *Cell* and its *Port*s as a many-to-one mapping. Thus, as shown in Figure 6.3 (b), the four relationships are:

- R: *Device* $\rightarrow$ *Cell*, a many to one relationship, *is-instance-of*
- S: *Terminal* $\rightarrow$ *Port*, a many to one relationship, *is-instance-of*
- T: *Cell* $\rightarrow$ *Port*, a one to many relationship, *has-ports*
- U: *Device* $\rightarrow$ *Terminal*, a one to many relationship, *has-terminals*

Then, the homomorphism maps links of relationship U into links of relationship T as a many to one mapping, that is, U(*Device*, *Terminal*) $\Rightarrow$ T(*Device.R*, *Terminal.S*).

6.4 Link Attributes

A class has properties represented as attributes. Link attributes are properties of links in an association. Many-to-many associations provide the most compelling rationale for link attributes. Such an attribute is a property of the link and cannot be attached to either object. Since these link attributes may be subject to operations (methods), we model the association as a *class*.

Some of the link attributes which are used to model in our domain are the following.:

- *Inter – net association:* The effect of the signal in one net causes parasitic effects in neighbouring signals. This is represented by the *Coupling* class. It may be denoted by a uniformly distributed capacitance or by a single value denoting the lumped capacitance. The values of these are stored in a *Coupling* object.

- *Port - Port Delay:* As noted earlier, a cell has a list of ports. The *InterPortInfo* object is a link object connecting two *Port* objects. It stores information regarding the intrinsic delay suffered by a signal in travelling between the input port and the output port. Similarly, the timing information between any *Source-Load* pair of terminals is represented by the *SourceLoadTiming* class.

Also, the node-to-node delay between two *Node* objects is depicted by the *NodeToNodeDelay* class. Figure 6.4 shows these associations.

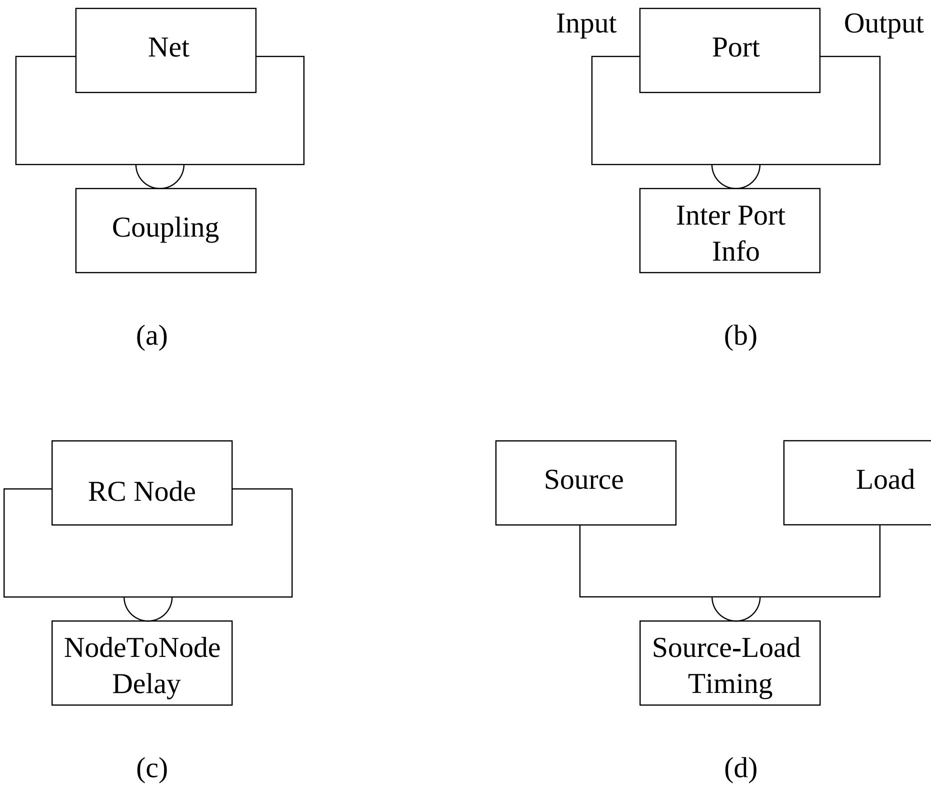


Figure 6.4: Link Attributes in the object model

6.5 Other Relationships

We now discuss other general associations among classes. Because of strong interdependence, we see many associations among classes. We briefly discuss the major ones here.

Typical layout details such as width, length of wire and geometries are stored in the *LayoutDetails* class. A *LayoutDetails* class is associated with an *Interconnect* class for which the *LayoutDetails* class stores layout details. Each *Terminal* object is associated with a *Device* object which is its "owner". The *Source* and *Load* classes have associations, "map to", to the *Terminal* class. Similarly, the node of an *RC Tree* "may map to" the *Terminal* class if it is one of the boundary nodes of the RC Tree. The *LibraryCell* class has associations to the classes which represent characterization data, namely the *TimingInfo* and the *PinInfo* classes.

6.6 Attributes and Operations of Classes

The values of the attributes represent the state of an object and actually store the data. The attributes of the *Design* class store information about the design as a whole. Because of the diverse nature of the parasitic information, the data representing this information is distributed among the attributes of the various classes that represent the structural interconnect model and the classes that represent parametric information.

Operations (methods) are used to modify and retrieve the state of an object. Simple operations get or set the attribute values. Others are used for querying. For example, given the name of a signal of a design, a get operation retrieves the signal object.

6.7 Inheritance

Some of the classes defined so far exhibit similarities in their structure and behaviour. The inheritance relationship defines a mechanism wherein one class refines the structure or behaviour of other classes. The following inheritance hierarchies are noted in the object model. Figure 6.5 shows some of these hierarchies.

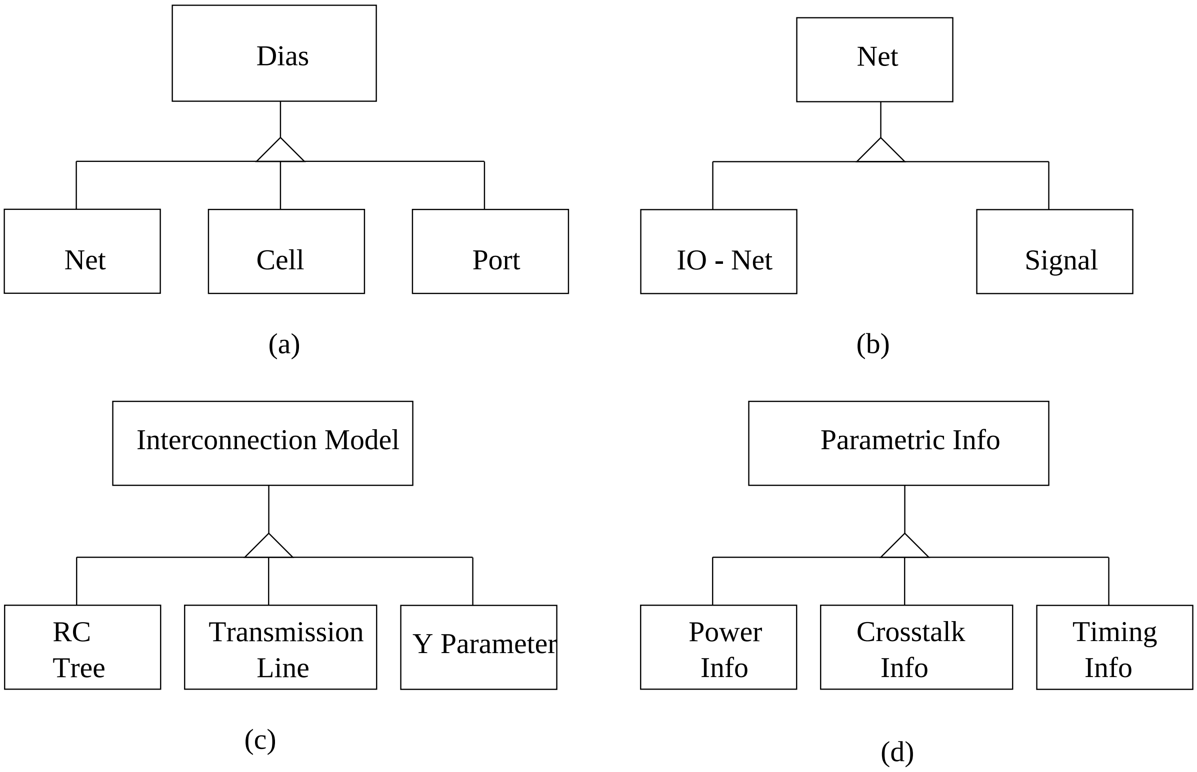


Figure 6.5: Example Inheritances in the DIAS object model

- *Base hierarchy:* The *Dias* class forms the base class for all objects. Design Management Information is stored in the form of free text in the *Text* class. A *Property* class is used to store one or more *name, value* pairs with any object. The refinements of the *Dias* class form the basic entities of the design domain, namely *Net*, *Cell*, and *Port* (Figure 6.5 (a)).

- *Derivations of Net class:* The *Net* base class is subsequently derived into *IO-Net* and *Signal* classes (Figure 6.5 (b)). The *IO-Net* class represents those *Net* objects that connect *Terminal* objects of a *Device* to the boundary ports of the *Design* object that contains the *Device* object. On the other hand, *Signal* objects connect only *Terminal* objects of *Device* objects. In all other respects, the two are similar. In case of *IO-Net* objects, additional information regarding the driving net that acts as input from outside the interface of the *Design* object is stored.

- *Interconnection Model hierarchy:* The abstract base class *InterconnectionModel* represents the structural interconnect model used for modeling interconnect parasitics. The specific model used forms a derivation of the base class. Examples are the widely used *RCTree*, *TransmissionLine* and *YParameter* techniques (Figure 6.5 (c)).

- *Parametric Information:* The parametric information calculated using a specific interconnect model is represented using the *ParametricInfo* class. The derivations of this class are the classes used to store different parameters such as power, crosstalk, and timing information (Figure 6.5 (d)).

- *Timing Types:* For different timing related data to be easily represented, we have two derivations of the base class, the *Coefficient* class and the *Delay* class. The *Delay* class is used to represent the different delays associated with the combinations arising among transitions of Low, High and Zero states such as LH, LZ, ZH, ZL etc. For each of these transitions, we store a triplet of Minimum, Nominal and Maximum values. The *Coefficient* class consists of a list of coefficient values which form the coefficients of an equation used to calculate the delay values.

6.8 Normalization of the Object Model

Normalization, for relational databases, is the process of decomposing the relational schema by breaking up their attributes into smaller relation schemas that possess certain desirable properties. Normalization is performed because a poorly designed relation incurs the overhead of redundant data and the risk of causing update anomalies. In short, a fully normalized relation is one whose only dependencies are functional dependencies where every non-key attribute is determined by the key attribute and there is no multi-valued dependency.

The normalization process takes a relation schema through a series of tests to "certify" whether or not it belongs to certain *normal forms*. Codd [Codd72], proposed the three normal forms, the first, second and third normal forms (1NF, 2NF, 3NF) and later a stronger version of the 3NF was proposed by Boyce and Codd [Codd74], called the Boyce

Codd Normal Form (BCNF), based on the functional dependencies among the attributes of a relation. Further normal forms, such as the 4NF and 5NF exist, based on multivalued and join dependencies respectively. However, in practical design cases, the 3NF and BCNF are sufficient [Lee95].

However, the object oriented data model differs substantially from the relational data model. The following characteristics of an object class make its normalization distinct from that of a relation.

- Because of aggregation, an attribute of a class can be an instance of another class.
- The attributes of an object can be containers for other objects such as list, set, etc.
- Objects are uniquely identified by their object identifiers that are assigned by the system. Hence, there is no need for the notion of a key attribute as in the case of relations.

As a result, there are no equivalent normal forms for an object class. Both complex attributes and multi-valued attributes make an object class non-1NF. Even if all attributes are simple and single-valued, the lack of a key attribute makes it violate 2NF. As pointed out in [Pratt94], some proponents of the object oriented approach claim that there is no need to normalize. This claim is based on the fact that there is no notion of a key attribute in an object class. In the succeeding paragraphs, we observe that object classes also suffer from some of the anomalies of relational schemas and hence require normalization. We then provide definitions of the object functional dependency and the object normal form as proposed in [Lee95]. We then normalize the object model developed for DIAS.

Update anomalies in object classes

Unlike the case of a relation, there is no insertion anomaly unless there exists a constraint prohibiting a null value for the attributes. For example, we can create an object and insert only the pertinent attributes. However, there are deletion and modification anomalies. The existence of these anomalies is sufficient reason to justify the need for normalization.

Object functional dependency

Given two attributes X and Y that belong to the same object class C, Y is said to be object functionally dependent on X, if and only if the value of Y is determined uniquely for each value of X.

Object Normal Form

An object class C is said to be in the object normal form if and only if the only functional dependencies are those determined by the object identifier (OID) of the class C.

Steps for normalization to an object normal form

If an object class does not satisfy the object normal form, then, the following steps can be followed to ensure that the resulting classes are in the object normal form.

1. Create a new class and introduce an attribute that refers to this class.
2. Check for attributes that form object functional dependencies with the new reference attribute.
3. Move these attributes to the referenced class, renaming the attributes if necessary.

The above steps are elucidated by the following example. Consider the following class:

```
class Employee
{
        integer EmpCode;
        string Name;
        integer Salary;
        string DeptName;
        integer DeptBudget;
        set<string> DeptLocations;
};
```

We can see that the Employee class suffers from deletion and modification anomalies. If we want to delete all the information regarding the Department, then all employees have to be deleted. Also, for a Department with only one Employee, we inadvertently lose the Department data if the Employee object is deleted. Likewise, in order to change the data about the department, we have to change all the objects of the employees working for the department.

Therefore, we apply the normalization steps to the class Employee. First, we introduce a new class Department and create a new attribute *Dept* belonging to the class Department. It is now exposed that the modified Employee class is not in the object normal form because of the additional object functional dependencies between dept (oid) and the attributes DeptName, DeptBudget, and DeptLocations. Hence, these attributes are moved to the class Department and renamed as Name, Budget, and Locations respectively. So, the final form of these two classes are:

```
class Employee
{
        string EmpCode;
        string Name;
        integer Salary;
        ref<Department> Dept;
};

class Department
{
        string Name;
        integer Budget;
        set<string> Locations;
};
```

Thus, it can be seen that these anomalies exist because of the fact that during the design of the classes, those attributes that should have actually belonged to an independent class are grouped as part of the same class.

In our object model, we begin by examining each class to see if it violates the object normal form. Then, each such class is subjected to the above three steps to ensure that it is normalized and hence does not suffer from update anomalies. We find that the use of modeling concepts such as "instantiation" results in classes that are normalized. The Cell and Device classes, the Port and Terminal classes are examples. At the end of this stage, we have the object model that is fully normalized with respect to the object normal form.

Summary

The object model we have proposed attempts to satisfy the modeling requirements mentioned earlier in Section 6.1. The aggregation and inheritance features help in representing the complex nature of the information directly in a natural manner. The entire model is shown in Appendix C. This concludes the analysis stage.

Chapter 7

Design and Optimization

After the analysis stage, we next consider the overall design of the database system. As mentioned in Chapter 5, *system design* includes the high-level strategy for solving the problem and building a solution while the object design phase is concerned with the design of appropriate data structures and algorithms. We discuss both these stages in this chapter.

7.1 DIAS System Architecture

DIAS follows a three-layered architecture as shown in Figure 7.1. As seen in Chapter 3, the SHORE system forms the lowermost layer of DIAS. On top of Shore is built the DIAS Kernel, that implements the object model developed in Chapter 6. The interfaces provided by DIAS form the topmost layer of the system and cater to the different user requirements. We next discuss briefly the components of DIAS and then discuss the various optimizations during the design of DIAS.

SHORE Server

The Shore server forms the lowest layer of DIAS. The Shore Storage Manager (SSM) is responsible for object management and transaction related features such as concurrency control and recovery. The Value Added Server (VAS) uses these features of the Storage

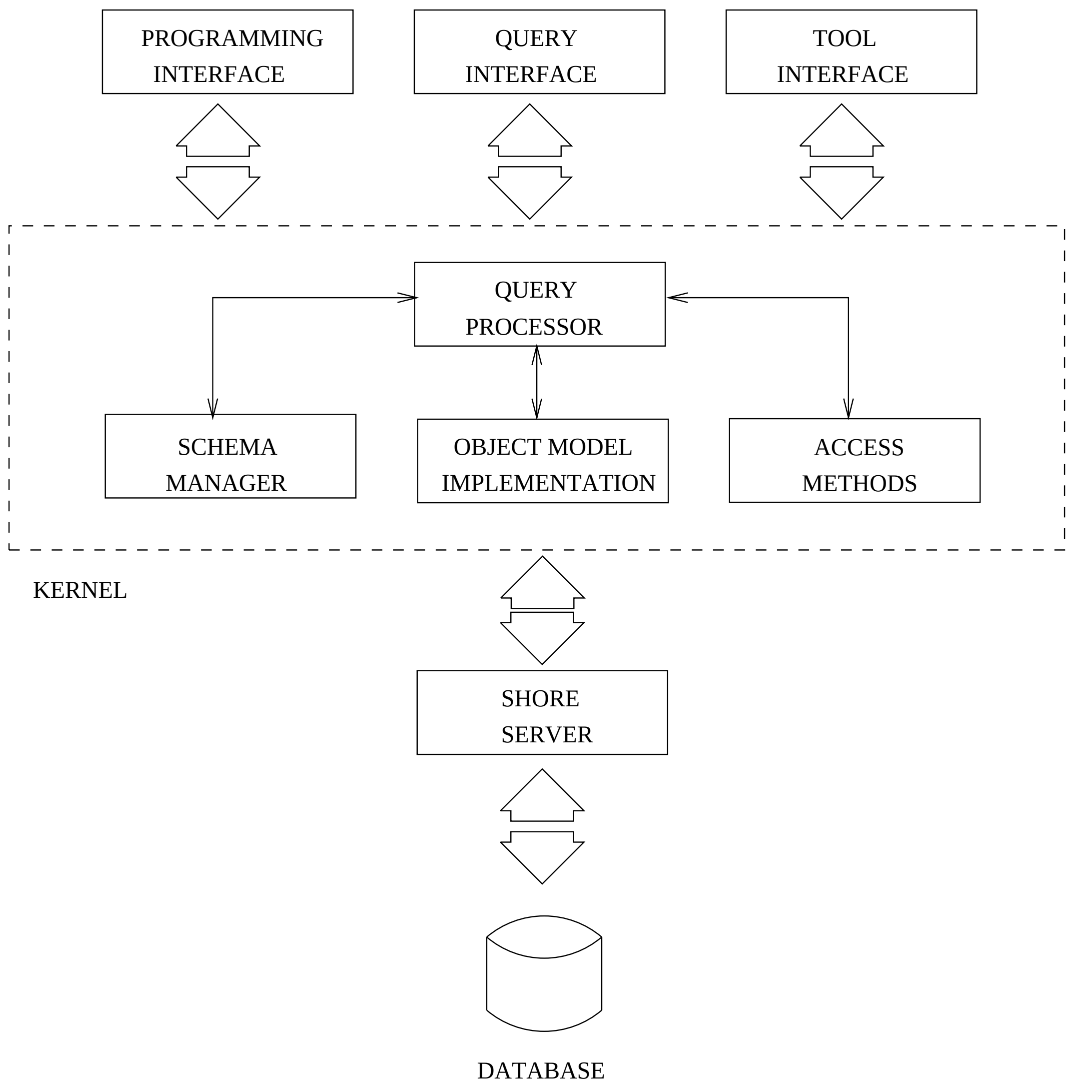


Figure 7.1: DIAS System Architecture

Manager and presents an interface targeted at specific client applications. DIAS uses two such VASs: the Shore Value Added Server (SVAS) and the Shore NFS server. The SVAS is mainly useful for applications using the object oriented database features while the latter is useful for applications relying on the file system features of Shore.

DIAS Kernel

The DIAS kernel implements the object model described in Chapter 6 and uses the Shore server for supporting both object based and file based applications. It contains the access methods used in query processing. It also includes the query processor and the schema manager. The query processor and the schema manager are described in detail in Chapter 10.

DIAS Interfaces

The DIAS system provides a variety of interfaces to support ad-hoc querying, legacy CAD tools and design of new CAD tools. These interfaces are called the *Query Interface*, the *Tool Interface*, and the *Programming Interface*, respectively.

The Query Interface is used to retrieve specific data from the DIAS database. The Query Interface serves as a front-end to the Query Processing System.

The Programming Interface forms a library of routines which abstracts away many of Shore's implementation dependent features to make it easier to write application programs. For example, the Programming Interface can be used to write applications that convert data from specific file formats to DIAS.

Existing file based tools can interact with the data using the Tool Interface. This does not require any changes being made to the tool. For supporting this feature, the DIAS system uses the SHORE NFS value added server to provide the tool-specific view of the objects. This form of access is specially of use for migrating legacy data to the object store.

Chapter 9 covers each of these interfaces in greater detail.

7.2 Object Design

As mentioned in Chapter 5, the analysis stage provides the framework for the implementation of the system and the system design phase decides the overall architecture of the system. The object design phase determines the full definitions of the classes and associations used in the implementation and the algorithms used for implementing the operations. The object model described earlier in Chapter 6 during the analysis phase forms the basis for this stage. We now look at the various steps in the Object Design phase.

7.2.1 Obtaining operations on classes

The operations of classes define the behaviour of objects belonging to that class. The operations are used to retrieve the state and to modify the state of the object. The basic operations are used for object creation and deletion. Others manipulate the state of an object by changing the attribute values or by retrieving the data members. The operations also *encapsulate* the relationships by hiding the implementation while providing an interface to the objects of other classes.

7.2.2 Designing algorithms to implement operations

In many cases, the algorithms for the implementation of operations of a class are evident from the description of the properties (attributes and relationships) of the class. In some other cases, particular algorithms may be chosen for efficiency. Data structures appropriate to the algorithms are used. Frequently, new internal classes and operations are defined for delegation of tasks.

7.2.3 Design of associations

Associations provide access paths between objects. Most associations are inherently bidirectional, but if some associations are only traversed in one direction, then the implementation is considerably simplified by providing for only one - way traversal. Unidirectional associations are usually implemented by pointers, persistent pointers in our case. However in case of two - way associations, there are different ways of implementing them:

- The association is implemented as an one way association. This allows the associations to be traversed in one direction. When the traversal is in the opposite direction, a search is performed. This implementation is useful when the traversal in the forward direction is much more frequent than in the backward direction.
- The association is implemented as two one way associations. The two associated objects have object references to each other as attributes. This implementation has the disadvantage that to maintain consistency and avoid dangling references in case of an update, both the references must be changed immediately. However, the speed of access is the same in both directions.
- An *association object* is used. It is a table of pairs of object references to the associated objects stored in a single association object. The number of entries in case of a m x n relationship is $max(m, n)$.

7.3 Additional classes

The classes in the object model are the abstractions of the major entities in the application domain. However, there is need for more classes for the implementation. We next discuss some of the other classes introduced mainly to support database functionality.

Transaction class

We define a transaction class to serve as a wrapper over the macros and functions of Shore related to transactions. All access, creation, modification and deletion of persistent objects must be done within a transaction. The class has the following methods: begin, commit, abort, and checkpoint. The begin operation creates a transaction. Transactions must be explicitly created and started once a database is opened. The commit operation causes all persistent objects created or modified during a transaction to become accessible to other transactions waiting to access the data. The transaction instance is deleted, and by default all locks are released. The abort transaction operation causes the transaction instance to be deleted and the state of the database to be returned to the state it was in prior to the beginning of the transaction. This is done using the *logging* mechanism. In case of a system crash, all committed transactions need to be redone. Frequently, *checkpoints* are taken when the effects of all data writes of committed transactions are written to the disk. Hence, all transactions that have their commit entries in the log before a checkpoint entry need not have their write operations redone in case of a crash. The checkpoint operation commits all modified objects to the database and retains all locks held by the transaction. The transaction instance is not deleted. The checkpoint method forces a commit and continues with the previous transaction by *chaining* the transaction, that is, a new transaction is started while retaining the locks acquired by the earlier transaction. Transactions can have degrees of isolation [GrRe93] 2 and 3, with degree 3 granting the full serializability of transactions.

Database class

A database provides storage for persistent objects of a given set of types. The database consists of instances of types defined in the *schema*. Each database is an instance of the type *Database*. It supports the following operations: opening and closing of the database, and the lookup of a named object.

In our domain, the major databases are the *Design database* and the *Library Cell database*. Each Library Cell database contains a collection of cells, based on the same technology. The Design Cell database consists of collections of designs which are composed of other designs and Library Cells. The designs contain cross-references to the library cells that are found in the Library Cell database.

To facilitate navigation, certain named "root" objects are provided for each database. Each database is stored under a separate directory in the Shore file system. The root objects are implemented as cross-references to pools containing the anonymous objects belonging to a database. After a database is opened, a transaction is then started. A lookup of a root object specified using a name is used to start accessing the top-level object, using which further access is performed through method invocations.

7.4 Further optimization

The basic design model uses the analysis model as the framework for implementation. The analysis model captures the logical information about the system while the design model adds further details to it. During optimization, the model is further refined for efficient implementation. One way is to add redundant associations to minimize access cost. Here a tradeoff is done between the update cost of these redundant associations after every update and the benefit obtained while accessing the object. For example, a signal connects a set of terminals. Given the terminal, to find the signal to which it belongs, reverse pointers are added that connect the terminal to the signal.

7.5 Access methods

Access methods are a key feature of every database system. They help in improving performance by retrieving a subset of the data in the database efficiently. We next discuss the various access methods that are part of the DIAS system.

The B^+ tree index

The B^+ tree index is used to retrieve objects based on the value of an attribute. Most of the design entities have unique names and hence the name forms the primary key. Thus, B^+ tree indexes with names as keys can be maintained. Methods are provided to insert, search and delete the members of the index.

Indexes on collections: A design object has a collection of signals and io-nets which provide electrical connectivity. There are also collections of devices and library cells which make up this design. Each of these collections can be mapped into a set or list class. For ease of access, a B^+ tree index is also maintained for each of these collections. Thus, each design object has B^+ tree indexes on devices, instances, cells and signals, with the name as the key. The indexes have search, insert, and delete methods.

Indexing techniques specific to object oriented databases

As seen above, the B^+ tree indexes are used in conventional DBMSs to expedite the evaluation of queries. Since an object oriented data model has many differences with respect to the relational model, suitable indexing techniques are necessary for supporting querying. Some of these aspects are:

- Nested predicates: An attribute of an object can be another object, which in turn can have other objects and so on till a simple value because of the property of aggregation. Due to such a nested structure, most object oriented query languages allow objects to be accessed by predicates on both nested and non-nested attributes of objects.

- Inheritance: The scope of a query may apply to a class or to the inheritance hierarchy rooted at that class.

Thus, querying in object oriented databases involve queries on a hierarchy in addition to queries on simple attributes of a class. In addition to the aggregation and inheritance hierarchies described in the above paragraph, relationships among classes not involved in aggregations, but just associations also lead to a *path traversal* with the several objects along the path having links connecting each other. In fact, such paths are a more general case of aggregation hierarchies, since an aggregation is also a special kind of association.

In object oriented data models, relationships among entities are supported through object references. The function that joins provide in the relational model to support relationships is provided more naturally by "path expressions" (implicit joins). A query can be conveniently represented by a "query graph". The strategy used to traverse the query graph can be "forward traversal" – the first class visited is the target class of the query, and "reverse traversal" – the traversal of the query graph begins at the leaves and proceeds bottom-up along the graph.

The efficient processing of queries requiring the traversal of aggregation and inheritance hierarchies is facilitated by suitable access methods. We next discuss the various index organizations for aggregation, and inheritance hierarchies and for path traversals of associations.

7.6 Index Organizations for Aggregation Hierarchies

In this section, we first present some preliminary definitions as defined in the literature. These definitions introduce certain terms that are used later in the context of different index organizations. Then, index organizations for evaluations of queries along aggregation and inheritance hierarchies are discussed.

Definitions

Given an aggregation graph H, a path P is defined as $C_1.A_1.A_2, \ldots, A_n (n \geq 1)$ where:

- C_1 is a class in H;
- A_1 is an attribute of class C_1;
- A_i is an attribute of a class C_i in H, such that C_i is the domain of attribute A_{i-1} of class C_{i-1}, $1 < i \leq n$;

with

- $\text{len}(P) = n$ denoting the length of the path
- $\text{class}(P) = C_1 \bigcup \{C_i | C_i$ is domain of attribute A_{i-1} of class C_{i-1}, with $1 < i \leq n\}$ denotes the set of classes along the path
- $\text{dom}(P)$ denotes the class of attribute A_n of class C_n.

A path is simply a branch in a given aggregation path. A *path instantiation* is a sequence of objects found by instantiating a path (Figure 7.2 shows one such path instantiation). Path instantiations may be *complete* or *partial* . Complete instantiations start with an instance belonging to the first class of the path, contain an instance for each

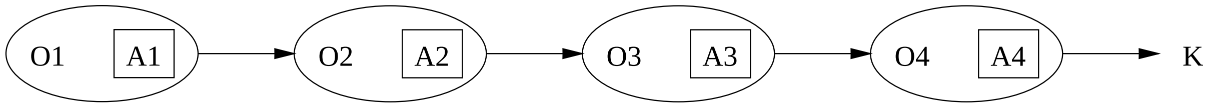


Figure 7.2: An example of a path instantiation

class found in the path, and end with an instance of the last class of the path. Partial instantiations are of two types: *left partial* and *right partial*. In left partial instantiations, the first component of the path instantiation is not an instance of the first class of the path, but rather an instance of a class following the first class along the path, whereas a right partial instantiations ends with an object which is not an instance of the class that is the domain of the path. That is, a right partial instantiation contains a null value for the attribute referenced in the path. Path instantiations can also be both left and right partial.

The queries targeted on aggregation hierarchies require retrieval of the objects that form the path given the root node value or the leaf value, that is the forward traversal and reverse traversal respectively. We next discuss indexes used for efficiently solving queries on aggregation hierarchies.

7.6.1 The Nested index

The nested index [Ber94a, Ber94b] provides a direct association between an object of the class at the end of a path and the corresponding instances of the class at the beginning of the path. The nested index is intended to facilitate reverse traversals, but can be used for forward traversals also. It uses the leaf node of an aggregation hierarchy as the key value which is mapped to objects at the beginning of all path instantiations that end in the key value.

The structure of the nested index consists of a B^+ tree with modifications to the leaf node. The leaf node consists of a set of records. Each record has the following fields: record length, key length, key value, the number of OIDs (object references), and the list of OIDs to the root nodes of path instantiations. Partial instantiations consist of incomplete path instantiations. Right partial instantiations are those that do not end in the leaf node. The nested index stores right partial instantiations by storing a *NULL* as the key value and maps to the root node of all right partial instantiations. Figure 7.3 shows a record in the leaf node of a nested index.

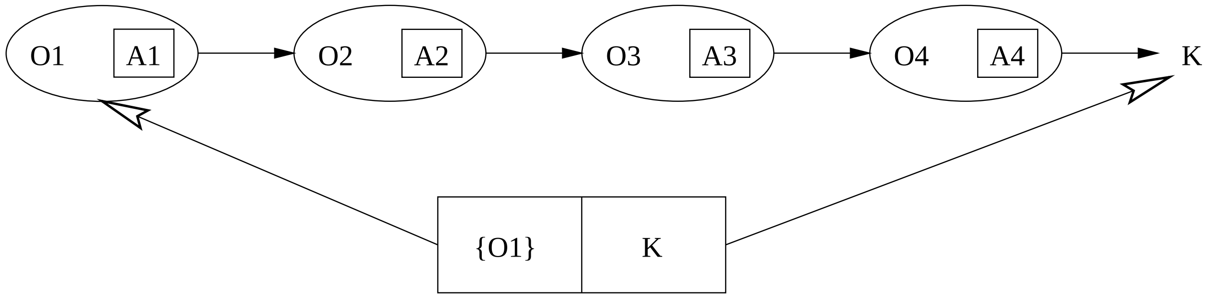


Figure 7.3: Nested index

Operations

Insertions and deletions to the index consist of a pair $< Key, Oi >$ where Key is the leaf node of the path and Oi, the OID of the root node. Insertions are handled by updating the corresponding record by including the OID in case the leaf value already exists. In case of a completely new path instantiation, a new record is created and inserted in the appropriate place. Deletions of key values are similarly handled by deleting the OID in the list or the record itself if no path instantiations exist ending in the key value.

Retrieval under this organization is fast as it requires only one index lookup. The major problem of this indexing technique is that updates on objects of the path require access to several objects in order to determine the index entries to be updated. Update operations require both forward and reverse traversals of objects. Forward traversals are required to determine not only the original value of the indexed attribute but also that for the modified object. Reverse traversal is required to determine the instances at the beginning of the path. The OIDs of those entries will be removed to the entry associated with the original key value and added to that determined by the forward traversal. Note that reverse traversal is very expensive when there are no reverse references among objects. Thus, the nested index is suitable where data updates are rare.

7.6.2 The Path index

A disadvantage of the nested index is that it provides a direct lookup of objects only at the root of the path instantiations and not to all the intermediate nodes of the path. The path index [Ber94a, Ber94b] remedies this by providing an association between the object at the end of a path and all path instantiations that end with that object. Thus, unlike the nested index, the path index can be used to solve nested predicates against all classes along the path.

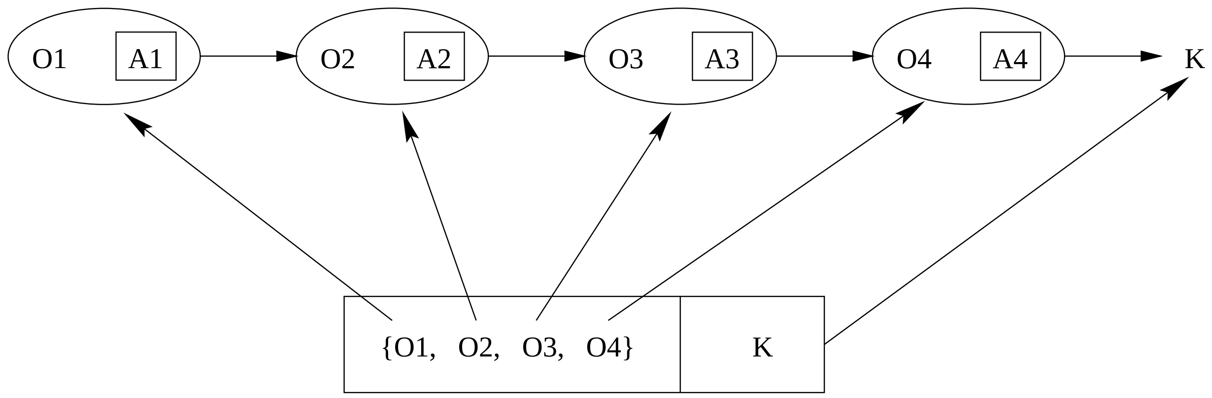


Figure 7.4: Path Index

The Path Index can be implemented using a B^+ tree as in the case of a nested index with modifications being made to the leaf node. Each leaf node record then consists of the following fields: the record length, the key length, key value, the number of path instantiations and the list of path instantiations. Both left and right partial instantiations can be stored in the index. Figure 7.4 shows a record in the leaf node of a path index.

Operations

Insertions are handled by adding the OIDs that constitute the path instantiation to be inserted and then adding them to the appropriate leaf record. If a path instantiation ending in that key value does not exist, then a new leaf record is created. Deletions are similar with the appropriate path instantiation being deleted from the leaf node. Updates to the path index also require two forward traversals to be made. However, unlike the nested index, no reverse traversals are required as the index itself can be used to obtain the whole path instantiation. Thus, updates are less costly as compared to nested index. Also, the path index can be used even when reverse references are not maintained. The main disadvantage of the path index is that leaf nodes contain more information about every path instantiation than in the nested index thus reducing the number of records in the leaf node. Thus, it may be necessary to scan more leaf nodes than in the nested index for retrieval of the same information. Thus, updates are costlier in the nested index while retrievals are costlier in the path index.

7.7 Index Organizations for Association Paths

The concept of path expressions can also be applied to a series of associations among classes. However, each succeeding object is not an aggregate of the preceding object. The nested and path indexes can also be used for such association paths with no change required to the storage structure.

7.8 Index Organizations for Inheritance Hierarchies

Inheritance or specialisation hierarchies are an integral part of object oriented systems. Inheritance based queries may be against a particular class in the hierarchy or against all the classes in the hierarchy, i.e., the hierarchy tree itself. A good query processing mechanism should be able to address both varieties of queries efficiently.

The schema of an object oriented database consists of a forest of trees, with some of the trees being inheritance hierarchies. Each object is in exactly one of the classes in this hierarchy. This partitions the set of objects in the database into several blocks, one for each class. The block corresponding to a single class is called its *extent*. The union of the extent of a class and all its descendants in that hierarchy is called the *full extent* of the class. Thus, the scope of a query on an inheritance hierarchy can be the extent or the full extent of the class.

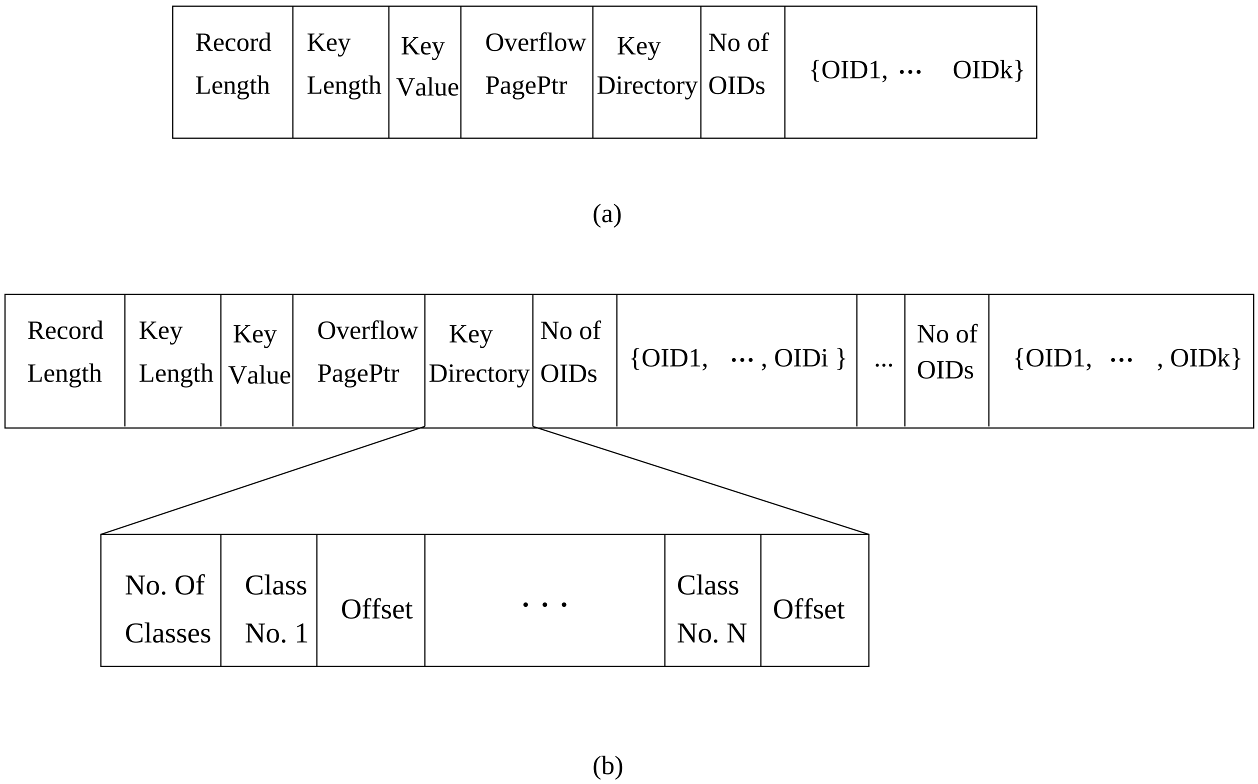


Figure 7.5: Leaf Node Records in Single Class and Class Hierarchy indexes

To support queries on both the extent and full extent of a class, a simple way would be to maintain a B^+ tree for each class of the hierarchy on the key attribute disregarding the fact that the classes are part of a hierarchy. This is called *Single Class* (SC) indexing. The structure of the leaf node record of a Single Class index is as shown in Figure 7.5 (a). The use of such a scheme has linear space usage in the number of classes in the hierarchy, and good update performance. However, when querying the full extent of a class, each additional class suffers from the overhead of retrieval of non-leaf nodes.

7.8.1 The CH tree

The Class-Hierarchy (CH) tree [KKD89] uses the alternative approach of maintaining one index for the entire inheritance hierarchy. The CH tree is also a modification of the B$^+$ tree. The internal nodes of the CH tree are similar to the B$^+$ tree. The leaf node is however modified and the structure is as shown in Figure 7.5 (b).

An important component of the leaf node record is the *key directory*. The key directory contains the number of classes in the inheritance hierarchy that contain objects with the key value as the indexed attribute. Each class in the indexed class hierarchy has an identifier called the *class identifier* (cid). The key directory also contains pairs of the form $< cid, offset >$ for all the classes in the hierarchy. The offset denotes the offset in the record from where the list of OIDs belonging to that class (represented by cid) is stored. Thus, queries directed at a particular class in the hierarchy are answered by looking up the cid corresponding to that class and then retrieving the OIDs using the offset for that cid.

Operations

Insertions are handled by inserting the new OID in the corresponding list of OIDs in the relevant record. If the new OID is the first one for that class, a new list is created and appropriate entries added to the key directory. Additional pages are maintained to accommodate overflow of the record. Deletions are done likewise by deleting the OID from the corresponding list. Deletions of all instances of a particular class is also possible because all the OIDs of a class are grouped together in the record. Merging and splitting of pages in case of underflow and overflow respectively are handled as in the case of B$^+$ trees.

Retrieval of OIDs satisfying the full extent of a class (root class and all the subclasses of the hierarchy) follows the same procedure as for retrieval of OIDs of the extent of a class. Performance studies based on a cost model [KKD89] have shown that the CH tree performs better than single class indexes maintained for the same hierarchy if the number of classes in the hierarchy is atleast two. For the storage cost in terms of disk space used, conclusive results could not be obtained. For queries directed at a particular class in the hierarchy, the index page containing the list of OIDs for that class also has the OIDs for all the classes in the hierarchy, which is an overhead. Hence, in general, CH trees perform better for queries targeted at the full extent of the root class rather than those targeted at the extent of a particular class.

The disadvantage of both the above configurations is that in case of intermediate situations, (e.g., the hierarchy size is 15, number of classes in the scope of the query being 5), both CH and SC perform poorly, because, the SC has too much overhead in terms of retrieval of non-leaf nodes and CH needs to filter too much data. One simple way of getting around this problem is to maintain an SC index on each class and maintain a CH index for the full extent of each class in the inheritance hierarchy. Though this has optimal retrieval time, the cost in terms of storage and updates make such an organization prohibitive. We next discuss another index organization that has average performance better than the ones discussed so far.

7.8.2 The hcC tree

The hcC tree [SrSe94] is based on the B^+ tree and attempts to efficiently process both single-class queries and hierarchical queries. The organisation of the non-leaf nodes is similar to the B^+ tree with a minor modification in that the non-leaf nodes contain an additional bitmap that gives advance information to prevent an unnecessary traversal of the tree till the leaf.

In addition to the leaf node of the tree, another node called the *OID node* is introduced which stores the actual OIDs. The OID nodes are organized in two kinds of chains, the *class chain* and the *hierarchy chain*. Each record of the class chain OID node has key value and OIDs belonging to a particular class in the hierarchy while each record of the hierarchy chain OID node has key value and OIDs of all the classes in the hierarchy (Figure 7.6 (a) shows a class chain OID node record and Figure 7.6 (b) shows a hierarchy chain OID node record). Each hcC tree consists of n class chains, one for each class in the hierarchy and one hierarchy chain, where n is the number of classes in the hierarchy.

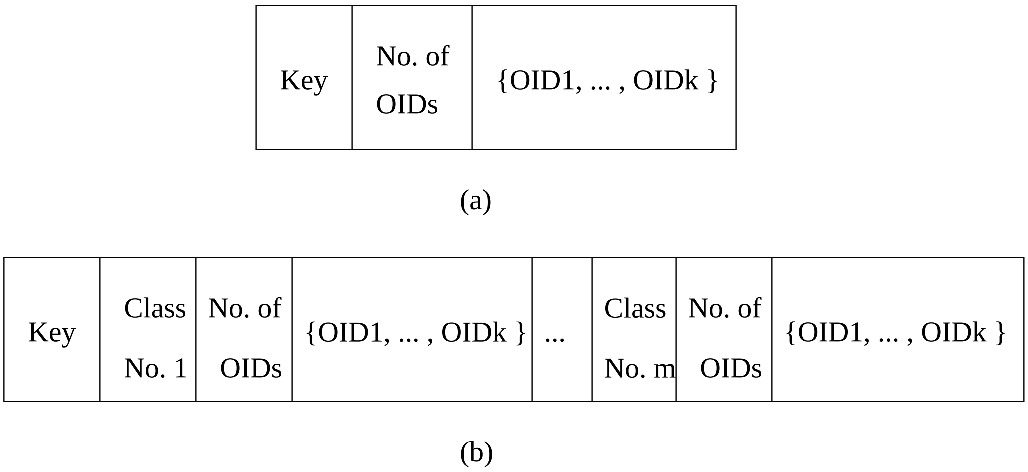


Figure 7.6: hcC Tree

All the OID nodes belonging to a particular chain are chained together. Thus, there are $n + 1$ chains in the tree.

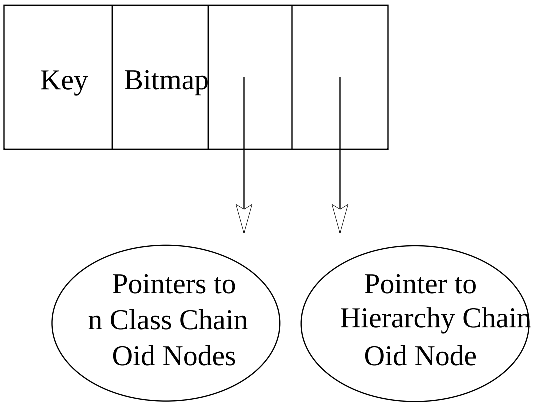


Figure 7.7: Leaf node record of a hcC Tree

The structure of a leaf node record is as shown in Figure 7.7. The leaf node consists of the key value, a bitmap, and a set of $n+1$ pointers. Of these, n pointers point to the OID nodes of the class chains whereas the last pointer points to an OID node of the hierarchy chain. In case there are no instances corresponding to particular classes, the bitmap is not set for the corresponding *class identifier*, cid, where the cid can take values from 1 to n.

Operations in the hcC tree

Insertions into the hcC tree involve inserting the value at *both* the class-chain OID node and the hierarchy-chain OID node. Deletions are similar with OIDs being deleted from both the OID nodes. The advantage of the hcC tree is that it facilitates the efficient retrieval of OIDs satisfying both the single-class as well as hierarchical queries. In case of range queries that require a retrieval for a subset of the classes in the hierarchy, the individual class chains are used following the links connecting one OID node to another. For range queries targeted at the entire hierarchy, the hierarchy chain is used following the same procedure.

The performance results obtained in [SrSe94] indicate that the hcC tree provides an efficient mechanism for point and range queries for single class as well as hierarchical queries. A disadvantage though is the increase in storage space because of duplicate OIDs being maintained at the OID nodes of both the class chain and the hierarchy chain.

7.9 Unified Index Organizations for Composite Hierarchies

The index mechanisms considered so far deal with either aggregation or inheritance hierarchies in isolation. However, frequently classes participate in both aggregation and inheritance hierarchies. For example, consider the path $P = C_1.A_1.A_2 \ldots A_n$, traversing n classes $C_1 C_2 \ldots C_n$, with the class C_1 forming the root of an inheritance hierarchy, H. The inheritance hierarchy, H, consists of classes, C_1, the root class, and $H_2, H_3 \ldots H_n$, the subclasses of the hierarchy. We call such a hierarchy a *composite hierarchy*.

Now, consider a query requiring reverse traversal consisting of retrieving all instances A_1 whose paths P end in the instance A_n. The query was earlier solved by maintaining a nested index with key being A_n. Now, the problem involves scanning the full extent of the hierarchy, H. This can no longer be solved using the nested index since this involves the full extent of the class C_1 as opposed to the extent of the class earlier. Similarly, retrievals of instances in the middle of the path cannot be solved using the path index considered earlier for the same reason. Thus, the indexes used for aggregation hierarchies implicitly assume that the target of the query involves the extent of the class alone and *not* the full extent of the class.

To efficiently solve the above problem, we now propose *Unified Index Organizations* that can solve queries on both the extent and the full extent of a class efficiently.

Since A_1 is an attribute of C_1, all the classes in H, $H_2, H_3 \ldots H_n$, inherit the attribute A_1. As a result, the path, $P = C_1.A_1.A_2 \ldots A_n$, is also inherited by all the classes in H, leading to several *inherited paths*, such as $P_2 = H_2.A_1.A_2 \ldots A_n$, $P_3 = H_3.A_1.A_2 \ldots A_n$, and so on till $P_n = H_n.A_1.A_2 \ldots A_n$. Thus, a single path has now become N inherited paths.

Now, the problem is reduced to retrieving objects along these inherited paths. However, the fact remains that the root of these paths are part of an inheritance hierarchy. Queries that are targeted at the extent of a single class involve retrieval of objects corresponding to path instantiations of *one* of these inherited paths. Those targeted at the full extent of the class involve retrieval of objects corresponding to path instantiations of *all* the inherited paths. We next discuss unified configurations that can be used to solve these queries.

	SC Index	CH index	hcC index
Nested Index	SCN	CHN	hcCN
Path Index	SCP	CHP	hcCP

Figure 7.8: Unified index configurations

Figure 7.8 shows the various unified configurations that can be used to solve queries on a composite hierarchy. The SCN index consists of a set of nested indexes $SCN_i, i = 1, \ldots, n$, one for each of the inherited paths, H_i. The key is A_n mapping to OIDs of class C_1 and H_i, $i = 2, \ldots, n$. However, this configuration suffers from the drawbacks of SC index for full extent queries. Similarly, the SCP index consists of a set of path indexes $SCP_i, i = 1, \ldots, n$, one for each of the inherited paths. This too suffers from the same drawbacks of SC index for full extent queries.

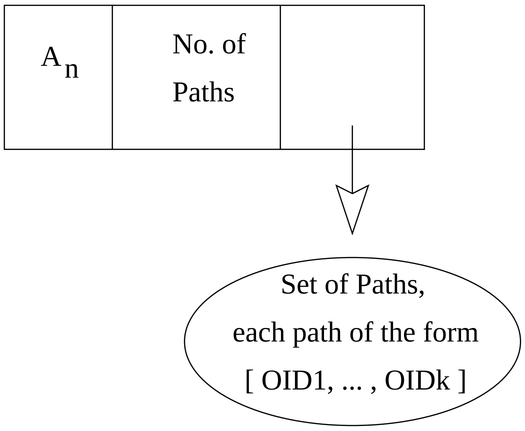


Figure 7.9: Class chain OID node record of a hcCP Tree

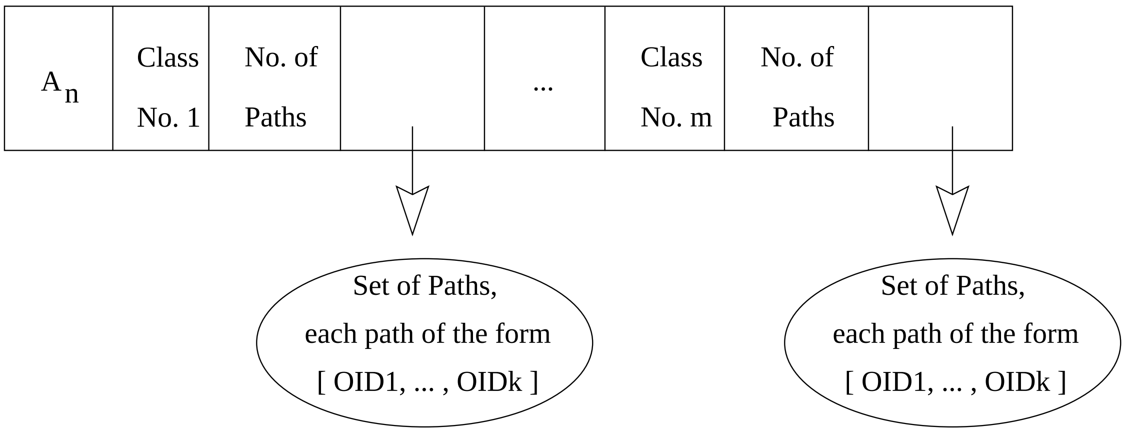


Figure 7.10: Hierarchy chain OID node record of a hcCP Tree

The CHN index consists of a single CH tree with the only modification being that the key is now A_n instead of an attribute of the class C_1. The CHP index, in addition to the key being A_n, has an additional modification in the record of a leaf node with the OID value being replaced by a list of OIDs corresponding to a path instantiation.

The modification required to the hcC tree to transform it into a hcCN index is that the key needs to be changed to A_n instead of an attribute of C_1. The hcCP index, in addition to the above modification, requires the OID in the OID node to be replaced by a list of OIDs representing a path. Figure 7.9 shows an OID node record of a class chain while Figure 7.10 shows an OID node of a hierarchy chain in a hcCP tree.

The performances of these unified indexes are expected to be similar to their corresponding inheritance indexes. That is, the CHN is expected to perform similar to the CH tree and the hcCN is expected to perform similar to the hcC tree as there are no major modifications to the storage structure. Moreover, the comparative performances of CHP versus hcCP and that of CHN versus hcCN would be similar.

7.10 Efficient execution of queries

We next discuss some sample queries and see how they are solved efficiently using the index mechanisms discussed earlier.

- **Simple Value based queries**

  Handling simple value based queries entails extraction of an object based on a simple attribute value. An example query is

  List the names of all signals used in the design FPU_206N.

  In answering this query, it is sufficient to locate the design instance FPU_206N of a floating point unit design. To speed up execution of the above query, a B^+ tree index is maintained on the design objects for attribute *DesignName*. Once the reference to the design object is obtained, signals used in the design are listed.

- **Inheritance based queries**

  Inheritance based queries require the traversal of inheritance hierarchies and identification of all instances along the inheritance path that satisfy the query predicate. An example query is

  List all the Nets which have fanout greater than 17.

The *Net* class specialises into *IO-Net* class and *Signal* class and hence all the objects of these classes which have the required fanout need to be located. A *hcC* tree maintained on the inheritance hierarchy is used to identify all objects in the hierarchy satisfying the given predicate.

- **Aggregation based queries**

  Aggregation based queries require either forward or reverse traversals of the aggregation hierarchy. An example of a forward traversal aggregation based query is

  List the source to load RC delays due to terminal RPO.IN5.OUT_3 in the signal SIM5AND_IN of the design FPU_206N.

  Answering this query consists in locating the design FPU_206N in the database, navigating the object reference to the particular signal SIM5AND_IN, picking the particular source, RPO.IN5.OUT_3, and then listing the delays contained in the *Timing* objects for all the loads. An example of a reverse traversal based aggregation query is

  List all signals in design FPU_206N which suffer slew greater than 5 ns and their sources.

  This query needs to access objects which form neither the root nor the leaf of the hierarchy. This necessitates use of the path index on the path

  $$Design.Signal.Source.Transition.Value$$

  The query is processed by first identifying the *Timing* objects satisfying the constraint provided in the query. Once this is done, the path index is used to get the respective signal names. The source terminals can also be obtained with the *same* lookup since the path index indexes all objects in an aggregation path.

  However, for queries targeted at the end points of a path, the nested index is more appropriate since it eliminates the overhead of maintaining references to all instances along a path. An example of such a query is

Give all nets with an effective capacitance greater than 10 pF.

A nested index keyed on capacitance values establishes direct connections with the respective net objects. Thus, nets satisfying the above query for all designs in the database can be retrieved efficiently.

- **Queries involving both aggregation and inheritance**

  If the previous query were qualified as follows:

  Give all io-nets with an effective capacitance greater than 10pF.

  the query can be solved by maintaining another nested index with keys as capacitances and values as OIDs of io-net objects. However, if the query were changed to retrieve signals instead of io-nets, then this would necessitate having another nested index.

  An efficient way of solving all the three queries is to use the Unified Index organisation. Choosing a hcC tree and a nested index as our combination, we maintain a hcCN index with key being capacitance values and the OIDs pointing to objects of the net hierarchy. This ensures that all the above queries are answered efficiently with just *one* index being maintained.

This concludes the Object Design phase of the development methodology. The final stage is the implementation which is considered in the next chapter.

Chapter 8

DIAS Kernel Implementation

The implementation of the DIAS system is discussed in this chapter. As seen in Chapter 7, the Shore server is the lower most layer of the system. The DIAS kernel is implemented on top of Shore. The object model discussed earlier in Chapter 6 is implemented using Shore's support for persistent objects. The access methods described in Chapter 7 are then implemented.

8.1 Object Model implementation

The first step in the implementation of a database system is the definition of the *schema*. The Shore Data Language (SDL) is used for this purpose. The major steps involved are:

- Identify the "interface" declarations.
- Organize groups of related classes into modules.
- Identify data that need to be exported to other modules.
- Declare operations of the interfaces.

Each class declared for storing persistent data has a corresponding interface declaration. The classes that were identified in the object model are now converted into their corresponding *interfaces*. Associations among classes in the object model are implemented as either *local references* or *remote references*. Local references are used for intra-object associations (such as between the elements of a linked list attribute of an object) while inter-object associations make use of remote references. Each interface is composed of declarations of *properties* and *operations.* The properties model the state of the object while the operations model its behaviour. The properties constitute *attribute* declarations and *relationship, ref* declarations which are used to represent the data members and the associations. The operations of the interface constitute the return type, the operation name, and a set of arguments. The arguments may be of *in, out* or *inout* type. The *in* arguments are not modified inside the operation while the *out* arguments are used for data that are changed as a result of the operation. The *inout* operations may be used inplace of either the *in* or the *out* operation. An argument may also be *transient* (not persistent), in which case it is preceded by the keyword *external*. Similarly, the attributes and operations defined earlier in the model are converted into their corresponding SDL declarations. Figure 8.1 shows a sample SDL declaration for the *Cell* interface. The *Cell* has a set of ports, belongs to a particular *type*, and is identified by the *CellName*. Appendix A has the SDL declarations for a few important interfaces.

A logical group of related interfaces form a *module*. Interfaces within the same module share data variables and constant declarations. Sometimes, declarations of one module may need to be accessed in some other module. This is done by *exporting* those identifiers in the module in which they are originally declared and then *importing* them in the other module which needs to access them.

The next step after declaring the interfaces in SDL is to generate the appropriate language binding, in our case C++. The SDL compiler creates a persistent module object in the Shore database corresponding to each module declaration. Each module has a collection of type objects, one for each SDL interface declaration. Each type object stores information about the core and heap data requirements for an object of that type. During runtime object access, this type object is checked to ascertain whether the object is of that valid type and the operation invoked is permitted. In addition to creating these module and type objects, the SDL compiler generates the C++ language binding

```
// This class holds all information that are associated with
// both Library Cell and Design.
interface Cell
{
private:
 attribute string    CellName;              // name of the cell
 attribute set<Port> Ports;                 // ports of the cell
 attribute CellType  Type;                  // type of the cell
public:
 void       Construct(in lref<char> NewName);// initializing function
 void       SetName(in lref<char> NewName);  // set the cell name
 lref<char> GetName();                       // get the cell name
 short      NumberPorts() ;                  // get port count
 ref<Port>  GetPort(in short PortIndex);     // get a specific port
 CellType   GetType() ;                      // get cell type
 CellType   SetType(in CellType NewType) ;   // set cell type
 ref<Delay> FindPath(in ref<Port> Input,
            in ref<Port> Output);            // get delay between ports
 ref<Port>  GetPort(in lref<char> Name) ;    // get port with given name
 void       AddPort(in ref<Port> NewPort);   // add a new port
};
```

Figure 8.1: Sample SDL declaration

in the form of header files containing C++ *class declarations* for the corresponding SDL declarations.

After generating the header files, we next implement the definitions of the methods corresponding to the *operation declarations* of interfaces declared earlier in SDL. This is similar to definitions of member functions in C++, with some differences in the manipulation of persistent objects, as discussed in the following section. The overall flow of the implementation follows the basic stages as shown in Figure 8.2.

Thus, the SDL declarations correspond to the Data Definition Language (DDL) statements in conventional database systems. The result of compilation of these DDL statements is the *metadata* or “data about data”. The metadata in our case are the type objects created by the SDL compiler.

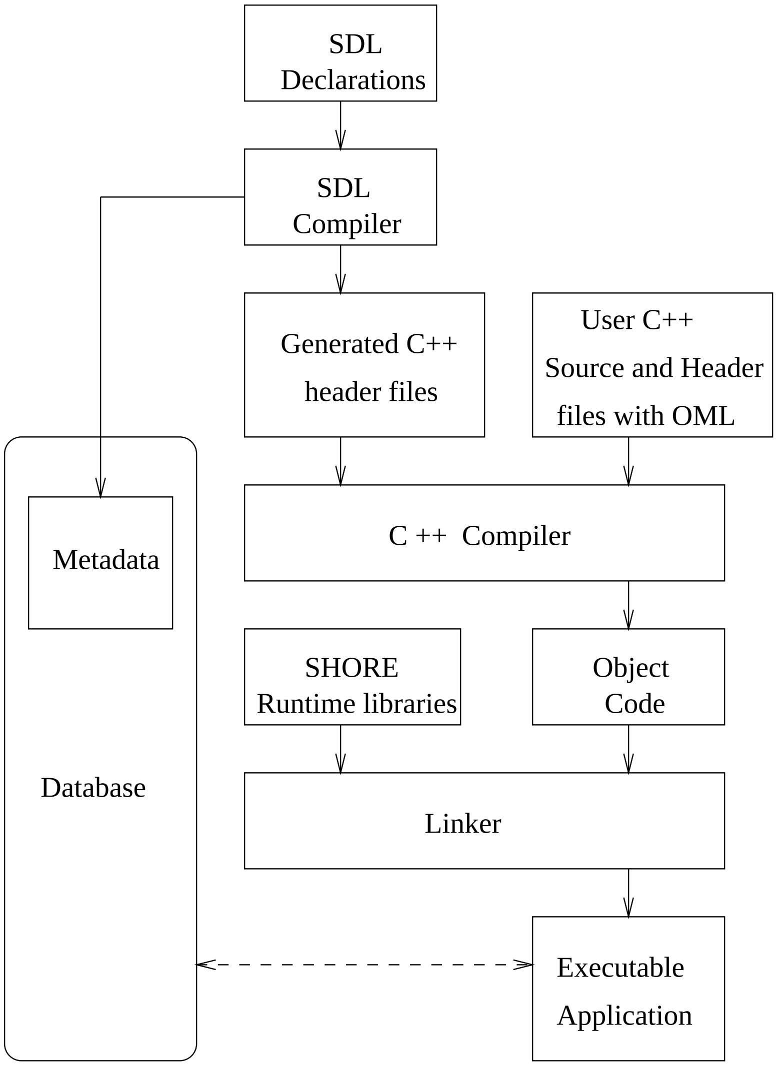


Figure 8.2: Implementation Flow

8.2 Data Manipulation

Data manipulation includes the following: insertion of new information into the database, retrieval of information from the database, modification of data stored in the database and deletion of information from the database. These features are implemented using C++ as the database programming language or the Database Manipulation Language (DML). The C++ to OODBMS *language binding* is based on one principle: The programmer feels that there is one language, not two separate languages with arbitrary boundaries between them, whether one is dealing with the database or with the transient data structures of the program, thus avoiding the oft-quoted problem of "impedance mismatch". This principle has four corollaries [Catt94]:

1. There is a single unified type system across the programming language and the database. Individual instances of these types can be transient or persistent; that is persistence is orthogonal to type.

2. The binding is structured as a set of additions to the base programming language; it does not induce sublanguage-specific constructions that duplicate functionality already present in the base programming language.

3. The binding respects the syntax and semantics of the base programming language into which it is introduced.

4. Expressions in the manipulation language freely compose with the base programming language and vice versa.

In general, the object accessed may have one of the following lifetimes:

- coterminous with procedure
- coterminous with process
- coterminous with database

The first kind consists of instance variables defined locally in a procedure and are not visible outside the procedure. The "coterminous with process" variables are global and static, heap variables which are destroyed at the end of the execution of the program. The "coterminous with database" variables are those whose contents survive the termination of the process that created them and can be accessed by subsequent processes. The first two kinds constitute the transient data and the last the persistent data. Transient data manipulation falls in the domain of all programming languages. The support for persistence calls for adding new functionality in programming languages. We next discuss some of the relevant features.

Object References

Objects may refer to other objects using object references. Object references are instances of the template class Ref<T> for any class T. The dereference operator $\rightarrow$ is used to access members of the persistent object "addressed" by the object reference. Method invocations are through this Ref variable which acquires a *shared lock* on the requested object, and then fetches the object into the object cache after "swizzling the OID" and including an entry (OTE) in the Object Table. Updates to the object are not possible and hence only "const" member functions can be invoked. However, updates can be made by first calling the *update()* member function of class Ref<T> which obtains an exclusive lock on the object. An alternative way is to directly obtain a shared lock through a variable of the class WRef<T>. All these operations must be performed inside a *transaction*.

Object Creation

Objects are created using an overloaded form of the *new* operator. This form also allows the programmer to specify where the newly allocated object is to be clustered. Thus, the new operator can be used to create an object of a class previously declared in SDL. The object may be registered, in which case it is accessible through a fully qualified path in the Shore file system, or anonymous wherein it belongs to a pool and cannot be accessed individually.

Object Deletion

The deletion of previously created objects is done through calls to *Ref< T>::unlink()* or *Ref< T>::destroy()* member functions respectively for registered and anonymous objects.

Operations

Operations are defined as in C++. Operations with transient and persistent objects behave entirely consistent within the operational context defined by standard C++. This includes all overloading, argument passing and resolution, and other compile time rules.

Collection Classes

Collection templates are provided to support the representation of a collection whose elements are of any persistent type. These are of type Set, Bag, etc. The declaration Set<T> declares a set of elements of type T while Bag<T> declares a bag of elements which differ from a set in that it can have duplicates.

Transactions

All access, creation, modification, deletion of persistent objects must be done within a transaction. Transactions are implemented by providing a wrapper to Shore's underlying transaction functions. The class Transaction defines the operations for starting, committing, aborting and checkpointing transactions.

Database

An instance of the *Database* class as described in Chapter 7 is created. Its methods are used to open a database, close a database, and to navigate to the desired object.

Populating the database involves creating persistent objects and invoking appropriate member functions to set the attribute values. Accessing objects involves following inter-object references starting from a distinguished *root* object in the Shore namespace. A database usually consists of collections of persistent objects with each collection containing objects of a particular type. All objects of a particular collection are clustered in the same *pool*. Thus, each database consists of several *anonymous* objects with the root object being a *registered* object.

Database creation begins with creating a new pool in a particular directory of the Shore namespace. Then, anonymous persistent objects which form the database are created in the pool. References to any particular anonymous object in the pool are implemented using cross references. Subsequent database accesses start with either the root object or the cross reference and then follow method invocations.

8.3 Access Methods

The implementation of the access methods for efficient query processing constitutes an important optimization step in efficient data retrieval. We next discuss the details of the implementation of the access methods described earlier in Chapter 7.

8.3.1 B^+ tree

The B^+ tree forms the basic building block for all the access methods that are implemented. The classes that implement the B^+ tree are provided by Shore. Two variants of the B^+ tree are provided: one with unique key values and another that accommodates non-unique key values.

8.3.2 Nested Index

The Nested Index was built using the B^+ tree with modifications to the leaf node. The leaf node consists of several entries, with each record consisting of a key and a persistent reference to a *Query* object. The class *Query* forms the root class that is inherited by every other class in the schema. This helps in having a generic nested index whose OIDs may be dynamically resolved to any particular class in the schema. Figure 8.3 shows the SDL declaration of the main interface for a nested index maintained for an aggregation

```
interface NestedIndex
{
   private:
        attribute index<float,Query> Tree;
                                // the index with key type float
   public:
        void Construct();
                                // the initializer function
        void Insert( in float Key, in ref<Query> QueryRef );
                                // insert a key-value pair
        void Delete( in float Key, in ref<Query> QueryRef );
                                // delete a key-value pair
        short Delete( in float Key );
                                // delete all values for a key
        ref<Query> Search( in float Key );
                                // retrieve the matching value
};
```

Figure 8.3: Nested Index SDL declaration

hierarchy ending with an attribute of type *float*. The methods of this interface and other auxiliary interfaces are then implemented in C++.

8.3.3 Path Index

The Path Index is similar in construction to the Nested Index with modifications to accommodate the whole path instead of a single OID. Each path is a set of pairs of the form $< Class, OID >$, where the Class represents the particular class in the path to which the OID is an object reference.

8.3.4 CH tree

The CH Tree also involves modifications to the B$^+$ tree. The leaf node consists of $< key, CHPage >$ pairs where the CHPage points to a *page* consisting of several *Class Hierarchy Elements* (CHE). Each CHE is constituted of a key and a list of OIDs for each class in the hierarchy identified by its class identifier as shown in Figure 8.4.

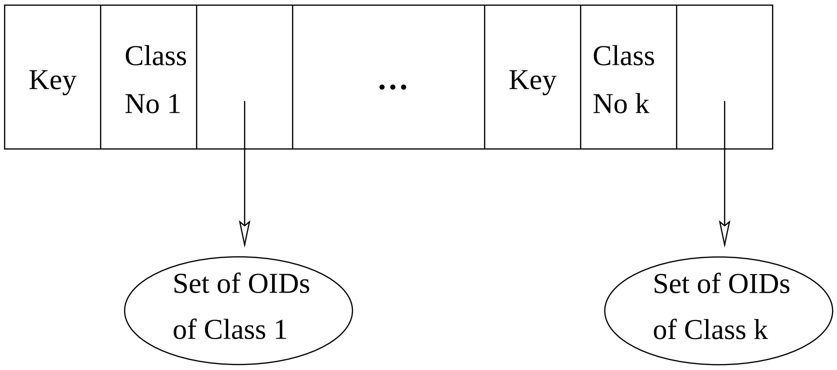


Figure 8.4: Class Hierarchy Element of a CH tree

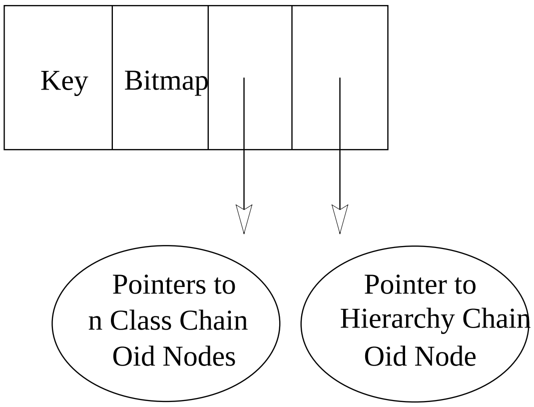


Figure 8.5: A LONE of a hcC tree

8.3.5 hcC tree

The implementation of the hcC tree involves changes to the leaf records of the normal B^+ tree. In addition to the non-leaf and leaf nodes of the B^+ tree, the OID node is introduced. The hierarchy chain OID node consists of CONE (Class OID Node Entry) records and the class chain OID node consists of HONE (Hierarchy OID Node Entry) records. Each CONE is made of a key and a set of OIDs and stores the set of references to objects of a particular class. A HONE, on the other hand, stores references to all the objects of the entire hierarchy corresponding to a key value. The leaf node consists of pairs of the form $< key, LONE >$ where the LONE forms a Leaf Node Entry. The

Key	Set of OIDs

(a)

Key	Class No 1	Set of OIDs of Class 1	...	Key	Class No k	Set of OIDs of Class k

(b)

Figure 8.6: CONE and HONE of a hcC tree

LONE consists of the Bitmap, a set of references to *CONEPages*, and a reference to a *HONEPage* (Figure 8.5). The CONEPage implements the class chain OID node while the HONEPage, the hierarchy chain OID node. The chained list of HONEPages forms the hierarchy chain while the chained list of CONEPages forms a class chain. A CONEPage consists of several CONEs while a HONEPage consists of several HONEs. The structure of a CONE and HONE is shown in Figure 8.6 (a) and Figure 8.6 (b) respectively.

The hcC tree is used for efficient retrieval of objects satisfying both range and point queries. The queries may be targeted at any class in the hierarchy or a set of classes of the hierarchy. Consequently, queries may be classified as SCP (single class point), SCR (single class range), CHP (class hierarchy point) and CHR (class hierarchy range). The methods to implement these operations are based on algorithms given in [SrSe94].

8.4 Other implementation details

The object model implementation consisted of around 50 major classes. This was implemented in around 7000 lines of code in SDL and C++. The access methods and other auxiliary classes totaled approximately 9000 lines of code.

Chapter 9

Interfaces

The main interface requirements of DIAS are support for efficient querying, an interface for tools to input and retrieve data, and a programming interface which helps in the development of additional applications such as other tools. In this chapter, we provide descriptions of each of these interfaces.

9.1 Tool Interface

Most CAD tools are file based with specific input and output file formats. The migration to a database system introduces the following problems: First, the earlier file based applications can no longer be used to retrieve or feed data. Second, the existing data resident in files must be moved to the new system.

One solution is to keep the data permanently in files as it is and use the DBMS mainly to store metadata such as file names in the database. This solution has the obvious disadvantage that the DBMS is not utilised to its full extent. Another solution is to transfer the contents of the objects to files, operate on them and then copy back the contents. This suffers from the drawback of runtime cost in translation to and from files. Yet another solution is to change the applications themselves so that they operate directly on objects instead of files. Though this solution is the most satisfactory in the long run, it suffers from two disadvantages. The existing file based data must be converted to objects and until the new applications are developed, the OODBMS cannot be used.

DIAS addresses this situation by allowing the existing file based tools to be used while completely new tools are built that directly manipulate the database. This is facilitated by providing a tool interface that utilises the file system features of Shore. This can be done in the following two ways.

- Converting to a "text" object:

  Each file is converted into a Shore object with a single attribute of type "text" that stores all the contents of the file. The file based applications can now access this object in the Shore namespace, without any changes made to the application, by specifying the path name corresponding to the Shore file system.

- Using the Shore NFS Value Added Server:

  For applications that cannot be modified even slightly, the Shore NFS VAS can be used. An entire subtree of the Shore namespace is "mounted" on an existing Unix file system. When applications attempt to access files in this portion of the name space, the Unix kernel generates NFS protocol requests that are handled by the Shore NFS value added server.

  In this case, the tool interface is used as follows. The legacy tools and other data are first moved into the Shore namespace. This is done by copying these over to the mounted subtree. Hereafter, existing tools can be used just as before with the only difference being that they are now part of the Shore file system.

9.2 Programming Interface

Some of the advantages of the object oriented paradigm are reusability and extensibility. The programming interface helps in developing new applications that can access the database. It provides a level of abstraction by hiding the implementation details regarding Shore. It is organized as a library of classes. The classes that were implemented from the object model provided the basic functionality to store the persistent data. The programming interface uses these classes as the base and extends their functionalities.

Uses of Programming Interface

- New applications: Application programs that process the stored data can be quickly developed. Newer applications need to be written to support additional functionality. The programming interface fills this need by hiding the details of Shore as much as possible and by keeping the interactions to the stored data transparent to the user.

- Schema Evolution: The schema of the underlying data may require changes with time. For example, a new attribute may be added to a class. The programming interface is used to derive a new class from the parent class and add the new attribute. Then, all new instances that are created are of the derived class. The instances of the parent class are duplicated with instances of the new class and finally, the instances of the parent class are deleted.

- Intertool communication: The legacy data residing in files needs to be moved to the database system. This requires applications that read in the data according to the input format and then load them into the DBMS. Subsequently, the stored data in the database may be fed to analysis tools that again require the data in a specific format. Thus, as long as replacement tools that access the data directly from the database are not available, applications are needed that convert data from the database to files of specific format and vice versa. These can be developed quickly using the programming interface.

Categories of classes

Based on the data that objects of the classes will be storing, the library of classes can be divided into the following categories:

- Classes for base entities

  The base entities that are used repeatedly in the domain are the triplet class that stores a combination of three values, *String* which provides the string functionality, the *Coefficient* class, and the *Transition* class for storing timing related information.

- Classes for design entities

  The basic entities of a design fall in this category. These are the classes for the design entities such as *port*, *design*, *cell*, *library cell*, etc.

- Interconnect Model related classes

  The classes that describe the interconnect model fall under this category. For example, the *Edges* and *Nodes* of an RC tree representing a signal.

- Parametric Info related classes

  The effects of interconnect parasitic have been captured with classes. The notable ones are those for Slew, RC Delay, etc.

- Characterization data related classes

  These classes are the ones for storing and manipulating the pre-supplied characterization data.

Categories of functions

The member functions of these classes are again of the following types:

- Constructor and Destructor functions

  SDL does not provide for constructors and destructors. Therefore, explicit *Construct* and *Destruct* functions are provided. The construct function is invoked once every time a persistent object is created. The destruct function is similarly invoked when an object needs to be destroyed. This is necessary since Shore does not provide implicit means of garbage collection and hence objects need to be explicitly deleted from the database.

- Iterator functions

  Persistent collections of objects such as sets are extensively used in the data model implementation. For traversing through the set in sequence, iterators are provided that retrieve one element at a time.

- Output functions

  The functions that can be used for printing the state of the object fall under this category. The results of a query are displayed using these functions.

- Navigational functions

  These functions provide a means of following associations among the objects in the database. For example, from the design object, we can follow a particular signal, look at its sources and load terminals, and the delay between them.

- Simple retrieval and update functions

  These can be used to obtain the current value of the attributes or to change the value of the attribute.

Index Library

The DIAS system provides a library of specialised access methods as discussed earlier in Chapter 7. The access methods for aggregation hierarchies, namely the path index and the nested index, and those for inheritance hierarchies, the CH tree and hcC tree constitute the index library. These indexes can be created as easily as that of creating an instance of any other class.

Thus, the Programming Interface is useful for retrieving the stored data through an application program and can be used to extend the functionality of DIAS. As one of the first uses of the Programming Interface, tools were developed for bulkloading of parasitic data from legacy text files to and from DIAS. The details are given in [Kris96].

9.3 Query Interface

The query language specification was inspired by OQL [Catt94], and implements a subset of the specifications that deal with retrieval of data. The query language built was however found sufficiently powerful for data manipulation in the parasitics database. The language employs the *SELECT-FROM-WHERE* statement which is similar to the popular SQL.

The query can contain predicates of the form $< PathExpression, operator, value >$. The *operator* is one of the comparison operators ($>, <, >=, <=, ==, !=$). The *value* is one of integer, double or string. The language itself is defined as shown in Figure 9.1.

```
SELECT <PathExpression>
FROM   <TopLevelObject> in <TopLevelClass>
WHERE ( <expression> OP ... OP <expression> );
```

Figure 9.1: Query Language Syntax

The language consists of the *SELECT* operator which performs the object projection operation, which is essentially extracting attributes from the objects. The *FROM* operator is used to define the toplevel class name which represents the type of the root node of the path expression. In DIAS, the Design class is the toplevel class and has been implicitly assumed in all path expressions for the sake of convenience. Thus, it is enough to specify only the design instance for the *FROM* operator. The *WHERE* operator is used to define predicates that the result objects of a query must satisfy. The *WHERE* clause is abstracted as a series of *expressions* separated by the *OP* logical operator (which could be an *AND* or *OR*). Each expression contains predicates and an arbitrary number of them can be linked with the *AND* and *OR* operators. The path expressions in the predicates have the root node implicitly assumed to be the design instance specified in the *FROM* operator clause. The expression clause can also contain other *SELECT-FROM-WHERE* operators thus forming a nesting hierarchy. Thus, the expression may be deep in two orthogonal dimensions – one dependent on the number of predicates, and the other dependent on the number of nesting levels of expressions. An example query is given below and its query language specification is given in Figure 9.2.

QUERY

List the names of nets in the design FpuDesign that have low to high transition delay greater than 20 ns and have greater than 4 terminals associated with them and the estimated prelayout wire capacitance is less than 0.5 pF.

```
SELECT Signals.NetName
FROM   FpuDesign
WHERE ( Signals.Sources.SLTimings.Transitions.LHValue > 20
                AND  Signals.NumTerminals > 4
                AND  Signals.PreWireCap   < 0.5
        );
```

Figure 9.2: An Example Query Specification

The relevant classes for the above query are the *Design* class representing a design entity, the *Signal* class representing signals in a design, the *Source* class for the source terminal, the SLTimings class for representing the source to load timing, and the *Transition* class that stores the values for the various combinations among low, high, and zero transitions. Note that all the predicates contain the toplevel object (in this case, *FpuDesign*), implicitly specified for user convenience. The details of the Query Processing System are covered in the next chapter.

Summary

DIAS provides several interfaces to cater to the wide spectrum of users. To enable existing tools to continue to access the data, it supports a Tool Interface. For retrieving a subset of the stored data, users can ask queries using the query interface. To develop applications that manipulate the database directly, a Programming Interface is provided. Thus, these interfaces support the various functional requirements of the system for data management of VLSI CAD parasitic data.

Chapter 10

The Query Processing System

In this chapter, we discuss the DIAS Query Processing System (Figure 10.1). This system was designed and implemented in collaboration with M. Krishna, a fellow graduate student, and the complete details of the system are described in his thesis [Kris96]. Here we describe the salient features. The query processing system in DIAS is designed to provide efficient data retrieval. This is accomplished by means of a query processor, a schema manager, and the use of the specialized access methods disussed earlier in Chapter 7.

In this chapter, we describe the query processing architecture and its relation to other components of the DIAS system. Each of the components are described in detail in subsequent sections.

The query processor accepts a query specified in a declarative query language as mentioned in the previous chapter. The query processor translates this query with the help of the Schema Manager. The Schema Manager maintains information about the classes defined in the Object Model, their attributes, and the methods. The query is then executed by the query processor using access methods and making runtime optimizations. The methods of types in the object schema are used to query the database and such calls are made through the SHORE server. The database has two parts: the user-defined data and the metadata (schema). The Schema Manager interacts with only the schema while the method definitions of the data model access only the user-defined data part of the database. These interactions are made through the SHORE server.

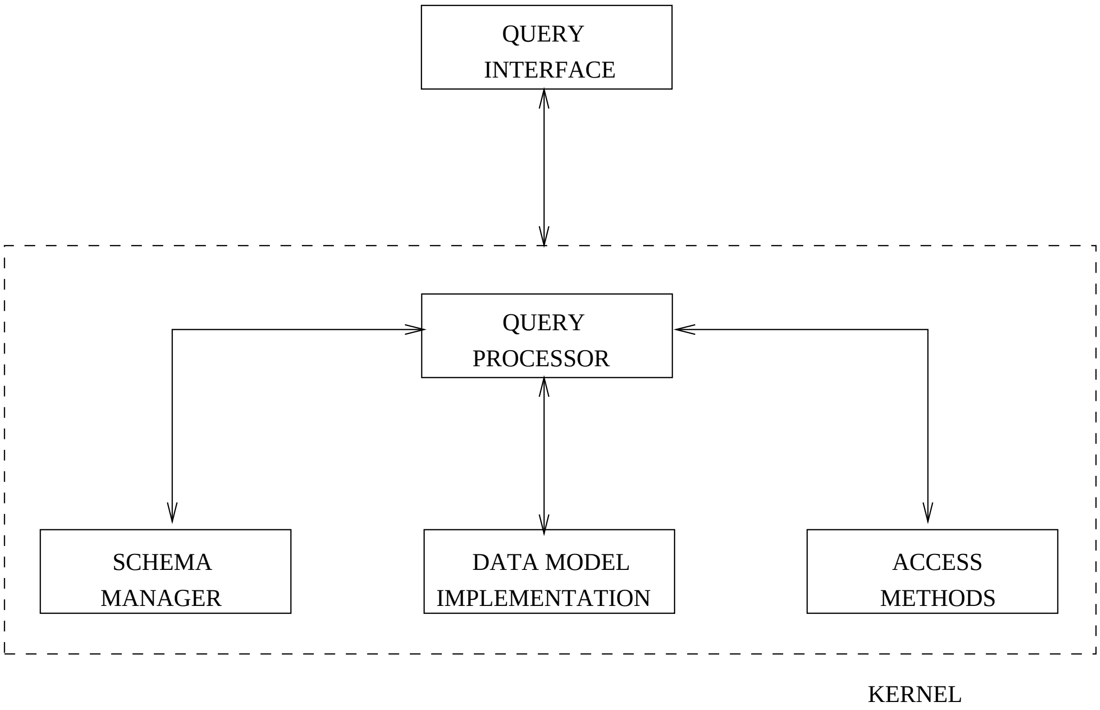


Figure 10.1: DIAS Query Processing System

10.1 Schema Manager

The Schema Manager contains information about all the types in the data model. This information is necessary for ad-hoc querying. The schema is stored as a sequence of records with each record describing information related to an attribute of a class. The template of a schema record is shown in Figure 10.2.

User AttrName	System AttrName	Class Name	Collection Type	Set of Methods	Other Information

Figure 10.2: A Schema Record

10.2 Query Processor

This section describes the construction of the query processor. The architecture of the query processor is shown in Figure 10.3. In the figure, the QUERY and the RESULT are input to and output from the query processor, respectively.

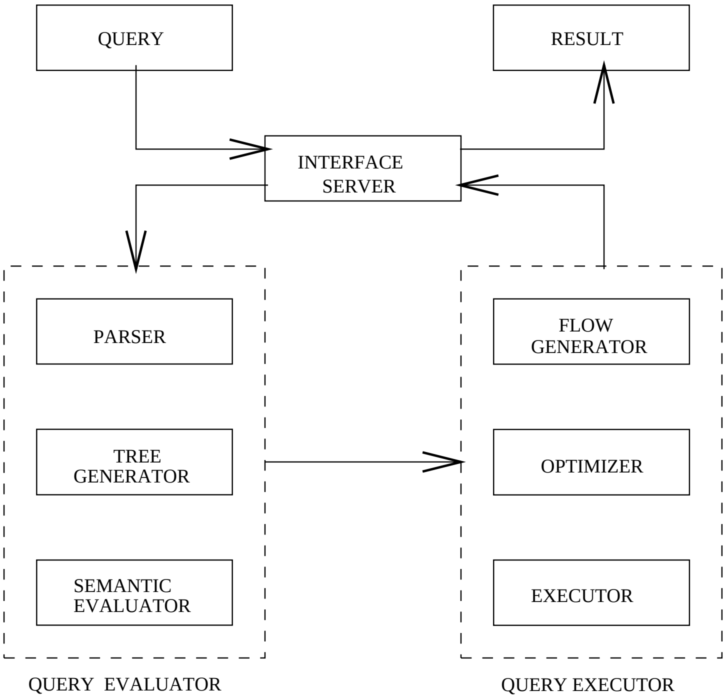


Figure 10.3: Query Processor Architecture

As described in Chapter 9, the query processor takes as input the query in a form similar to the Object Query Language, OQL [Catt94]. The interface server transfers the query to the query evaluator which parses the query, creates a corresponding parse tree and evaluates the query for semantic inconsistencies with the help of the Schema manager. The query evaluator discards the query if an error is found and informs the interface server. If the query is semantically correct, the query evaluator transfers the parse tree to the query executor which generates a flow, optimizes it, and then executes it. The results of the query are transferred to the interface server which then displays the results to the user in a specific format. We now describe each of these components in greater detail.

10.2.1 Interface Server

The interface server module takes the input query specification and transfers it to the query evaluator. If the query evaluation results in the query specification being found incorrect, the query evaluator signals the server to report the error. The interface server is also responsible for tasks such as interpreting the path expressions in the top level *SELECT* clause and applying them to the result objects of the query. The interface server, thus, iterates through the result objects of a query and formats the output.

10.2.2 Query Evaluator

The Query Evaluator consists of the Parser, the Tree Generator, and the Semantic Evaluator. The query evaluator takes the input query from the interface server and transforms it to a parse tree after checking for semantic inconsistencies.

Parser and Tree Generator

The Parser module of the query evaluator is responsible for the parsing of the input query specification and the creation of a parse tree. The query language grammar itself is specified as input to a parser generator tool (*yacc*) and a parser is generated. The lexical analyzer to feed the parser is generated using a token specification specified to the lexical analyzer generator (*lex*). As the parser scans the input query specification, it builds internal data structures that transform the input query into a parse tree. The path expressions specified in the *SELECT* clause are stored as a set for each level of nesting, along with the top level object name. The parse tree is built by a series of operations which essentially entails building subtrees for each predicate and merging them with the corresponding operators for each level of nesting in the query. The parse tree built for each level of nesting is then combined into a single parse tree by a series of merge operations preserving the semantics expressed by the query. Thus, at the end of the parse operation, the nesting in the query specification is converted to a tree structure which enables easier processing of the query. Figure 10.4 shows the parse tree generated for the predicates in the query specification of Figure 9.2.

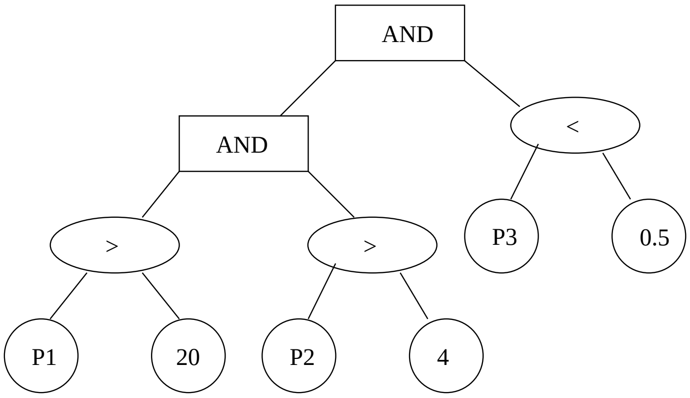


Figure 10.4: An Example Parse Tree

Semantic Evaluator

The semantic evaluator is responsible for evaluating the semantic correctness of the query. It builds a query graph for semantic checking. The query graph is built in stages. For each leaf node representing a path expression in the parse tree, a query graph is constructed. The query graph formed by the union of the query graphs of all the path expressions represents the query. The query graph constructed in this manner for the example query specification of Figure 9.2 is shown in Figure 10.5.

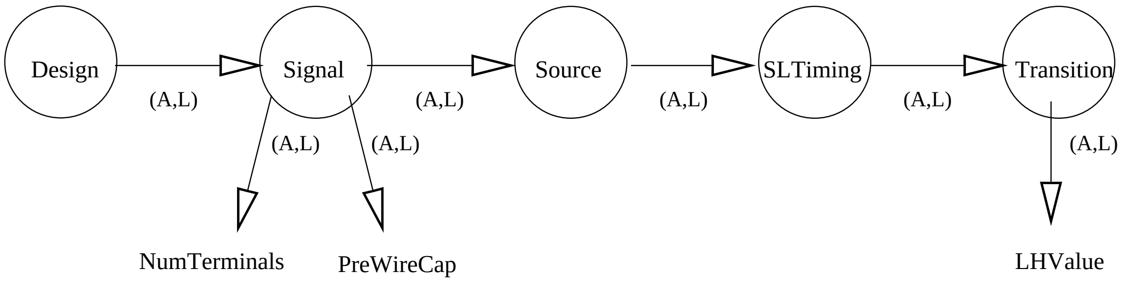


Figure 10.5: Query graph for specification in Figure 9.2

Once the query graph is constructed, semantic evaluation is initiated. Some example evaluation operations are as follows. The paths specified in the query are checked for consistency by testing whether the graph is a subgraph of the graph maintained by the schema manager. The intersection of the query graph for a path expression with the subgraphs representing the predicates of the query should not result in a null graph, since a null graph indicates that a constraint clause has been specified that has no effect on the result.

10.2.3 Query Executor

The Query Executor consists of the Flow Generator, the Optimizer, and the Executor modules. The query executor takes as input the parse tree from the query evaluator and generates a plan for the execution of the query. Using this plan, the query is then executed incorporating certain runtime optimizations.

Flow Generator

The flow generator module is responsible for implementing the order of evaluation of operators in the parse tree. In the current implementation, the flow generator evaluates the subtrees in a postorder fashion. Each subtree consists of an operator and its immediate children in the parse tree. Thus, for the parse tree of Figure 10.4, the subtrees are evaluated in the following order: the subtree representing the predicate $< P1,>,20 >$, the subtree representing $< P2,>,4 >$, the subtree rooted at the AND operator that is the root for both the predicates, the subtree representing the predicate $< P3,<,0.5 >$, and finally the subtree at the root node of the parse tree.

Optimizer

The optimizer is responsible for speeding up operations at the internal nodes of the parse tree and currently concerns itself mainly with the logical operators. This speedup is obtained by implementing these operations on the logical OIDs itself without actually requesting the objects on disk. This operation on surrogates instead of the actual objects themselves can be implemented in main memory itself thus speeding up the evaluation of the query.

The optimizer thus deals with subtrees whose root is a logical operator (AND, OR). The operands consist of set of logical OIDs. It first sorts the the OIDs in the respective sets using a sorting technique such as quicksort and then implements the logical operation. The AND corresponds to a intersection of the sets while the OR corresponds to a union operation.

Executor

The executor module is used to compute the query results. Each subtree generated by the flow generator is used to execute a portion of the query with the optimizer being used for logical operations. The executor uses the query graph constructed by the semantic evaluator to evaluate path expressions. The query graph is traversed node by node with method information supplied by the Schema Manager, sometimes multiple times if the graph nodes contain parametrized types The query graph may also contain references which have to be handled as a separate case by the executor. These references however, do not cause repeated traversals of the query graph by themselves. Access methods are used by the executor if available for the path in consideration. The executor also decides from the select expression, the objects in the path expression that need to be stored as the result and that which should be propagated up the tree. These resultant objects themselves are determined by an intersection of the query graphs representing the select expression and the predicates at a given query nesting level. The resultant objects obtained are finally transferred to the interface server which hands the output over to the user.

Summary

In this chapter, we considered the query processing system of DIAS. The query processor forms an additional component of the DIAS kernel and uses the other components of the kernel to execute queries efficiently. The query processing system interacts with the query interface described in the previous chapter to obtain queries from the user. Thus, the DIAS query processing system provides efficient retrieval mechanisms for parasitic data and supports a declarative object oriented query interface.

Chapter 11

Performance

The DIAS system has been completely implemented according to the details as described in the previous chapters. We have conducted a preliminary set of tests to evaluate its performance. These tests were done on real design data samples provided by Texas Instruments, Inc. The performance testing was carried out in two dimensions: First, the performance of DIAS was compared to the earlier file based system in terms of bulk population of data and retrieval of specific data. This helped to quantitatively evaluate the benefits, if any, of moving from the current file based system to the DIAS system. Second, we then quantitatively determined the impact of the several access methods on query processing in DIAS. That is, the times taken to answer representative queries without the use of access methods were noted. Then, the times for answering the same queries using appropriate access methods were observed. These experiments helped in evaluating whether the benefits of the use of access methods outweigh the disadvantages in terms of the extra storage space used. In this chapter, we present details of the above performance study and provide a summary of the results.

Testing details

The performance evaluation of DIAS was conducted on a Sparc20 machine running Solaris2.4. The response times indicated in the rest of the chapter represent wall-clock times. The file sizes considered represent medium-sized designs and larger designs were not tested because of resource constraints such as disk and main memory. Further testing will be done at Texas Instruments, Inc. We next discuss the details of the performance study.

11.1 Bulk transfer of legacy data

Our first experiment investigated the time taken to load data from legacy text-based design files into the DIAS database, and the time taken to subsequently dump (retrieve) the entire design contents from this database. For this purpose, translators that were developed using the Programming Interface of DIAS were used. The translators also monitored system resource requirements in terms of disk space and main memory.

Design	FileSize(MB)	Terminals	Signals	Load Time	Dump Time
Design1	0.6	150	58	0:10	1:05
Design2	1.3	542	212	2:09	4:08
Design3	2.0	945	349	3:24	8:52
Design4	3.1	1583	542	7:16	16:11
Design5	5.2	2830	889	14:00	30:10

Table 11.1: Load and dump times for varying design sizes (in minutes:seconds)

We evaluated the load time and dump time for different designs that represent varying degrees of complexity. These results are presented in Table 11.1, which shows the designs, the size of the text files that contain their parasitic information, the number of terminals, the number of signals and the corresponding load and access times. From these results, we observe that the times taken for the transfer of data in either direction are relatively small compared to the overall time taken for all the four stages of interconnect analysis, which is typically in the order of several hours. Thus, DIAS provides efficient mechanisms for bulk transfer of data to and from the system. (The dump times are higher than the corresponding loading times due to asynchronous I/O in the cases of writes.)

11.2 Access of specific data

Frequently, the amount of data that needs to be accessed is restricted to a small subset of the whole information, say a signal or a set of signals. We next compare the response times of DIAS with respect to the legacy file-based querying mechanism. The query response times were recorded for representative queries to retrieve a set of signals on the text files and the corresponding design stored in DIAS. The results of this experiment are shown in Table 11.2. They show that DIAS provides a significant improvement in query response time as compared to the file-based system. The main reason for the performance improvement is that, in the file-based system, accessing the specific data requires sequentially parsing the file until the data is located. This scan can take considerable amount of time depending on the size of the file and the position of the required data in the file. In contrast, in DIAS, data objects can be directly accessed by utilising the appropriate access methods. We expect that these performance differences will increase even further for more complex designs.

Design	FileSize (MB)	DIAS	File
Design1	0.6	0.057	3.00
Design2	1.3	0.057	4.00
Design3	2.0	0.057	6.29
Design4	3.1	0.063	11.29
Design5	5.2	0.066	18.71

Table 11.2: Query times for varying design sizes (in seconds)

11.3 Impact of access methods on query processing

Having established the utility of DIAS as compared to the file-based system, we now move on to study the effect of access methods on query processing. As described in Chapter 9, users can submit queries using the Query Interface. The DIAS system uses a variety of access methods to help in executing the query efficiently. We next describe the tests conducted to evaluate the performance of DIAS with respect to the effect of access methods on the query response times.

11.3.1 Simple value based queries

To start with, we used a typical simple value based query and tested it on a set of designs of varying sizes. We evaluated the query in terms of the time taken to execute the following query navigationally (without using access methods) and associatively (using access methods). The representative query chosen is

Query

Give the prelayout wire capacitance of the signal "OUT2" belonging to the design "Design1".

with the query language specification as below:

```
SELECT Signals.PreWireCap
FROM   Design1
WHERE ( Signals.SignalName == "OUT2" );
```

Design	FileSize(MB)	Signals	Navigational	Associative
Design1	0.1	13	1	0.06
Design2	0.7	58	21	0.06
Design3	1.3	212	22	0.07
Design4	2.0	349	24	0.07
Design5	3.3	542	24	0.08

Table 11.3: Response times (in seconds) for a simple value-based query

Table 11.3 shows the times in seconds for answering the query for varying design sizes. It clearly shows the performance gains achieved by the use of suitable access methods, in this case a B$^+$ tree keyed on the names of signals of the design.

11.3.2 Queries targeted on hierarchies

As seen earlier in Chapter 10, the presence of hierarchies in object oriented database systems poses challenges to effective query processing. We next evaluated queries requiring the reverse traversal of aggregation hierarchies.

A representative path corresponding to an aggregation hierarchy was chosen. The path is denoted by the following path expression:

$$Design.Signals.RCNodes.RCEdges.Resistance.NomValue$$

The path chosen is significant because of the enormous number of path instantiations. This is because a Design consists of a number of Signals, each of which has a RC tree composed of a number of RC nodes. Each of these RC nodes are in turn connected to a number of edges. We subsequently use this path in formulating queries to evaluate query processing for hierarchical queries.

Point Queries

We consider the following query as a representative one to test the performance of DIAS with respect to point queries requiring reverse traversal of an aggregation hierarchy:

Query

Retrieve the names of all signals in the design "Design1" with the nominal resistance value of a node in the RC tree being equal to 20 units.

The corresponding query specification is as shown below:

```
SELECT Signals.SignalName
FROM   Design1
WHERE ( Signals.RCNodes.RCEdges.Resistance.NomValue == 20.0 );
```

In the absence of access methods, the query involves the forward traversal from each *Signal* object down to the *RCEdge* objects and testing the nominal value of the resistance for a match. In contrast, the use of a *nested index*, keyed on the nominal value of

Design	FileSize(MB)	Total signals	Navigational	Associative	Signals retrieved
Design1	0.1	13	0:12	0:01	2
Design2	0.7	58	1:13	0:02	30
Design3	1.3	212	3:44	0:03	53
Design4	2.0	349	6:57	0:04	58
Design5	3.3	542	13:24	0:06	101

Table 11.4: Response time (in minutes:seconds) taken for point queries

resistances of all the signals in the design makes it possible to retrieve directly the signal objects that match the predicate. This is evident from the results in Table 11.4 for varying design sizes. The navigational times represent the time taken to evaluate the query using forward traversals while the associative times are those obtained using the nested index. The last column shows the number of signals that satisfy the predicate specified in the query.

Range Queries

Range queries are more common while accessing the parasitic data of a design. For example, a typical query would involve the retrieval of all signals that have the resistance value of an RC node falling in a given interval. This could help in analysing a small critical portion of the circuit instead of the whole circuit. To measure the performance with respect to such queries, the representative query shown below is used.

```
SELECT Signals.SignalName
FROM   Design1
WHERE ( Signals.RCNodes.RCEdges.Resistance.NomValue > 30.0 and
        Signals.RCNodes.RCEdges.Resistance.NomValue < 200.0 );
```

Table 11.5 depicts the comparative results of forward path traversal with respect to the use of access methods. The time taken in the former case is almost the same as the corresponding entries in Table 11.4. This is because the query processor executes the query in a manner similar to the point query by performing a forward path traversal for

Design	FileSize(MB)	Total signals	Navigational	Associative	Signals retrieved
Design1	0.1	13	0:13	0:01	12
Design2	0.7	58	1:16	0:03	39
Design3	1.3	212	3:44	0:03	53
Design4	2.0	349	7:20	0:04	67
Design5	3.3	542	14:26	0:06	117

Table 11.5: Response time (in minutes:seconds) taken for range queries

each signal and matching the predicate. The use of a nested index, on the other hand, results in much faster response due to direct associative retrieval.

Note that the associative times taken in the case of range queries is similar to that for point queries (but are not identical). The reason for this is the following: Because of a large page size used by the Shore storage manager to store the leaf nodes of the index, all the index records fit into a few leaf nodes (for the designs considered here). As a result, processing the range query does not require many additional disk accesses, and no perceptible change is seen at the granularity used in our time measurements. Of course, for larger designs, we would typically expect to see range queries take noticeably more processing time than point queries.

Another point to note is that because of the large number of path instantiations originating from a signal, many signals are associated with a single nominal value of the resistance. As a result, there is considerable overlap in the signals retrieved even among disjoint intervals of resistance values. That is, many of the signals that satisfied the point predicate ($Resistance.NomValue == 20.0$) in the previous query also satisfy the range predicate ($30.0 < Resistance.NomValue < 200.0$).

In summary, we see that the use of the access methods for hierarchies leads to significant speedup both for point and range queries.

Summary

We have conducted initial tests to investigate the performance of DIAS. The tests evaluated two aspects of DIAS: First, its performance compared to its file-based counterpart. Second, the impact of specialised access methods on query processing times was studied. The results have been encouraging and it appears that DIAS shows significant gains over the file based system and that the use of specialized access methods leads to considerable speedups.

Chapter 12

Summary and Future Work

The trend of increasing complexity of integrated circuits has led to the significant adverse impact of interconnect parasitics. This makes it imperative that these effects of interconnect parasitics be taken into account in chip design. The study of interconnect parasitics may be broadly considered to have the following four stages: extraction of interconnect parasitics from the circuit, modeling these using appropriate techniques, incorporation of these parasitics back into the original netlist, and finally analysis of the circuit for various performance parameters. All these stages involve large and complex data calling for effective data management. The file-based approach currently followed by chip design systems is not satisfactory for various reasons and hence forms a serious bottleneck in the chip design process in terms of both functionality and performance. To effectively manage the ever-increasing vastness and complexity of this data, we have proposed an alternative *database technology based* approach. A database management system overcomes the drawbacks of the file system by virtue of its ability to handle vast and complex data effectively.

In this thesis, we have addressed the above issue by developing an object oriented database system, DIAS, that is customized to supporting the needs of interconnect parasitics data. We first developed a comprehensive object oriented model that captures the semantics of the domain. The model represents the various aspects of design and analysis related data. We then designed and implemented the DIAS system that implements the object model. The DIAS system architecture has the Shore server as the backend. The server takes care of object management and transaction processing. The DIAS kernel forms the core of the system and interacts with the Shore server. It implements the schema for storing interconnect parasitic data. It incorporates various access methods for efficient object access and has a complete query processing system. DIAS supports a variety of interfaces to meet the different requirements of the system. A query interface is provided to access specific data. Existing tools can continue to access the data using the tool interface. The programming interface helps in easy development of new tools that access the database directly. To the best of our knowledge, this system represents the first database-related work in the area of interconnect analysis.

12.1 Our contributions

We next highlight the contributions of the work described in this thesis.

Integrated database system

We proposed a database solution for effective data management of VLSI interconnect parasitic data replacing the existing file based approaches. The use of a database system provides a single integrated system for all phases of interconnect study.

Object Model

The Object Modeling Technique was chosen as the methodology for software development. After analysing the modeling requirements, we then presented a detailed object model for representing interconnect parasitic data.

Implementation

The proposed object model was implemented on the Shore extensible database system, using a combination of SDL (Shore Data Language) and C++.

Variety of access methods

The DIAS system incorporates a variety of access methods that are tuned to the pattern of queries in object oriented databases. These include the CH tree, the hcC tree for inheritance hierarchies, the nested index and the path index for aggregation hierarchies. We propose modifications to these access methods in the form of unified indexes to efficiently answer queries targeted on a combination of inheritance and aggregation hierarchies.

Support for legacy applications

A frequent concern for CAD applications is the migration from file based systems to object oriented database systems. DIAS handles this issue by providing a Tool Interface that can be used by file-based applications till new applications that directly access the database are developed.

Declarative Query Interface

A query interface that supports path expressions and aggregate types such as sets is an integral part of DIAS. The query language employs *SELECT-FROM-WHERE* statements similar to the popular SQL.

Efficient query processing

The Query Processing System uses the appropriate access methods and performs other runtime optimizations for the speedy execution of queries. The system is generic enough that it can accommodate changes to the schema easily.

Easy extensibility

DIAS supports the development of various application programs through a library of classes that forms the Programming Interface. The Programming Interface has been already used to develop translators that convert existing design data from files to the object store.

Working Prototype

DIAS is a working prototype and currently runs to approximately 40,000 lines of code. It is currently operational and is being field-tested at Texas Instruments, Inc. A significant aspect of the software is that it is readily portable to a new platform having been completely developed on public-domain software.

12.2 Future Work

As mentioned above, the DIAS system is currently operational, but we feel the system can be extended in several ways to support additional functionality and improved performance. We next outline some suggestions for future research.

Parallelism

For efficient utilisation of resources such as main memory and CPU, DIAS can be extended from its present centralized version into a parallel, distributed system. The Shore system has a symmetric peer-peer server architecture that is readily scalable. An approach in this direction would be to use a message passing library such as PVM [Sund90], for a network of workstations. An important issue then is that of inter-object references. In the centralized case, these were easily resolved since all objects reside on the local disk. Now, with a distributed system, two related objects may no longer reside on the local disk. In the case where the data can be divided into isolated partitions, this problem does not exist. However, in the general case, there must be a mechanism wherein on being presented an OID in a global namespace, the server must be able to retrieve the object from a remote site, calling for location transparency. Another area where parallelism may be exploited is in the area of query processing. The existing Query Processing System may be scaled for parallel execution of queries, leading to significant optimizations in query processing.

Extending the Object Model

The present system is designed to store mainly analysis related data. The object model could be integrated with other models developed for representing VLSI design data to ensure a more comprehensive representation of data in the VLSI design environment.

Multithreading

The Shore server provides multithreading features. These features could be put to a variety of uses. For example, the query processing system could provide a query server that can support multiple users. The server then spawns a thread to handle each user request.

DIAS Value Added Server

One of the features of Shore is the symmetric peer-to-peer architecture. Shore offers capabilities by which a value-added server can be built by bundling additional features specific to the application domain onto the Shore server itself. This feature can be utilised by developing a DIAS Value-Added Server (DVAS). The DVAS may be built using TCL (Tool Command Language) with a customised shell interface having a set of built-in commands.

Frameworks

Existing standardisation work in the area of VLSI CAD includes the development of frameworks. A database system is an integral part of such a framework. DIAS could be extended by providing other components for design methodology management so that it forms a framework integrating both design and analysis aspects of VLSI CAD.

12.3 Final Remarks

In closing, we believe that our work has helped in gaining a better understanding of the issues of data management for VLSI interconnect parasitic analysis. The development of the database system has been influenced by the ideas and contributions of various researchers in the field of database systems and VLSI CAD. We hope that DIAS would be a useful addition to the current efforts in the area of data management for VLSI interconnect parasitic data and help in effective analysis of VLSI interconnect parasitics.

Bibliography

[Alba83] A. Albano, "Type Hierarchies and Semantic Data Models", *SIGPLAN Notices*, June 1983.

[AHKD89] T. Andrews, C. Harris, K. Sinkel, J. Duhl, "The ONTOS Object Database", "Technical Report, Ontologic Inc., Burlington, Massachusetts", 1989.

[AKMP85] H. Afsarmanesh, D. Knapp, D. McLeod and A. Parker, "An Extensible Object-Oriented Approach to Databases for VLSI/CAD", *Proc. of VLDB Conference*, August 1985.

[AgGe89] R. Agarwal, N. Gehani, "ODE (Object Database and Environment): The Language and the Data Model", *Proc. of ACM SIGMOD*, 1989.

[Atk+89] M. Atkinson et al., "The Object-Oriented Database System Manifesto", *Proc. 1st Intl. Conf. on Deductive and Object-Oriented Databases (DOOD'89)*, 1989.

[BaWi64] C. Bachman and S. Williams, "A General Purpose Programming System for Random Access Memories", *Proc. of the Fall Joint Conference*, 1964.

[BKK88] J. Banerjee, W. Kim and H. Kim, "Queries in Object-Oriented Databases", *Proc. of IEEE Intl Conference on Data Engineering*, 1988.

[Bat+90] D. Batory, et al., "The GENESIS Extensible Database System", In *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers, 1990.

[BaKi85] D. Batory and W. Kim, "Modeling Concepts for VLSI CAD Objects", *ACM Trans. on Database Systems*, March 1985.

[Ber94a] E. Bertino, "Index configuration in Object-Oriented Databases", *The VLDB Journal*, July, 1994.

[Ber94b] E. Bertino, "A Survey of Indexing Techniques for Object-Oriented Database Management Systems", In *Query Processing for Advanced Database Systems*, Morgan Kaufmann Publishers, 1994.

[BeKi89] E. Bertino and W. Kim, "Indexing Techniques for queries on nested objects", *IEEE Trans. on Knowledge and Data Engineering*, 1989.

[BPR88] M. Blaha, W. Premerlani, J. Rumbaugh, "Relational database design using an object-oriented methodology" *Communications of the ACM*, April, 1988.

[BlZd87] T. Bloom and S. Zdonik, "Issues in the design of an Object Oriented Database Programming Language", *Proc. of the 2nd OOPSLA*, October, 1987.

[Booc94] G. Booch, *Object Oriented Analysis and Design with Applications*, Benjamin Cummings, second edition, 1994.

[Bret+88] R. Bretl, et al., "The GemStone Data Management System", In *Object-Oriented Concepts, Databases and Applications*, Addison-Wesley Publishing Company (ACM Press), 1989.

[Breu+88] M. Breurer, et al., "Cbase 1.0: A CAD database for VLSI circuits using Object Oriented technology", *Proc. of ICCAD*, 1988.

[CFI94] CAD Framework Initiative, "Design Representation Programming Interface", version 1.4, 1994.

[Car+90] M. Carey et al., "The EXODUS Extensible DBMS Project: An Overview", In *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers, 1990.

[Catt94] R. Cattell, *The Object Database Standard: ODMG-93*, Morgan Kaufmann, 1994.

[CaWe85] L. Cardelli, P. Wegner, "On understanding Types, Data Abstraction, and Polymorphism", *ACM Computing Surveys*, December 1985.

[CDKK85] H. Chou, D. DeWitt, R. Katz, A. Klug, "The Design and Implementation of the Wisconsin Storage System", *Software Practice and Experience*, October, 1985.

[CoAd91] P. Coad, E. Yourdon, *Object-Oriented Design*, Prentice Hall, 1990.

[Coc+89] W. Cockshott, et al., "Persistent Object Management System", *Software Practice and Experience*, January, 1984.

[Codd70] E. Codd, "A Relational Model for Large Shared Data Banks", *Communications of the ACM*, June, 1970.

[Codd72] E. Codd, "Further Normalization of the Relational Database Model", In *Database Systems*, Prentice-Hall, 1972.

[Codd74] E. Codd, "Recent Investigations in Relational Database Systems", *Proc. of the IFIP Congress*, 1974.

[Cole+94] D. Coleman, et al., *Object-Oriented Development: The Fusion Method*, Prentice Hall International Edition, 1994.

[Come79] D. Comer, "The Ubiquitous B-tree", *ACM Computing Surveys*, June 1979.

[DaDi91] J. Daniell and S. Director, "An object oriented approach to CAD tool control", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, June, 1991.

[Deux+90] O. Deux et al., "The story of O_2", *IEEE Trans. on Knowledge and Data Engineering*, March 1990.

[EDIF93] Electronic Industries Association EDIF Steering Committee, "EDIF Information Model", version 3 0 0, 1993.

[Ga+87] J. Gannon et al., "Theory of Modules", *IEEE Transactions on Software Engineering*, July 1987.

[GrRe93] J. Gray, A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, 1993.

[Gup+89] R. Gupta et al., "An Object Oriented VLSI CAD Framework - A Case Study in Rapid Prototyping", *IEEE Computer*, May 1989.

[Gutt84] A. Guttman, "R-Trees: a dynamic index structure for spatial searching", *Proc. of ACM SIGMOD*, June 1984.

[HMSN86] D. Harrison, P. Moore, R. Spickelmier and A. Newton, "Data management and graphics editing in the Berkeley design environment", *Proc. of ICCAD*, 1986.

[HNSB90] D. Harrison, A. Newton, R. Spickelmier and T. Barnes, "Electronic CAD Frameworks", *Proc. of the IEEE*, February 1990.

[IEEE96] IEEE India Bulletin, "Newsletter of IEEE India Council", April 1996.

[IlUG95] Illustra Users Guide, *Illustra Server Release 3.2* Illustra Information Technologies, Inc.

[Jone79] A. Jones, "The Object Model: A Conceptual Tool for Structuring Software" In *Operating Systems*, Springer Verlag, 1979.

[Khan91] S. Khanna, "Sorting out Signal Integrity", *Electron. Engin. Times*, June 1991.

[Kim90] W. Kim, " Object-Oriented Databases: Definition and Research Directions." *IEEE Transactions on Knowledge and Data Engineering*, September 1990. *Object-Oriented Design*, Prentice Hall, 1990.

[KKD89] W. Kim, K. Kim and A. Dale, "Indexing Techniques for Object Oriented Databases", In *Object-Oriented Concepts, Databases and Applications*, Addison-Wesley Publishing Company (ACM Press), 1989.

[Kris96] M. Krishna, Query Processing and Bulkloading in OODBMS, *ME Thesis, CSA Dept., Indian Institute of Science*, 1996.

[Lamb+90] C. Lamb et al., "The ObjectStore Database System", *IEEE Trans. on Knowledge and Data Engineering*, March 1990.

[Lee95] B. Lee, "Normalization in OODB Design", *SIGMOD Record*, September, 1995.

[LOL92] C. Low, B. Ooi and H. Lu, "H-trees: A Dynamic Associative Search Index For OODB", *Proc. of ACM SIGMOD*, 1992.

[Maie89] D. Maier, "Making Database Systems fast enough for CAD Applications", In *Object-Oriented Concepts, Databases and Applications*, Addison-Wesley Publishing Company (ACM Press), 1989.

[MaSt86] D. Maier and J. Stein, "Indexing in an object oriented database", *Proc. of IEEE Workshop on Object-Oriented DBMSs*, September 1986.

[MaSt90] D. Maier and J. Stein, "Development and Implementation of an Object-Oriented DBMS", In *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers, 1990.

[McGee77] W. McGee, "The Information Management System IMS/VS", *IBM Systems Journal*, June, 1977.

[MoLe92] C. Mohan, F. Levine, "ARIES/IM: An efficient and high concurrency index management method using write-ahead logging", "Proc. of the ACM SIGMOD", 1992.

[MHL+92] C. Mohan, et al, "ARIES/IM: An efficient and high concurrency index management method using write-ahead logging", "Proc. of the ACM SIGMOD", 1992.

[MoPu92] D. Monarchi, G. Puhr, "A Research Typology for Object-Oriented Analysis and Design", *"Communications of the ACM"*, September, 1992.

[MoSi88] J. Moss, S. Sinofsky, "Managing Persistent Data with Mneme: Designing a Reliable Shared Object Interface", In *Advances in Object Oriented Database Systems: Second International Workshop on Object-Oriented Database Systems*, Springer Verlag, 1988.

[Pratt94] P. Pratt, J. Adamski, *Database Systems Management and Design*, Boyd and Fraser Publishing Company, Danvers, MA, 1994.

[RaKa95] S. Ramaswamy and P. Kanellakis, "OODB Indexing by Class-Division", *Proc. of ACM SIGMOD*, 1985.

[RaPi94] C. Ratzlaff and L. Pillage, "RICE: Rapid Interconnect Evaluation using AWE" *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, June 1994.

[RoSt90] L. Rowe and M. Stonebraker, "The POSTGRES data model", In *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers, 1990.

[Ru+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[SAL93] A. Singhal, R. Arlein and C. Lo, "DDB: An Object Oriented Design Data Manager for VLSI CAD", *Proc. of ACM SIGMOD*, 1993.

[Sch+90] P. Schwartz, et al., "Starburst Mid-Flight: As the dust clears", *IEEE Trans. on Knowledge and Data Engineering*, March 1990.

[ShMe88] S. Shlaer, S. Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data*, Prentice Hall, 1988.

[SPDL89] A. Singhal, N. Parikh, D. Dutt and C. Lo, "A Data Model and Architecture for VLSI/CAD Databases", *Proc. of ICCAD*, 1989.

[SrSe94] B. Sreenath and S. Seshadri, "The hcC-tree: An Efficient Index Structure For Object Oriented Databases", *Proc. of VLDB Conference*, 1994.

[StRo86] M. Stonebraker and L. Rowe, "The design of POSTGRES", *Proc. of ACM SIGMOD*, 1986.

[Sund90] "PVM : A framework for parallel distributed computing", *Concurrency: Practice and Experience*, 2(4), 1990.

[Vbase] "Ontologic, Inc.", *Vbase Functional Specification*, Ontologic, Inc., 1986.

[WiBr90] R. Wirfs-Brock, B. Wilkerson, L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990.

[Your89] E. Yourdon, *Modern Structured Analysis*, Yourdon Press, 1989.

[YoCo79] E. Yourdon and L. Constantine, *Structured Design*, Yourdon Press, 1979.

[ZdMa90] S. Zdonik and D. Maier, "Fundamentals of Object-Oriented databases", In *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers, 1990.

[Zwi+94] M. Zwilling et al., "On Shoring Up Persistent Applications", *Proc. of ACM SIGMOD*, 1994.

Appendix A

SDL Declarations

The following are declarations in the Shore Data Language, SDL of a few important classes.

```
          ///////// The basic design entities ///////////
interface Port
{
        private:
                attribute string PortName;
                attribute nameset PortNames;
                attribute ref<Triplet>  PortCapacitance;
                attribute set<Coefficient> Coefficients;
                attribute ref<Transition>  PortDtp;
                attribute set<PathDelay>  PathSet;
                attribute PortDirType  PortDirection;
                attribute ref<Transition> OutSlew;
                attribute short  PortNumber;
                attribute ref<Cell>  OwnerCell;
        public:
                void Print( out ostream os, in lref<lref<char> > args,
                                in short numargs );
```

```
                void Construct( in short Number);
                void AddCoefficient( in ref<Coefficient> NewCoeff);
                ref<Cell> GetOwnerCell() ;
                ref<Cell> SetOwnerCell(in ref<Cell> NewOwner) ;
                PortDirType GetDirection();
                PortDirType SetDirection( in PortDirType NewDirection) ;
                ref<Coefficient> FindCoefficient(
                                                in lref<char> CoefName);
                lref<char> GetName( in short Number );
                void SetName( in lref<char> Alias);
                short  GetPortNumber()  ;
                void  SetPortNumber( in short NewNum );
                ref<PathDelay> FindPath( in ref<Port> Source );
                ref<PathDelay> GetPath( in short ind );
                ref<Coefficient> GetCoefficient( in short ind );
                void AddPath( in ref<PathDelay> NewPath) ;
                void Capacitance(in ref<Triplet> NewRef);
                ref<Triplet> Capacitance ( );
                void SetDtp ( in ref<Transition> NewRef );
                ref<Transition> GetDtp ( );
                void SetOutSlew ( in ref<Transition> NewRef );
                ref<Transition> GetOutSlew ( );
};
interface Cell
{
        private:
                attribute string CellName;
                attribute set<Port> Ports;
                attribute DeviceType Type;
```

```
                void Display( out ostream os, in short fieldWidth,
                                                in short attrNumber );
        public:
                void Print( out ostream os, in lref<lref<char> > args,
                        in short numargs );
                void Construct ( in lref<char> NewName );
                void SetName(in lref<char> NewName);
                lref<char> GetName();
                short NumberPorts() ;
                ref<Port> GetPort(in short PortNumber);
                DeviceType GetType() ;
                DeviceType SetType(in DeviceType NewType) ;
                ref<PathDelay> FindPath( in ref<Port> Input,
                                                in ref<Port> Output);
                ref<Port> GetPort(in lref<char> Name) ;
                void   AddPort( in ref<Port> NewPort );
};
interface LibraryCell : public Cell
{
        private :
                attribute set<Transistor>  Transistors;
                attribute ref<TimingInfo> Timing;
        public :
                short HowManyTransistors( );
                ref<Transistor> GetTransistor( in short counter );
                ref<Timimg>  GetTiming();
};
interface Device
{
```

```
        private:
                attribute string DeviceType;
                attribute short DeviceNumber;
                attribute string ModelName;
                attribute string DevName;
                attribute ref<Cell>  Master;
                attribute set<Terminal> PortInstances;
        public:
                void Print( out ostream os, in lref<lref<char> > args,
                        in short numargs );
                void Construct( in lref<char> Dev, in ref<Cell>
                                                        MasterTemplate);
                ref<Cell> MasterCell();
                lref<char> Name();
                short HowManyInstances( );
                ref<Terminal> AddPortInstance( in
                                        ref<Terminal> NewPort ) ;
                ref<Terminal> GetPortInstance( in short PortNum );
                ref<Terminal> SetPortInstance( in short PortNum,
                                in ref<Terminal> NewPort );
                short NumPorts ();
};
            ///////// The RC Tree related classes ///////////
interface RCTreeEdge
{
        private:
                attribute ref<Triplet> Resistance;
                attribute ref<Triplet> Capacitance;
                attribute ref<Triplet> SmNC;
```

```
                attribute ref<RCTreeNode>  EdgeNodes[ 2 ];
        public:
                void Print( out ostream os, in lref<lref<char> > args,
                                                in short numargs );
                void Construct(in ref<RCTreeNode>  Node1,
                                        in ref<RCTreeNode> Node2,
                                        in ref<Triplet> Res,
                                        in ref<Triplet> Cap,
                                        in ref<Triplet> SmNC);
                void Construct(in ref<RCTreeNode>  Node1,
                                        in ref<RCTreeNode> Node2 );
                ref<RCTreeNode> Partner(in ref<RCTreeNode>
                                                        CurrentNode);
                void SetCapacitance(in ref<Triplet> NewRef );
                ref<Triplet> GetCapacitance( );
                void SetSmallerNodeCap(in ref<Triplet> NewRef );
                ref<Triplet> GetSmallerNodeCap( );
                void SetResistance(in ref<Triplet> NewRef );
                ref<Triplet> GetResistance( );
};
interface RCTreeNode
{
        private:
                attribute ref<Terminal> RCNodePort;
                attribute set<RCTreeEdge> Edges;
                attribute short NodeNumber;
                void Display( out ostream os, in short fieldWidth,
                                                in short attrNumber );
        public:
```

```
                void Print( out ostream os, in lref<lref<char> > args,
                                        in short numargs );
                void Construct(in short Num) ;
                void InputEdge( in ref<RCTreeEdge> NEdge );
                ref<RCTreeEdge> GetEdge ( in short Num );
                short GetNumber();
                short HowManyEdges();
                ref<Terminal> SetPort(in ref<Terminal>
                                                NewRCNodePort);
                ref<Terminal> GetPort();
};
            ///////// The Signal class ///////////
interface Signal
{
        private:
                attribute string SignalName;
                attribute ref<Design> SignalOwner;
                attribute set<SourceLoadTiming> TimingSet;
                attribute set<RCTreeEdge> RCEdgeSet;
                attribute set<Terminal> Ports;
                attribute set<RCTreeNode> Nodes;
                attribute ref<Triplet> PreWireCap;
                attribute short  CriticalValue;
        public:
                void Print( out ostream os, in lref<lref<char> > args,
                                                in short numargs );
                void Construct(in lref<char> Name);
                lref<char> GetName();
                void SetCritical( in short NewValue );
```

```
short GetCritical( );
ref<RCTreeEdge>  AddEdge(in short Node1,
                                in short Node2);
short AddPort (in ref<Device> Instance, in lref<char>
                        Name);
ref<RCTreeNode> MakeNode(in short NodeID);
ref<RCTreeEdge> GetEdge ( in short ind );
ref<RCTreeNode> GetNode ( in short ind );
ref<Terminal> GetTerminal( in short ind );
ref<SourceLoadTiming> GetTiming( in short ind );
void InputTiming( in ref<SourceLoadTiming> NewRef);
void InputNode( in ref<RCTreeNode> NewRef);
void InputEdge( in ref<RCTreeEdge> NewRef);
void InputTerminal( in ref<Terminal> NewRef);
ref<Terminal> SearchTerminal( in short Number );
ref<SourceLoadTiming> CreateTiming(in
        ref<Terminal> Source, in ref<Terminal> Load);
ref<SourceLoadTiming> FindTiming(in ref<Terminal>
                        Source, in ref<Terminal> Load);
short Attach(in short RCNodeID, in short PortID);
ref<Port> FindPort(in lref<char> Name,
                  in lref<char> Instance) ;
ref<RCTreeNode> FindNode(in short ID);
ref<Design> Owner() ;
ref<Design> Owner(in ref<Design> NewOwner);
short HowManyTerminals();
short HowManyTimings();
short HowManyEdges();
short HowManyNodes();
```

```
                void SetPreWireCapacitance(in ref<Triplet> NewRef );
                ref<Triplet> GetPreWireCapacitance( );
};
            ////////////// The DESIGN class /////////////////
interface Design
{
        private:
                attribute string DesignDate;
                attribute string DesignDir;
                attribute string DesignLibrary;
                attribute string DesignName;
                attribute string Version;
                attribute short Loads;
                attribute DesignType  DesignType;
                attribute ref<Triplet> DesignVoltage;
                attribute ref<Triplet> DesignTemperature;
                attribute ref<Triplet> DesignProcess;
                attribute double DelayUnits;
                attribute double ResistanceUnits;
                attribute double CapacitanceUnits;
                attribute ref<Triplet> ConversionFactors;
                attribute double ThresholdValue;
                attribute string ProgramName;
                attribute string ProgramVersion;
                attribute string ProgramProperty;
                attribute string TechVersion;
                attribute short TechMetalLayers;
                attribute string<50> TechName;
                attribute short PVTFromCellLib;
```

```
        attribute boolean Designisreduced;
        attribute boolean Designiscollapsed;
        attribute short Designpcmultfactor;
        attribute float Designblockarea;
        attribute set<Cell> Cellset;
        attribute set<Device> Deviceset;
        attribute set<IONet> IONetset;
        attribute set<Signal> Signalset;
        attribute index<string,Cell> Cellindex;
        attribute index<string,Device> Deviceindex;
        attribute index<string,IONet> IONetindex;
        attribute index<string,Signal> Signalindex;
        void Display( out ostream os, in short fieldWidth,
                                 in short attrNumber );
public:
        void Print( out ostream os, in lref<lref<char> > args,
                         in short numargs );
        void Construct();
        void Destruct();
        ref<Triplet> GetConvFactors (  );
        void SetConvFactors( in ref<Triplet> NewRef );
        lref<char> SetTechName( in lref<char> name);
        lref<char> GetTechName( );
        lref<char> GetDate();
        void SetDate( in lref<char> NewDate );
        DesignType GetDesType();
        void SetDesType( in DesignType NewDesignType );
        lref<char> GetDesDirectory();
        void SetDesDirectory(in lref<char> NewDesignDir);
```

```
lref<char> GetLibrary();
void SetLibrary(in lref<char> NewLibrary);
short GetLoads();
void SetLoads(in short NewValue );
lref<char> GetDesName();
void SetDesName ( in lref<char> NewName );
lref<char> GetProgramName();
void SetProgramName(in lref<char> NewProgramName);
lref<char> GetProgramProperty();
void SetProgramProperty(in lref<char> NewProp);
lref<char> GetProgramVersion();
void SetProgramVersion(in lref<char> NewProgramVersion);
short GetTechnologyMetalLayers();
short SetTechnologyMetalLayers(in short NewMetalLayers);
lref<char> GetTechnologyVersion();
void SetTechnologyVersion(in lref<char> NewVersion);
double GetThreshold();
void SetThreshold( in double value );
void SetUnits(in double Res, in double Cap,
                         in double Del);
void GetUnits(in lref<double> Res, in lref<double> Cap,
                         in lref<double> Del);
double GetDelUnits();
double GetResUnits();
double GetCapUnits();
lref<char> GetVersion();
void SetVersion(in lref<char> NewVersion) ;
void SetProcess(in ref<Triplet> NewRef );
ref<Triplet> GetProcess( );
```

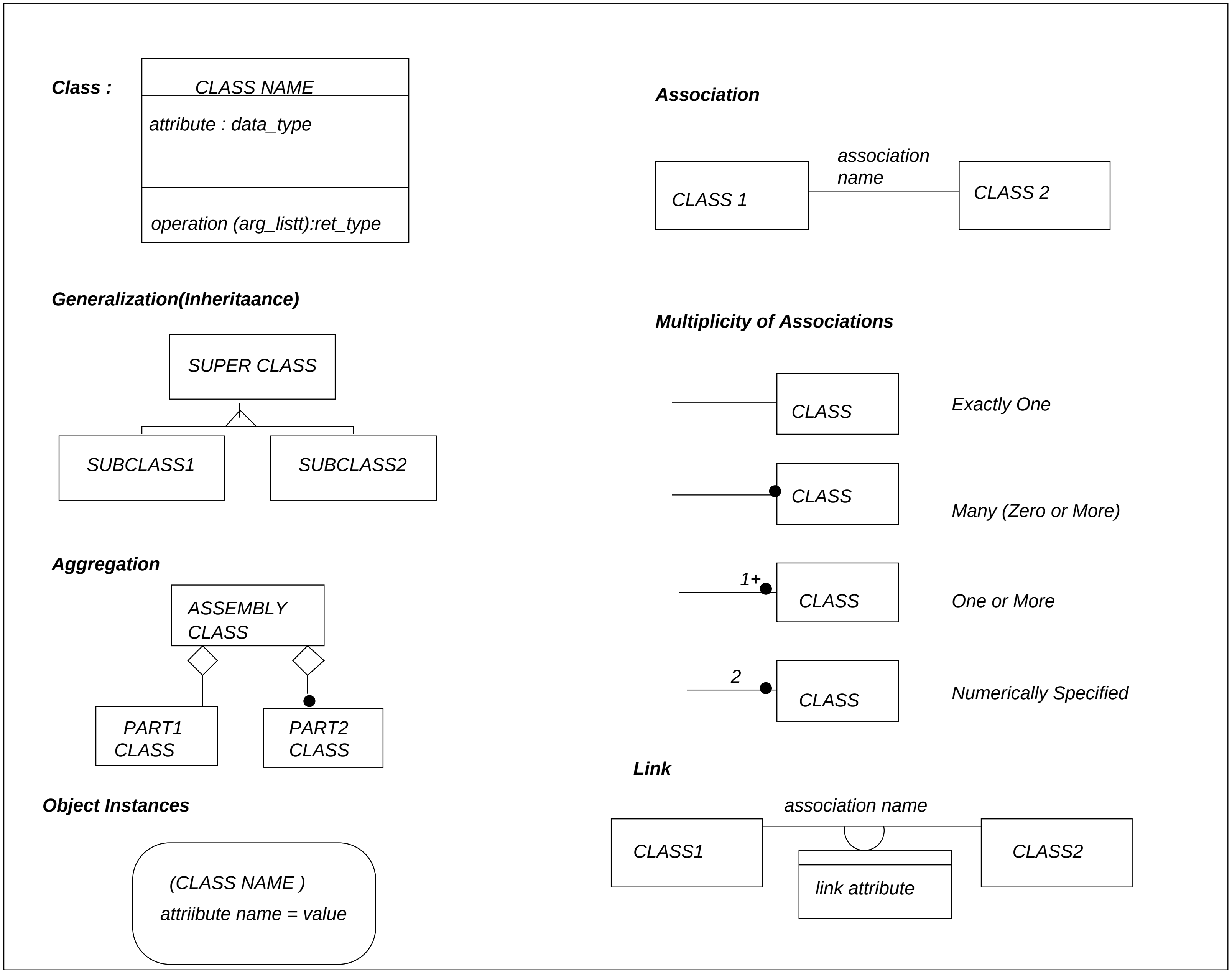
```
void SetTemperature(in ref<Triplet> NewRef );
ref<Triplet> GetTemperature( );
void SetVoltage(in ref<Triplet> NewRef );
ref<Triplet> GetVoltage( );
void  AddCell(in ref<Cell> Dev);
ref<Cell> FindCell(in lref<char> CellName);
void  DeleteCell(in lref<char> CellName);
ref<Device> FindDevice(in lref<char> DeviceName);
void  DeleteDevice(in lref<char> DeviceName);
ref<Device> AddDevice(in lref<char> DeviceName,
                         in lref<char> CellType);
ref<Device> AddDevice(in ref<Device> NewDevice );
ref<IONet> AddIONet(in lref<char> NetName);
ref<IONet> AddIONet(in ref<IONet> NewIONet );
void  DeleteIONet(in lref<char> NetName);
ref<IONet> FindIONet(in lref<char> NetName);
void  AddSignal(in ref<Signal> Sig);
void  DeleteSignal(in lref<char> SigName);
ref<Signal> FindSignal(in lref<char> SigName);
ref<Port> FindPort(in lref<char> DeviceName,
        in lref<char> PortName);
ref<Port> FindPort(in lref<char> DeviceName,
                               in short PNum);
short HowManyCells();
short HowManySignals();
short HowManyDevices();
short HowManyIONets();
ref<Cell> GetCell ( in short ind );
ref<Device> GetDevice ( in short ind );
```

```
                ref<Signal> GetSignal ( in short ind );
                ref<IONet> GetIONet ( in short ind );
                ref<Cell> SearchCell ( in lref<char> Target );
};
        ////////////// HCC Tree  ///////////////////////
//The main class declared for implementing the hcC tree
typedef ref<Query> OID;
interface HCCTree
{
        private:
                attribute index<string,LONE> Tree;
                attribute ref<CONEPage> FirstCChainPage[ NumClasses ];
                attribute ref<HONEPage> FirstHChainPage;
        public:
                void Construct();
                ref<OIDSet> SCP( in short cid, in lref<char> v1 );
                ref<ClassListSet> CHP( in ref<CIDSearch>
                                TargetList, in lref<char> v1 );
                ref<CONESet> SCR( in short cid, in lref<char> v1,
                                        in lref<char> v2 );
                void CHR( in ref<CIDSearch> TargetList, in
                                lref<char> v1, in lref<char> v2 ,
                                out ref<ClassCONESet> ResultPtr1,
                                out ref<HONESet> ResultPtr2);
                void Insert( in lref<char> v, in OID NewOID,
                                in short Cid );
                void Delete( in lref<char> v, in OID NewOID,
                                in short Cid );
                void Print();
```

```
                ref<CONEPage> SearchClassOIDNode( in lref<char> v1,
                                  in lref<char> v2, in short Cid );
                ref<HONEPage> SearchHierOIDNode( in lref<char> v1,
                                  in lref<char> v2, in short Cid );
                ref<CONEPage> ClassChainOIDInsert( in lref<char> v,
                 in OID oid, in short Cid, in ref<CONEPage> Page );
                ref<HONEPage> HierChainOIDInsert( in lref<char> v,
                 in OID oid, in short Cid, in ref<HONEPage> Page );
                ref<CONEPage> FindClassChainOIDNode( in lref<char> v,
                              in short Cid, in ref<CONEPage> Page );
                ref<HONEPage> FindHierChainOIDNode( in lref<char> v,
                              in short Cid, in ref<HONEPage> Page );
                ref<CONEPage> ClassOIDInsert( in ref<CONEPage> Page,
                     in lref<char> v, in OID NewOID, in short cid );
                ref<HONEPage> HierOIDInsert( in ref<HONEPage> Page,
                     in lref<char> v, in OID NewOID, in short cid);
                ref<CONEPage> ClassDeleteOID( in lref<char> v,
                     in OID oid, in ref<CONEPage> Page );
                ref<HONEPage> HierDeleteOID( in lref<char> v,
                     in OID oid, in ref<HONEPage> Page,in short cid );
                lref<char> MaxKey( in ref<HONEPage> Page );
};
```

Appendix B

The Rumbaugh Notation



Class :
CLASS NAME
attribute : data_type
operation (arg_listt):ret_type
Generalization(Inheritaance)
SUPER CLASS
SUBCLASS1
SUBCLASS2
Aggregation
ASSEMBLY CLASS
PART1 CLASS
PART2 CLASS
Object Instances
(CLASS NAME)
attriibute name = value
Association
CLASS 1
association name
CLASS 2
Multiplicity of Associations
CLASS
Exactly One
CLASS
Many (Zero or More)
1+
CLASS
One or More
2
CLASS
Numerically Specified
Link
association name
CLASS1
link attribute
CLASS2

Appendix C

DIAS Object Model