CADIS: A KERNEL APPROACH TOWARD THE DEVELOPMENT OF INTELLIGENT DATA MANAGEMENT SUPPORT OF COMPUTER AIDED DESIGN SYSTEMS

Dorothy Miller Landis

February 1988
CSD-880012

UNIVERSITY OF CALIFORNIA

Los Angeles

CADIS: A Kernel Approach Toward the Development of Intelligent Data Management Support of Computer Aided Design Systems

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Computer Science

by

Dorothy Miller Landis

1988

The dissertation of Dorothy Miller Landis is approved.

[illegible]

Bruce Miller

[illegible]

R. Clay Sprowls

[illegible]

D. Stott Parker

[illegible]

Bertram Bussell, Committee Co-Chair

[illegible]

Gerald Estrin , Committee Co-Chair

University of California, Los Angeles

1988

TABLE OF CONTENTS

page

LIST OF FIGURES

page

ACKNOWLEDGMENTS

I would like to thank Professors Miller, Sprowls and Parker for serving on my doctoral committee. A special thanks to Professor Bertram Bussell for serving as co-chair, as well as providing much needed support and friendship. Not only thanks, but also my most sincere gratitude is extended to Professor Gerald Estrin, who took invaluable time to co-chair this dissertation. Without his support, guidance and patience, I would not have been able to finish this work.

The members of the UCLA SARA/IDEAS Research Group: Daniel Berry, Gerald Estrin, Eduardo Krell, Eddie Lor, Sergio Mujica, Dorab Patel, and Duane Worley are acknowledged for their contributions.

At this time, I would also like to recognize the administration and faculty of the School of Engineering and Computer Science at California State University, Northridge for their *genuine* commitment to the concept of Affirmative Action, which was the impetus for my embarking on this academic journey. Their support came in all forms; providing release time from my teaching duties, as well as providing the proverbial shoulder to cry on. In particular, I would like to thank Marlene Bloch, Professors Diane Schwartz, Mike Barnes, and John Motil for their support and friendship.

This period of academic endeavor has extended over many years. During this time many changes have occured in my life. However, some parts of my life have remained unchanged. This would include the constant support provided by my family; my parents, Luther and Rosamond Miller; and my children, Susan Woodgrift, Nancy Landis, and Gary Landis. I would particularly like to thank my children for not only being my friends, but also serving, each in their own way as role models for me.

The life changes also included the acquisition of many new friends. I am particularly indebted to Dr. Berta Davis, for providing me with the support necessary to focus on the task at hand, finishing this dissertation. Last, but not least, a special thanks to Dr. Ralph Sabella, my friend and confidante, for helping me create an environment within which I could complete this work.

This research was supported in part by the Exxon Foundation through a Doctoral Fellowship and by the State of California MICRO Program.

VITA

June 14, 1940	Born, Salisbury, N. C.
1965	B.A., California State University Northridge
1968-1978	Mathematics Teacher Simi Valley High School, Simi
1973	M.A., Mathematics Education
1978-1983	Assistant Professor California State University Northridge
1982	M.S., University of California, Los Angeles
1982-1987	Research Assistant, School of Engineering
	University of California, Los Angeles
1983-1987	Associate Professor California State University Northridge

PUBLICATIONS AND PRESENTATIONS

Cai, S., Landis, D., Patel D., and Worley D. (1985), "An Experiment to Determine Approiate Data Structuring Methods for the SARA-IDEAS Structure Model (SM)" Internal Memorandum #212, Computer Science Department, University of California, Los Angeles, 1985.

Landis D. (1983), *Design Considerations for the Satisfaction of CAD Library Requirements*, Master Thesis, University of California, Los Angeles June 1983.

Landis D. (1985), *CADIS: A Kernel Approach Toward the Development of Intelligent Data Management of Computer Aided Design Systems*, in the Proceedings of the 1985 ACM 13th Annual Computer Science Conference, New Orleans, LA.

ABSTRACT OF THE DISSERTATION

CADIS: A Kernel Approach Toward the

Development of Intelligent Data Management for

Computer Aided Design Systems

by

Dorothy Miller Landis

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1988

Professor Gerald Estrin, Co-Chair

Professor Bertram Bussell, Co-Chair

This dissertation presents the design and development of CADIS (Computer Aided Design Information System). CADIS addresses the data management needs of Computer Aided Design (CAD) systems by providing a system which overcomes the inherent problems found when using conventional data base management systems (DBMS). Limitations found with other research and development efforts marked for correction include: lack of flexibility in the supported CAD system, separation of the design interaction from database interaction, separation of knowledge base issues from data management issues, lack of a distinction between a DBMS' implementation of system policy and a DBMS mechanism to support system policy.

A four level data base architecture is defined for CADIS. An augmented entity-relationship model (AERM) is also defined which supports the CADIS architecture. This model includes structures which are developed to specifically overcome limitations found in the use of the more traditional data model. A definition and manipulation language for the model are defined. Translation rules are derived which enable the AERM to be translated into an undelying data base scheme. A proof-of-concept model is presented via the the definition and implementation of a system tool, the SARA/IDEAS System Browser.

CHAPTER 1

Introduction

1.1 Historical Perspective

Data Base Management Systems (DBMS) evolved from File Management Systems primarily in response to data-processing's need for the integration of information. This evolution resulted in benefits such as reduced data redundancy, sharable data, enforcement of standards, maintenance of integrity, and avoidance of inconsistencies. Today, the use of large integrated data bases in the commercial data processing world has been fully established. Computers were also being used in the engineering/scientific community to solve numerical problems. These solutions produced large volumes of data which were stored on files. Data reduction techniques often were applied to reduce this volume and numerous translators were written to reformat this data for use in other areas. Since the emphasis at this time was on the development of batch applications, such as the generation of engineering drawings, logic simulators and component placement techniques, the need for an integrated database was at first given low priority. However, when effective, efficient interactive I/O devices became available, many of these same engineers began creating data interactively within a Computer Aided Design (CAD) environment. Soon the need to integrate data became apparent [Lori81].

The recognition within the engineering community of a need for some type of data management support resulted in various types of data management design implementations. The three approaches, described in the following paragraphs, represent the first two stages in the evolution of data base system development: design by default, and design for the introduction of the data base concept [Inmo81]. CAD data base systems have now evolved to the latter two stages: design for performance, and design for flexibility. However, since the problems encountered in the first two stages define the directions taken within the final stages, a brief description of these approaches is appropriate.

One approach was to provide embellishments to existing file systems. An illustration of this approach is the CAD data base developed at Raytheon [Ciam76]. The approach taken resulted in a system which provided convenient fast access (superior to DBMS) to large amounts of data, but lacked essential characteristics of a DBMS, such as processing based on data values, recovery, access control, and concurrency control. In fact, this type of system provided for little more than the storage and retrieval of designs, along with some minimal query facilities. The designers at Raytheon experienced frequent restructuring of the file system and increased complexity as new requirements were added.

Another approach was to build a tailored DBMS for each individual CAD tool. This approach was an expensive undertaking since much software, personnel and time were needed for this type of development. Often these attempts might take so long that the CAD system was no longer in use or had changed so much that the resulting DBMS no longer met the original requirements. RCA [Kore75] began developing a DBMS for its integrated circuit design system and upon completion abandoned the system because of new requirements which had arisen. Indeed, small organizations found the cost factor associated with this approach unduly high. Another problem encountered with this approach was that as CAD systems became more numerous the users not only had to learn to use these different systems, but also had to learn each related DBMS facility.

The third approach taken was to attempt to retrofit commercially available database systems. Numerous implementations used this approach. The major difficulty with this approach was found to be the fitting of design data into the data model supported by the specific DBMS. Other difficulties included the inability to update structures in the generalized data base management system (GDBMS), the inability to process more than a record at a time, and the lack of support for the complex relationships found in a CAD environment [Sidl80]. These difficulties resulted in either no data management or the development of ad-hoc systems.

Many insights were gained from the experiences of these attempts at providing design data management support. The general consensus was that CAD DBMS support needed:

- more functionality than provided by a file management system;
- a greater degree of integration than provided by tool specific DBMSs;
- features attuned specifically toward the design process and the design artifact.

In order to effectively meet these requirements three factors needed to be closely examined:

- the *total* environment into which this system is to be embedded;
- the types of information to be stored and the unique structures needed;
- the manipulations to be performed upon these structures.

The following section looks at the state of current research within the CAD DBMS area. These specific efforts have attempted to meet some of those needs previously cited by addressing particular areas of concern.

1.2 Motivation

The technical community currently exhibits a major interest in the area of CAD (also CAD/CAM). This growth, however, could cause proliferation of accidental systems. That is, software/hardware packages might be put together in an ad-hoc fashion, so as to negate the benefits gained from a CAD system. Proper planning could utilize the pressure of driving forces to arrive at an integrated system [Myer82]. A crucial feature of such an integrated design system is the data management of the both the design and the design process. The data management problem associated with a design is recognized by both the academic and industrial communities as an important area of data base research. Such recognition provides motivation for this particular research.

1.2.1 Recognition of Problem by Industrial Community

Considerable effort has been expended to develop and implement some type of DBMS support for specific CAD systems. This on-going effort is supported by industry, academic institutions and various government agencies such as the Department of Defense (DOD) and the National Aeronautics and Space Administration Agency (NASA). A major concern is the increase of productivity through the application of computers to manage engineering data. Since much of engineering data is generated during the design process, the CAD environment is targeted as an area for research and development. Companies such as RCA [Kore75], Bell Labs [Frie82], Mentor Graphics, Inc [Benn82], Boeing [John82], IBM [Hask82], and Mitel [Blai85] are but a few that have invested considerable resources in order to arrive at an understanding of the features needed and the concomitant problems. Some of the areas of concern are the establishment of requirements for integrated CAD systems [John82], the acceptance of design aids by a user [Kore75], the support of work from design development through manufacturing [Madd81], and the need for a flexible and extensible data base system which will support the changing use of the computer in the CAD/CAM environment [Ulfs81, Tsub81].

1.2.2 Recognition of Problem by Academic Community

Academic researchers have also been investigating such problem areas as: the appropriateness of existing data models for the representation of design [Gutt82, Hask82] , extending and modifying existing data models to accommodate the representation of design [Hayn81] , maintaining the integrity of the design process [East81a] , defining the concepts of concurrency and transaction processing within a design environment [Katz84] , and increasing the scope of data base management systems to more fully support the design process [Katz85, Brow83]. Data base systems to support the design process have been designed and developed at such institutions as Carnegie-Mellon [East80], Stanford [Beet82, Brow83], Wisconsin [Chou82], Berkeley [Gutt82], USC [Afsa85a], and UCLA [Mars83].

Chapter Two of this dissertation contains detailed descriptions of these and other approaches. Each description presents the salient features of the specific contributions claimed by the particular research effort. Classification of these research efforts into three major categories provides the organizational structure of Chapter Two. Using this classification scheme as a basis, some overall limitations with these approaches are presented.

1.2.3 Statement of Research

The intent of this research was to investigate the problem domain associated with the design and development of data management support of the computer aided design process. This investigation has resulted in the identification of particular features needed and a definition of the problems encountered while attempting to provide such features. Solutions to these problems have been proposed. First however, *the data management problem* needs to be explicitly defined. A closer examination of the three factors listed in section 1.1 helps to formulate a definition of this data management problem.

1.3 Design Management Problem

1.3.1 Introduction

The design management problem is defined within the context of an integrated CAD system. The development of such an integrated system - one which interactively provides a structured and consistent data representation combined with user-friendly human-machine interaction - is at best, difficult. This is not the topic of this research. However, this research does take place within the confines of such an integrated system and the development of that system is the topic of a companion dissertation [Worl86]. In order to provide an appropriate definition, we need to have an understanding of both the design process and the CAD system into which data management facilities are to be embedded. Next, we must examine the various types of information that are to be represented and the ways they can be represented. This information representation needs to be mapped onto appropriate data structures and operations for manipulation of those data structures must then be defined.

1.3.2 Design Process Support

A top-down refinement design procedure uses a hierarchical approach whereby systems are decomposed into subsystems and each of these subsystems is again decomposed into subsystems, and so on. At any point in the design process a previously designed subsystem might be usable within the new system design. When all of the decompositions are so realized, the design process reaches completion. Thus, the design process is describable as iterative and recursive. Alternative decompositions may result in different subsystems, giving rise to alternate designs. The design process may also require different representations to evaluate a design's multiple performances [East81b]. During the design process, decisions need to be made based on both design criteria and other dependencies which must be satisfied.

Although we tend to think of creation of a design as being carried out in a vacuum, the process, in many instances, is not an individual task. Complex design problems require many designers working at the same level of design and also at multiple levels. Coordination of the design efforts becomes a managerial task. Documentation of the design and its history are essential aids in achieving such coordination.

The DBMS support needed for such a design environment is considerable. A design data base system must be able to *represent the iterative and recursive* nature of the design. The capability to *retrieve completed subsystem designs based on attributes* is necessary to support composition in design. The *multiple versions of a design* need to be managed by such a system, as well as the *different representations*. This gives rise to complications resulting from maintaining relationships among these various versions and representations. In order to support decision making within the design process it is apparent that *an evolving design knowledge base needs to be* *maintained.*

Support of the collaborative efforts of a team require that the DBMS provide for *configuration control, access control, and report generation.* The ability to *cluster groups of information* created by the design process during various phases is also essential in order to efficiently respond to the designer's needs.

1.3.3 System Environment Interaction

The design process described above is not isolated. That is, it takes place within some system's environment. This environment has been defined to be an integrated CAD system. An integrated CAD system is one which provides a collection of tools to aid in the design process, and a basic subsystem (SYS/KERNEL) which handles the interaction between the user and the tools. Therefore, the DBMS embedded in such a system must provide support for *both* the tools and the SYS/KERNEL.

A data base system developed for the management of design must reflect the structure of the CAD system in which it is embedded. This support consists of providing *facilities for defining, creating and maintaining various types of data structures* needed for the proper functioning of the SYS/KERNEL and tools. In an integrated environment, data structures needed by one tool may also be used in another. Therefore, *maintenance of the integrity and consistency of the data structure* is of paramount importance in achieving an acceptable CAD environment.

1.3.4 Information Representation

A major difference between conventional design and CAD is that in CAD the design information needs to be represented in a formal way. An information model or schema is the formalized way of representing such information. The traditional DBMS solution is that the schema is explicitly stored as an integral part of the data base itself. A conceptual schema of a design process will necessarily have to be represented on two levels.

The development of an information model to represent the design process consists of *identification of the design entities, classification of the entities* into classes or types, and *identification of relationships* between entities and/or classes of entities. In a CAD environment there will be many types and classes of entities and relationships between entities will be complex. Unlike entities found in a traditional database, entity types in this environment can consist of both *structured and unstructured* information.

1.3.5 Data Structure Representation

Transformation of the information model into a structure supported by a DBMS is necessary in order to realize a data management system. Currently DBMS software supports three major ways of building data structures: network structure, hierarchical structure, and relational structure. These basic structures are static, that is, they are defined and do not change their original structure unless the entire data base is reorganized. Since the information within a design environment evolves, the static nature of data base structures appears to constrain representation of the design process.

Representing the design process within a hierarchical data structure might seem to be the appropriate choice since we have described this process as being hierarchical in nature. However, the design process generates a design which consists of m to n relationships more appropriately modeled by a network data structure.

Using a predefined, fixed data structure is usually inadequate in a design database since *many types of data structures need to coexist.*

1.3.6 Data Structure Operations

In the design environment, a designer is concerned with manipulating the representations of the design. These manipulations are performed either upon the entire design or upon individual portions of the design. A design is a complex entity consisting of multiple entities and relationships and therefore, the data structure operations need to handle various levels of complex objects. Present day DBMS operations are defined so as to only perform record at-a-time processing. We refer to this as atomic processing. The complexity of a design object necessitates a type of molecular processing so that the designer might manipulate meaningful design entities.

1.3.7 Summary of Design Problem

We have seen that CAD systems need something which incorporates both data management functionality and some form of software development control techniques to support the design process. Such a system needs to be more flexible than the available data base management systems, so as to represent the design process and to support the various tools that aid in the design process. Besides providing flexible data structures, this system must provide practical solutions to the inherent problem of data redundancy and dependency found in a design environment. The concept of a design environment and its problem domains will be treated in the following chapters.

The DBMS system needed is by far more robust and complex than present day DBMSs. The cost of design and implementation is considerable. In order to ameliorate this cost factor, integration of existing software into the system is demanded. The resulting CAD DBMS should become an integral part of the entire system.

Before proceeding with a presentation of the research hypothesis, we digress in order to present the following sections briefly defining the basic concepts and terminology associated with a GDBMS. These terms will be referred to frequently throughout the remainder of this dissertation and the concepts will be used as a basis for the work presented in the later chapters. The reader who is familiar with the basics of GDBMS technology may skip these sections.

1.4 Data Base Management Systems

1.4.1 Definition

A GDBMS is defined to be that system which stores and manages information pertaining to an enterprise. This system consists primarily of data, software, and users. Hardware also is a part of a GDBMS. The data found in such a system are the objects of the enterprise that it supports. These data base objects are entities and the relationships among these entities. The software are those programs and utilities which are necessary for the proper functioning of the system. The user population is not just the end user, but the application programs, the application programmer and the system programmer (sometimes known as the DBA). The hardware of major concern to the system are those devices upon which the data is stored, and the particular computer which hosts the DBMS.

1.4.2 Architecture

Figure 1.1 presents a simple three-level architecture of a GDBMS.

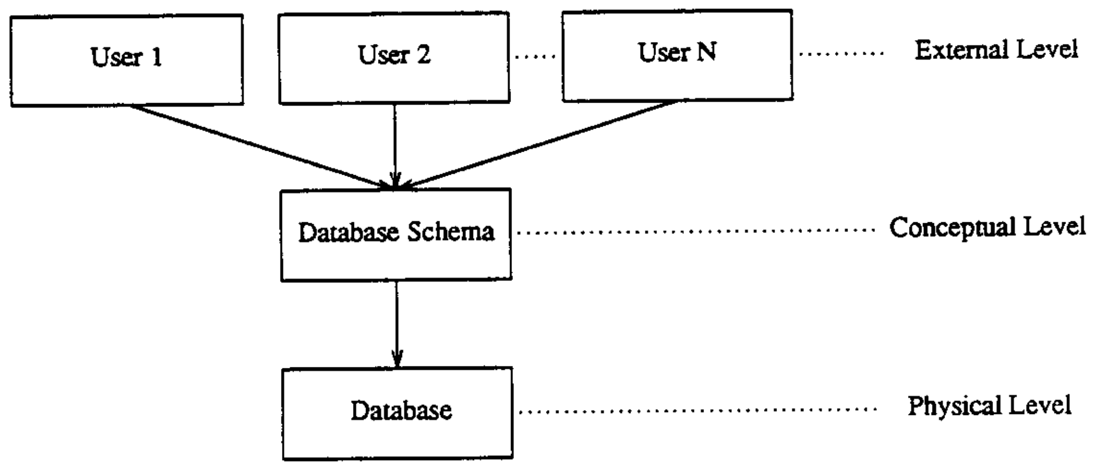


Figure 1.1 Three Level Database Architecture

Level 1, the external level, is the view of the data base which each end-user has. As the figure indicates, there can be many different views of the same data base, i.e., a single data base is capable of supporting diverse usage. Level 3, the physical level, is the organization of the data on the physical devices. This data can be organized using many of the available standard storage techniques such as indexed sequential records or hash tables. Level 2, the conceptual level, serves as the interface between the other two levels. This level supports the logical model of the entire database. The distinction between storage and conceptual level is important, for it permits the data base of an enterprise to be designed without regard for the requirements of the computer system. The distinction between the conceptual and external levels is necessary in order to support the many applications.

1.4.2.1 Logical Models

Most GDBMSs support one of three traditional data models. A data model is characterized by a set of constraints for defining data structures and a set of operators for manipulating the data structures. These data models provide the designer of a data base a means of developing the data base at the conceptual level. The models are the hierarchical data model, the network data model and the relational data model.

The *hierarchical model* is a collection of disjoint trees with records as nodes. The trees are constructed according to connections between records. These connections form an ordered tree called a hierarchical data tree. Each connection, or link, is functional in one direction. That is, for every R(j) record occurrence there is exactly one R(i) record occurrence connected to it if the R(i) record type is the parent record type of R(j). This restricts relationships between records to be 1:M and not N:M. The operators which manipulate the data supported by this model are procedural in nature. That is, the data model must be navigated in order to process any data.

The *network data model* also consists of record types and links. Formally, a link defines a connection between two record types. That is, for each record x of type R(i) the link identifies a set of records of type R(j) which are connected to x. In the opposite direction for each record y of type R(j) the link identifies a set of records of type R(i) connected to y. Links can be 1:1, 1:N or N:M. This model also requires navigational type operators for manipulation of data.

The *relational data model* is one in which data is stored in any number of two-dimensional relations or tables, each with a fixed number of columns, attributes, corresponding to a particular data item. Tables can be compared or contrasted to each other using a relational algebra or calculus. A relational algebra consists of the operators SELECT, PROJECT, and JOIN. These operators allow for the manipulation of data without requiring the predefinition of access paths.

The relational data model is considered somewhat richer than the other two models, in that more detail can be introduced in the definition of the structure. Relational theory of decomposition and normalization provides for analyzing the fine structure of the data. However, the relational model falls short of allowing a direct and precise representation of complex objects. A number of models have been proposed which retain the implementation independence of the relational model but provide a richer means of describing the structure of the data. These models collectively are called semantic data models. One such model, the Relational Model Tasmania (RM/T) [Codd79] is a direct outgrowth of the relational model. In response to the criticism that the relational model lacked semantic expressiveness, Codd extended the relational model to capture more meaning. This extension included the addition of abstraction capabilities along with relational operators for these capabilities. Chapter Six will describe this model in greater detail.

1.4.3 Objectives

For the past twenty years data base technology has been evolving and has reached a point where there is general agreement as to the objectives of a DBMS. The objectives of such a system are many and have been defined and refined by numerous committees and individuals. The following list [Snug79] describes the major objectives of a DBMS.

1. Data independence: allowing data formats to be changed without affecting existing application programs.

2. Data relatability: guaranteeing the automatic propagation of perturbations made at any level of data representation. Propagation of perturbations is necessary in situations where redundant copies of names are kept, additional information is added to a record type, or the system storage is changed from indexed sequential to hash storage.

3. Compatibility with the existing data management techniques found in the enterprise. This is needed, so that the data base system causes little reorganization of procedures which have proven effective within the enterprise.

4. Compatibility with technological developments so that the system does not become outdated too quickly.

5. Structural adaptability: allowing the system to respond to performance requirements and changes in the physical and logical structure of a data base. Such changes may be the result of shifts in query patterns or the rapid growth of data collection.

6. Data integrity: preventing the garbling of data values and organizations. Input errors, hardware failures, software bugs, and shared access all have potential to invalidate the data integrity.

7. Data recoverability: providing the capability to restore the data base to correct contents once an error has been detected.

8. Data security: provision for selective access controls based upon data sensitivity, requestor right to access, or need to access. This is achieved by clustering and isolating security responsibility and using a secure operating system.

1.4.4 Performance Considerations

Besides meeting the objectives listed in the previous section, certain considerations with respect to performance are regarded when designing or acquiring a DBMS. Among those considerations are resource utilization, time usage, space usage, response time for query and update activity, ease of use and application transiency.

Resource utilization is important with regard to the other activities that are concurrently carried out in the computer environment. Some DBMSs require heavy usage of the I/O channels, thus causing a probable bottleneck in the enterprises daily activities.

Time usage is another important factor to consider since the DBMS may need extensive time devoted to programming, or on the other hand may take considerable time when processing the data base.

Space usage varies widely among data base systems. Space usage needs to be regarded from two aspects: the amount of secondary storage required for the data, and the amount of primary memory needed to run programs.

The transiency measurement of a system is the rate at which the data base tends to be disorganized. Data base disorganization is costly in that the database must be restructured. During such restructuring the data base is not available for use.

Users of the system are most interested in the response time for query and update activity. This measurement greatly affects the usage, and hence the value, of such a management system. The final important performance issue is the ease of use of the system. Systems which are complex are often avoided by the very population which they are meant to serve.

1.4.5 Features

DBMSs provide many features to the end-user other than that of the usual storage and retrieval of data. Some of the more important features found in these systems are those that provide for performance optimization, concurrent usage, and data protection.

Facilities are found which enable one to evaluate performance and tune the data base. These facilities include the ability to capture statistics on the usage of the system as well as the ability to restructure the physical organization of the entire data base.

Concurrent usage of a data base system requires the complex facilities to allow the use of the data base by more than one application program at the same time. It is possible in a shared (multiuser) system for applications which execute concurrently to interfere with one another in such a way as to produce results that are not correct. Some sort of concurrency control mechanism is needed in order to avoid such problems.

Data protection features must address two aspects of security. First, a facility must be provided which guards against the loss or damage of data in the data base. Secondly, another facility provides for the protection of the confidentiality of the data from unauthorized persons.

1.4.6 Data Base Software

There are three major categories of software needed to support a GDBMS. These categories consist of the languages, utilities, and operational routines

Data base languages must include a language which allows for the manipulation of the data. This is referred to as the Data Manipulation Language (DML). Frequently, a completely different language is used in order to describe the structure of the data base. This Data Definition Language (DDL) produces what is called the data base schema. A third language, used primarily by the data base administrator (DBA), allows the storage structures found on the physical devices to be described. This language is the Data Strategy Description Language(DSL)

The second category of software includes such standard utilities as dump, edit and print routines. Utilities commonly associated with operating systems are also found in this category. These utilities include load routines, garbage-collection routines, reallocation routines, file-conversion routines, and audit routines.

The last category of software found in a data base environment is the set of operational routines. Such routines might include but are not limited to concurrency control routines, access control routines, data validation routines, data update routines, data access routines, recovery routines, and statistics-collection routines.

1.4.7 Benefits of GDBMS Use

The perceived benefits from use of a GDBMS are centralized control over the data, ability to modify data without major software impact, and support of a multi-user environment.

Centralized control over data affects both the users of the system and the data found in the system. Application programmers can be relieved from standard data management tasks and also need not concern themselves with performance aspects of such a system. Data standards can be enforced easily; data redundancy can be reduced; data can be shared; data integrity can be maintained; data inconsistency can be reduced; and data security restrictions can be easily enforced.

Modification of data without major software impact provides the enterprise with a flexible and somewhat extensible system. As the requirements of the users change or as new data is acquired, changes can be made so as to accommodate these changes and at the same time not disrupt current usage. This capability saves undue costs due to complete redesign efforts.

A multi-user environment, if properly implemented, maximizes the efficiency of the organization and thus offsets the high costs of acquiring a complex system.

1.4.8 Disadvantages of GDBMS Use

The two major disadvantages in use of a GDBMS are cost and compromise.

Cost is assessed both directly and indirectly. Direct costs include the acquisition price, which can be formidable, and also the person-power costs. Acquisition prices are dependent on the particular GDBMS and the modifications needed to bring the data base system into the environment. GDBMSs require many person hours for application programming, data base maintenance and data base use. The maintenance alone is mind-boggling since the amount of underlying software is vast and quite complex. This is not to minmize the cost of maintaining the hardware needed to support this system.

Compromise is necessitated when the provided features do not *exactly* fit the requirements of the enterprise which is acquiring the system. Such compromise might result in some activities being done manually, thus negating the positive aspects of using a data base system.

1.4.9 Design of the Data Base

The design of a data base begins by specifying the data base requirements. This specification involves a determination of what entities are to be stored, what the relationships between the entities are, how this information structure can be modeled in terms of the supported data model, what the operations are which are to be performed on the data structure, who the potential users are, and how frequently the different data management operations will occur.

A design method is employed in the development of a database. A method is characterized by the design technique it uses and the sequences in which it applies these techniques. Thus a method is iterative in nature. A method employed most frequently is first to ascertain an initial design. This initial design uses a set of rules to convert enterprise object sets to data model structures.

A data model conceived to facilitate this initial step in data base design is the Entity-Relationship Model. This model provides for the specification of an enterprise schema and is the documentation of the logical parts of the data base. The following section defines this model in greater detail.

1.4.9.1 Entity-Relationship Model

This section defines the Entity-Relationship Model (ERM) [Chen77] and associated terms. Chapter Five describes the augments defined in this work, needed to support a design environment.

The ERM consists of entities and the relationships among them. An *entity is a thing which can be distinctly identified.* A *relationship is an association among entities.* If e denotes an entity then E represents the entity sets into which these groups of entities are placed. There is a predicate associated with each entity set to test whether an entity belongs to it.

A relationship set is a mathematical relation among n entities, each taken from an entity set and each tuple of entities representing a relationship.

$$RT = [e_1, e_2, \ldots, e_n]: e_1 \varepsilon E_1,\ e_2 \varepsilon E_2, \ldots,\ e_n \varepsilon E_n$$

There is also a predicate associated with each relationship set.

The *role* of an entity is the function that it performs in the relationship. The information about an entity or a relationship is expressed by a set of attribute-value pairs. An attribute can be formally defined as a function which maps from an entity set or a relationship set into a value set or a Cartesian product of value sets;

$$f : E_i \text{ or } RT_i \rightarrow V_i \text{ or } V_{i_1} \text{ x } V_{i_2} \text{ x..x } V_{i_n}$$

These values are classified into different value sets. There is a predicate associated with each value set to test whether a value belongs to it. A value in one value set may be equivalent to another value in a different value set. Relationships as well as entities may have attributes. The concept of a relationship attribute is important in understanding the semantics of data and in determining the functional dependencies among data.

Identification of an entity is dependent upon an entity key. An entity key is a group of attributes such that the mapping from the entity set to the corresponding group of value sets is one-to-one. Relationships are identified by the involved entities. Thus, the relationship key consists of the keys of the involved entities. In certain cases, the entities in an entity set can only be identified by the entity(s) that participate in a relationship with them. Such entities are called weak entities or dependent entities. Similarly, if any entities in a relationship are identified by other relationships then the relationship is called a weak relationship.

Several important characteristics about relationships and entities in general are:

1. A relationship can be defined on more than two entity sets.

2. A relationship may be defined on only one entity set.

3. There may be more than one relationship set defined on given entity sets.

4. The relationship set mapping may be 1:n, m:n, or 1:1.

Chen also introduced a diagrammatic technique for exhibiting entities and relationships - the entity-relationship diagram (ERD). The symbols used in this ERD are a rectangular box, a diamond shaped box, an oval, lines, and text. Figure 1.2 illustrates the ERD representation depicting the entity sets WARD and PATIENTS related via a relationship set labelled OCCUPANCY. Each entity set is represented by a rectangular box and each relationship set is represented by a diamond shaped box. Roles can placed on the lines connecting the boxes. In Figure 1.2 the role attached to the PATIENTS entity set, *is assigned* and the role attached to WARD, *assigned,*, are used to indicate further descriptive information about the entity sets

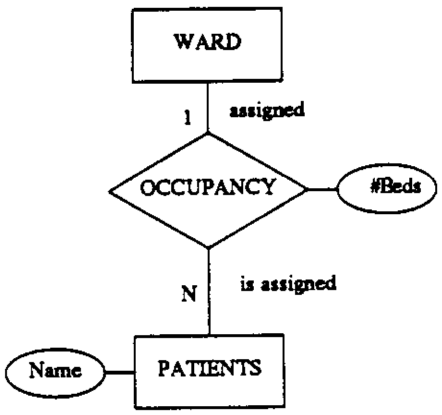


Figure 1.2 WARD - PATIENTS ERD

participating in the relationship OCCUPANCY. These lines are also used to indicate the relationships in which particular entities take place. Figure 1.2 indicates that there are many PATIENTS (N) associated with one (1) WARD. An oval is used to represent attributes defined for the entity or relationship set. The oval with *Name* enclosed within it represents a name attribute associated with PATIENTS, and the oval with *#Beds* within it represents that the relationship set OCCUPANCY has an attribute *number of beds* associated with it.

Chen concluded his definition of the ERM with a discussion of how this model could easily be mapped to the traditional models: relational, network, and hierarchical. This ability to map to, in particular, the more formal relational model is the major reason for the attractiveness in the use of the ERM and its ERD as a preliminary model for database design.

1.4.10 Development of the Data Base

After arriving at an initial design, the development of the data base begins by analyzing this design using selected design analysis techniques. Such techniques would possibly include the estimatation of storage requirements, the estimation of average transaction response time, and the average length of access paths. Certain criteria, determined by the data base environment, are applied to arrive at this estimate. Examples of such environmental criteria might be: limited secondary storage, restrictions that transactions occur only in an interactive environment, concurrent access of up to fifty users. As a result of this analysis, the design is amended or the requirements are modified.

The development of a database must satisfy other criteria as well. The supporting logical and physical structures must preserve the data correspondence to avoid data redundancy and anomalies.

The ultimate goal of the database design methodology used during the development phase is to produce a feasible design structure that satisfies all data access requirements of the enterprise.

1.4.11 Use in Design Environment

Databases found in a design environment are characterized by models of complex reality. The database system must not only provide primitive access to these models, but also must provide for the support of the design process which produces these models. These characteristics highlight the differences between the use of databases in commercial environments and the use in computer-aided design environments.

In CAD environments the information evolves with use of the various tools requiring that the conceptual schema be dynamic in nature. In commercial environments there are frequent changes to the values found in the database, whereas in CAD environments the changes are less frequent but involve both values and structures. We find also that the structures in CAD environments include graphics, as well as non homogeneous and recursive entities. Unlike commercial environments, the end-users are familiar with the application environment, but are not likely to have expertise with databases. Yet these end-users are the evolvers of the database. The unique situation found in CAD environments is that both definition and manipulation of the database occur simultaneously.

The complexity of a CAD data management facility is such that careful consideration must be given to the design of this facility. Merely incorporating a commercial DBMS into the CAD environment would serve only to add more complexity to the design process. The special needs addressed in the preceding sections lead us to propose an approach toward meeting such needs.

1.5 Research Hypothesis

One approach toward meeting the data management needs of a CAD environment is to design certain design-specific management facilities into the SYS/KERNEL. The primary hypothesis of this research, then, is that design knowledge base primitives can be introduced into a CAD system KERNEL such that it may be more powerfully extended than otherwise. Such a KERNEL would not only provide for the management of design data, but would also provide the basis for use of intelligent design tools in capturing design expertise and design histories. The investigation of this hypothesis and supporting hypotheses will be carried out using the SARA/IDEAS methodology and system as a testbed. Supporting hypotheses are that:

- Extensible data definition facilities can be developed for use by both design tools and the CAD system designer;
- Extensible and flexible data structures can be developed or existing structures modified to provide required data management support for a design environment;
- An extensible and flexible data model can be developed to support the acquisition and management of design knowledge;
- A data intensive tool can be developed to illustrate the power and flexibility of the data base design;
- Integration of design tools can be achieved through the use of this KERNEL;
- A design methodology for an integrated design information system can be developed.

1.6 Research Goals and Contributions

This research has several goals pertaining to the field of data base management support of the design process. These goals encompass the areas of design, data modeling, knowledge acquisition and representation, programming environments, and software engineering. The results of this research are intended to perhaps provide an environment in which the process of design can flourish, thus allowing for increased productivity at reduced cost. The quantities, productivity and cost, will not be directly measured. However, the quality of the environment in which design takes place will be analyzed.

Specific contributions of this work include:

- definition of a data model to support the design process;
- development of a data definition facility for defining design models;
- implementation of a design data base system within the IDEAS environment;
- implementation of a design tool - the system browser ;
- development of an environment in which research in the areas of data base technology and artificial intelligence can be carried out;
- integration of existing software tools such as RCS and Troll into the design data base system.

1.7 Dissertation Overview

The organization of this dissertation is such that the concepts and methods used in completing the research work are presented to the reader. Chapter One served as an introduction to this research by defining the design management problem, describing the fundamental concepts of management systems, and then presenting an approach to be used as a solution to the design management problem. Chapter Two contains a partial summary of the literature research that was carried out.

Various research and commercial efforts are described and analyzed in this chapter along with descriptions of numerous programming environments. The selection of a particular research effort for inclusion in this dissertation was based on the degree that the work influenced this research effort.

In order to understand the environment in which we are embedding data management support, Chapter Three characterizes the various types of CAD systems and then describes two such systems in detail. SARA, System's ARchitect Apprentice is detailed so as to provide a reference for the description of an integrated design environment, IDEAS (Integrated Design Environment for Analyzable Systems), within which this research takes place. A description of the data management facilities designed for this environment is presented in Chapter Four. The architecture of CADIS (Computer Aided Design Information Systems) is found in the beginning of that chapter. The concluding sections of Chapter Four describe the four major levels of the architecture.

The remaining chapters of this dissertation then present the various components of CADIS. Chapter Five describes the kernel of CADIS. The information model supported by CADIS is defined along with operations to build and manipulate this model. Chapter Six defines and describes the automated data definition facility provided by CADIS. The underlying implementation model is presented in that chapter. Chapter Seven continues with a presentation of the system level of CADIS. The introduction of a data intensive tool, a system browser, is defined in Chapter Eight. This tool calls upon the primitives provided by CADIS so that it may use objects which have previously been defined for other tools. Chapter Nine concludes the dissertation with an analysis of the results of the research and a comprehensive discussion of future research resulting from the efforts of this work.

CHAPTER 2

Related Approaches

2.1 Introduction

Current approaches for supporting the design process and the management of design data are surveyed in Section 2.2 - 2.4 of this chapter. These approaches are classified according to one of three major categories: those approaches which use commercially available data base systems: those which develop special application CAD data base systems: and those approaches which integrate data base management techniques with interactive dialogue functions to provide a wider range of design support. Within each of these categories further classification can be made as to the level of design which the CAD system supports. Many CAD systems primarily consist of one or more design tools operating at a lower level in the design chain (figure 2.1), such as circuit level design or printed circuit board placement and routing, whereas other systems support the higher levels of design such as requirements definition or behavior and structure modeling.

Each of the approaches described emphasizes different facets found in data management facilities. Therefore, the focus of their work is restricted to that area within data base technology which best provides the support so desired.

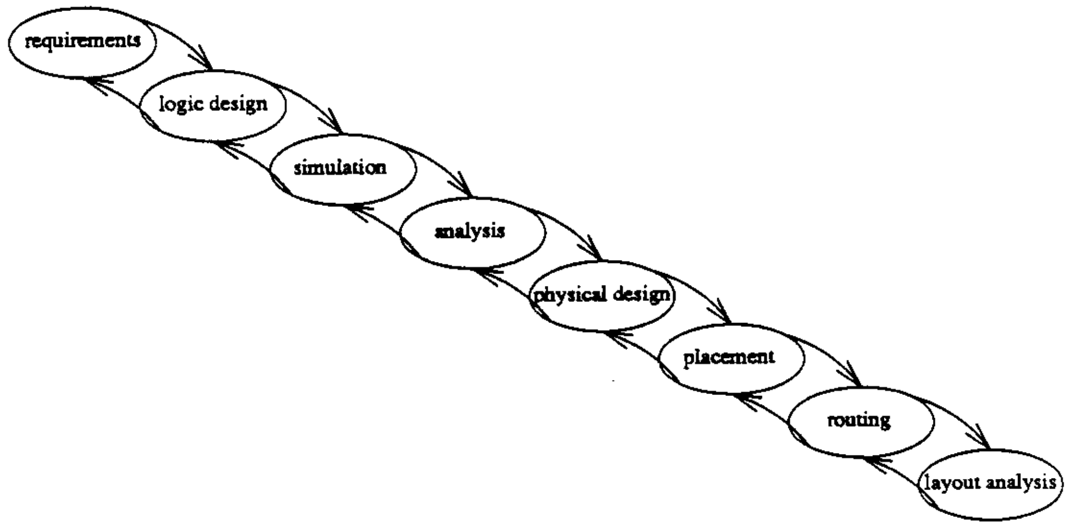


Figure 2.1 The Design Chain

Within data base technology, areas of interest to the various approaches are:

1. appropriate structures for design data

2. appropriate low level access to data

3. support for unstructured data

4. appropriateness of traditional data models to support design management

5. appropriate language to support design data base access and operations

6. support of a 'component' library

7. integration of existing tools through the use of a common data base

8. provision of version control, concurrency control,

configuration management

9. maintenance of design integrity throughout the various design representations

10. provision of a knowledge base of design procedures

It should be noted in the following descriptions that each approach concerns itself with some subset of the above items. We also point out that some approaches are entirely hypothetical, while others base their work on synthetic data, and still others are actual working implementations existing in either academic or commercial environments.

In addition to discussing data base systems which support CAD systems, data base systems which support specific programming environments are described in Section 2.5. Concepts used to build programming environments are important within this work in that CAD systems are an *environment* within which the *programmer (designer)* of a design functions. The concepts used in building a programming environment are those same concepts on which CAD systems are built.

2.2 Commercial Data Base Systems

2.2.1 System R

Work at IBM Research Labs in San Jose, CA includes the modification of System R to improve its ability to handle design data [Hask82]. The four major areas of concern are: the handling of complex design objects, the support of conversational transactions, the interface between the data base and data structure, and the management of data items of unlimited length.

The ability to handle complex objects (figure 2.2), which is often found in the design environment, is enhanced by allowing a designer to declare and specify structural relationships among semantically related data.

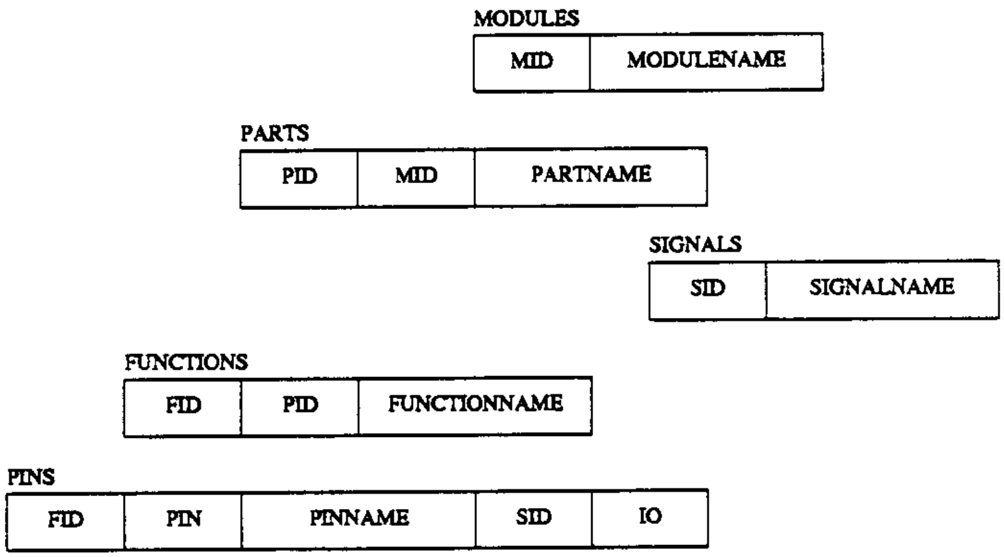


Figure 2.2 Example Complex Object Schema

Such complex objects might be the representation of a chip with its functions, components, pins, etc. Support for attribute types such as *identifier, component_of,* and *reference* allows for the object's structure to be defined to the system.

Since design transactions are highly interactive and may span long periods of time, the concept of a conversational transaction is introduced. Two problems addressed within this context are that of backout policy and deadlock avoidance policy. These policies differ radically from those found in a business data processing environment. In order to handle these design conversational transactions, the capability to lock a complex object is added to System R.

Modifications also provide an object oriented interface which allow a data structure to be built from one or more complex objects. System R's facilities for managing non-coded data are being modified to enhance performance on updates and to support items of unlimited length.

These extensions to System R have allowed work to proceed in the management of data for a large internal design system. This modified System R could be used to support design at all levels within the design chain.

IBM's development of a capability which allows a designer to handle complex design objects is particularly relevant to the work of this author. We also identified this area as one of major concern when managing design data and will later present another approach to developing a capability to handle complex objects.

2.2.2 INGRES

Researchers at Berkeley have been experimenting with implementation of a CAD application using INGRES, a relational data base management system [Gutt82]. Their work has identified four features which need to be added to INGRES in order to more fully support CAD applications. These features include: facilities for the support of ragged relations; support of transitive closure; access by spatial location; and the generation of unique identifiers.

Often in the design process the need for multiple values associated with an attribute arises and is best represented in relations as repeating fields. Ragged relations are needed to allow for repeating fields within a relation. The approach to support ragged relations is to allow for ordered relations and then allow the relations to be nested.

Transitive closure support is necessary so that the hierarchical nature of design can be exploited. Transitive closure is applied during the expansion of subcells within a cell. That is, if a description of a cell includes other cells within it, then the tuple which describes the top cell will contain a reference to each subcell. These subcells are also described by tuples. Transitive closure assures that all the references to other tuples are expanded until no more tuple references are found.

Many CAD programs need to retrieve design data according to its spatial location. Implementation of this capability is achieved through the use of spatial bins which are an extension of the secondary indexing facility of INGRES. These bins identify a window in which portions of a design reside.

The usefulness of unique identifiers has been suggested by many within the data processing environment. The system generation of such identifiers is crucial within a CAD environment so that the specific applications are not forced to manage their own identifiers. This automatic generation by the database system would be handled by supporting a special type - ID.

These enhancements to INGRES, the researchers claim, allow for a data base management system which would be easy to use and perform at acceptable levels within a CAD environment. INGRES could be used as data management support at both high and low levels of design.

The use of system generated identifiers, as well as the use of a commercially available relational data base management system is seen to influence the work of this author.

2.2.3 ADABMS

Systems Architect Apprentice (SARA) is a design system developed at UCLA. SARA supports both the top-down partitioning and bottom-up composition procedures for the design of hardware and software digital systems. Since the SARA/IDEAS system is being used as a test-bed of this research, Chapter Three contains a detailed description of the methodology and system. This section will focus primarily on the data base aspects of SARA. The current system recently acquired a library system developed by J. Marshall [Mars83]. This SARA Library system uses a commercially available data base system, A Data Base Management System(ADABMS), which is based on the CODASYL model. The major areas of concern while developing this library support were: storage structures for design specific models, support for various types of access capability, and the creation and use of a "building block model".

The storage structure (figure 2.5) chosen is based on a hierarchical schema, although the data base system used supports a network type model.

The reason for this choice is that this schema more closely represents the recommended organization of the various design models generated by the different SARA design tools. The schema consists of: an implementation set(B/B set); SL1 set; GMB set; interpretation set; test set; and driver set. The owner of the implementation set is the system record and its member is the B/B record type. This B/B record type is the owner of the other sets found in the schema. The B/B record contains information as

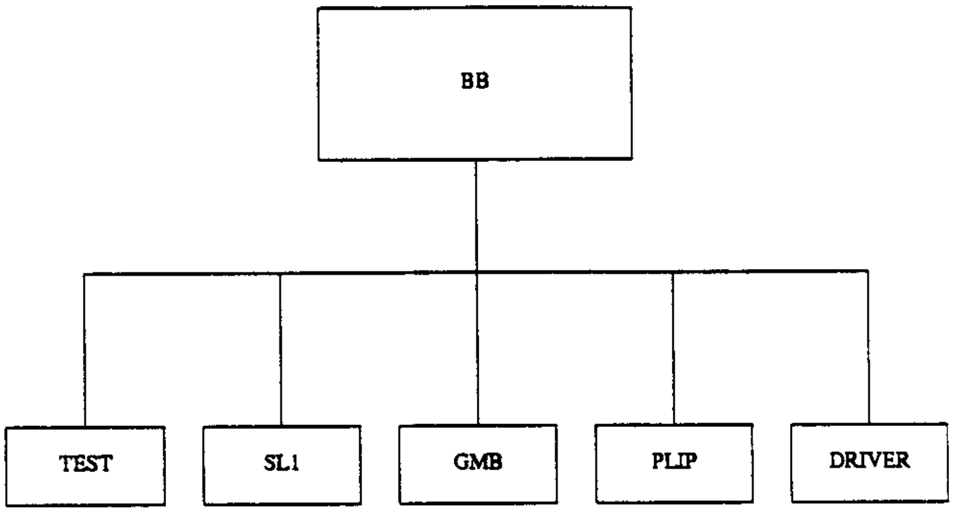


Figure 2.5 SARA Building Block Library Schema

to the specifics of the particular building block, such as, the design's generic name, synonyms for the name, the version number, the refinement level, descriptive information and classification information. The other record types contain the source or object code representation of the respective SARA model.

Access to a building block is through the implementation set. Once the users have positioned themselves at the implementation set of the building block required , then any of the representations (SL1,GMB,etc.) can be retrieved. The unit of access is at the level of a record, as is found in commercial data base systems. However, a record in the SARA library is a complete representation of one of the models found in the SARA system. In order to store a particular design, the user has to first create an implementation set and use this as a reference point.

The Building Block Library found in SARA is a database, either private or public, into which designers may place the design representations. During a design session the designer first opens a database and specifically creates an implementation set, defining the necessary parameters for this set. In order to use a building block, the designer specifies which implementation set to find , retrieves the appropriate model, and invokes one of the SARA tools.

The SARA Building Block Library does not concern itself with providing total data management support for the design process. Requirements for the new SARA/IDEAS system state that there should be support for all aspects of the design (see Chapter Three). This more ambitious undertaking is currently being implemented at UCLA. The research begun by Marshall served as a basis of this author's work.

2.3 Special Application CAD Data Base Systems

2.3.1 TORNADO

Technical ORiented Network Data Organization (TORNADO) is a data base management system developed for specific CAD/CAM application systems, in particular, the Scandinavian CAD project GPM (geometric product models) [Ulfs81]. The major areas of concern in TORNADO are: the creation and handling of complex data structures; the direct handling of many to many relationships; the support of dynamic length table records and variable length objects; and the development of a CAD system which demonstrates very high performance.

TORNADO is a subroutine package which is programmed in standard FORTRAN. The architecture of this package consists of two modules; a data manipulation module (DMM), and a storage administration module (SAM). The two modules are independent of each other to such a degree that other applications can use SAM independent of the DMM. In fact SAM is the general filesystem EASYBAS. The DMM has the capability of working on objects which are represented as sequential EASYBAS records.

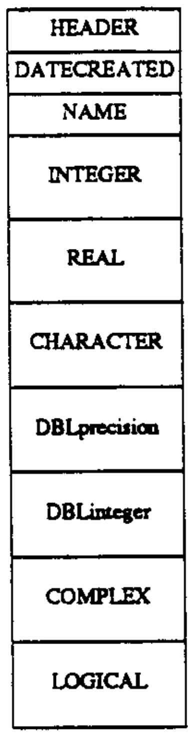


Figure 2.6 TORNADO Object Description

In TORNADO, an object (figure 2.6) is defined to be a logical group of data items which describe an instance of an object class. This object consists of a header, which contains the length of each data area within the object, the object class, and pointers to other objects which participate in the data structure. An object can have up to nine data areas containing different types. The size of these areas is determined either when the object is created or by the schema definition. Standard data types are supported. Direct access to data areas within the object is permitted. Objects may be combined with other objects to form sets and these sets may be combined with other sets or objects with little restriction. Object names are stored in a name table to facilitate finding an object by name. Object operations consist of creating an object, finding an object, storing and retrieving an individual data item in an object, connecting an object to a set, and renaming an object.

A dynamic length table record is a record which contains a table that can be increased or decreased dynamically. The table consists of a number of lines of equal length. Lines may be appended to the table or deleted from the table. Operations to support the creation, deletion and retrieval of records are available to the user through subroutine calls or as interactive commands.

The designers make claim that TORNADO exhibits flexibility as a CAD DBMS because it does not place any restriction on the complexity of the data structure that it can handle. No attempt was made to include recovery mechanisms, concurrency mechanisms or automatic integrity checks, primarily because these features are time consuming and would result in a much poorer performing system. It is meant to be a partial tool to handle the interactive design of complex geometrical objects.

The work which most influenced design decisions made by this author was TORNADO's complex data structure handling mechanism. Such a mechanism is of paramount importance for effectively managing design data.

2.3.2 IPAD

NASA awarded a contract to the Boeing Company to develop Integrated Programs for Aerospace-Vehicle Design (IPAD) [John82]. The major goals of this work were to establish requirements for an integrated computer based system for managing engineering data and to develop software to demonstrate these concepts. The concepts with regard to the data base support were: the ability for user definition of data sets which represents an engineering task; versioning capability for data sets; long term archival features; interactive retrieval and update of data sets; project management capabilities; and a uniform language.

The IPAD Information Processor (IPIP) data base management system is designed to manage the data which evolved over time. This system supports multiple data models(the network and relational model), multiple levels of schemas, concurrency and multiple users , and access in a distributed environment. A single Logical Schema Language(LSL) and Data Manipulation Language(DML) supported both the relational and network data models and the application programs. Composite objects called structures were supported to manage geometry and other scientific data. A structure was defined in the schema to consist of tuples from a tree or network of relations. This structure was manipulated as an entity through operations. These entities were meant to be natural occurrences in the application domain.

IPID was in the early stages of development, so versioning, releasing and archival features were not implemented.

2.4 Integrated Data Base Systems

2.4.1 FLOREAL

Experience with the design and realization of an integrated CAD system, GERMINAL, resulted in a proposal of a CAD model and a CAD system with data base. This proposal concerned primarily the design and implementation of the language FLOREAL, a language which included both data description and data manipulation facilities [Fois81]. The two major topics addressed by FLOREAL were the representation and the manipulation of information in a CAD oriented data base.

Comparing the uses of data bases for administration purposes with the uses of data bases in computer aided design, led to the formulation of requirements for a unique data definition and manipulation language which would support new classes of data and have the ability to refine classes or redefine them. The data model would contain both static and dynamic classes of entities, and would provide a set of powerful and flexible data structures and building mechanisms.

In FLOREAL four concepts are defined: the type, the object, the relation and the function.

A type models static objects by defining their characteristics. This type defines a set of values on which a set of actions operates. A type is defined by a list of properties where a property is the smallest logical item, characterized by a name and a value. Several organizations of properties are simple, compound and functional. A type is a collection containing at least one property. Such structures are: multiple type, structured type, union type, derived type. The ISA defines a types hierarchy. It allows one to define a type from another type. Extended types are built on structured source types and then are structured. Structured types may be constrained. A type is considered to be knowledge which the DBMS must be able to access.

The object is a class of elements used in the building phase. It is similar to a type but it may be modified by the user and is considered to be project information. Descriptions of the same object several times are referred to as views and are named. A view is a set of components and a set of relationships among them. An object may possess global properties.

A relation represents a link between types and/or objects. It is either knowledge needed by the DBMS or project information according to membership of the linked entities.

A function represents a mathematical function. It may be either data base knowledge or project information according to its generality.

The proposal was that the FLOREAL system be built on an available DBMS which manages two specific data bases for a design project. The Knowledge Data Base contained informative type information which may be altered by a user during a design process. Such information may be standard components, design procedures, standard function, design domain and ways to use that knowledge. The Project Data Base contains operational information, which was generated in a continuous way by the user and related to the designed objects(structural and functional descriptions). A set of application programs were to implement a particular CAD data model in terms of the chosen DBMS's primitives. That model consists of a set of concepts and rules handled through the language FLOREAL.

2.4.2 WiSS

The University of Wisconsin has on-going research into the ways of applying database technology to the management of design data. The major areas of concern include: the storage of unstructured data, support for multiple versions and representations of design data, conversational transactions, automation of the maintenance of design consistency, and support of design workstations. These concerns have coalesced into efforts to implement a prototype system which would allow existing tools to be integrated into a single system.

The Wisconsin research team's experience with existing database systems indicated these systems were difficult to modify and not well suited for the types of applications with which the group intended to experiment. Therefore a most ambitious project was the development of a low level system based on System R's RSS [Chou82]. The Wisconsin Storage System(WiSS) provides many of the same facilities as the RSS of System R, but is implemented in the Unix environment. WiSS supports high performance sequential and indexed access to data stored on disk through a relatively low level manipulation language.

One of the major goals of WiSS was to extend data base techniques to unconventional data. Unconventional data is data such as text and raster images which often arise in design environments. The approach taken was to build support for long records that could span physical disk pages. High performance was another desired goal, and so in order to achieve this goal the WiSS designers felt that the UNIX file system needed to be circumvented. It has since been used as the basis for many areas of research, including the development of a prototype system to support the integration of existing design tools for VLSI.

R. Katz [Katz83, Katz82] described the structure of a system which satisfies the needs of design tools, in particular the needs of VLSI design tools. The components of this system, called Engineering Database Management System(EDBMS), were being developed by various members of the Wisconsin research group.

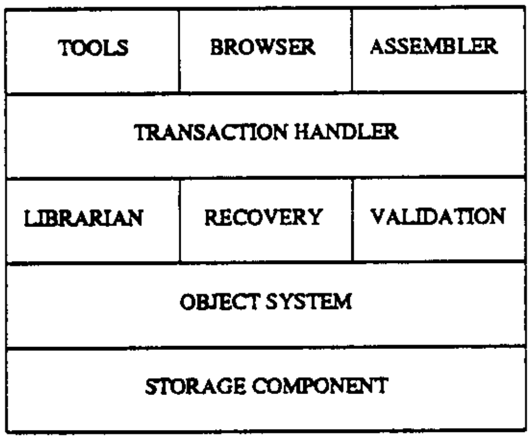


Figure 2.8 EDBMS Architecture

The EDBMS system architecture (figure 2.8) consisted of: a storage component which stores data on disk and guarantees that updates are atomic, an object system which maps design data into files, a recovery system which insures that changes to an object can be reconstructed after a crash, a design librarian which controls access to design objects, a design validation subsystem which interprets dependencies among design data to identify those portions affected by a change, and a design transaction system which ensures that the designers create new consistent versions. The browser/chip assembler is an interactive interface to the design management system.

The storage component of EDBMS is WiSS which was being extended to provide atomic update. The object component maps the abstract notion of an "object" into the descriptive data that are stored in storage component files. Design objects are stored either as a single file or as separate files. If the representation type is not known to the system, then the data are stored in a separate file referenced within the design object file, thus acting as a conventional file system.

The recovery system provides for survival in the face of a system crash by maintaining multiple copies with alternative failure modes. The manager supports save points. Data and change file buffers are forced to disk by the workstation's buffer manager.

The design transaction management system deals with three phases of the design: the work, validation, and completion phases. When a designer requests a design the work phase is entered. Mirrored copies are made to provide redundant copies to be used for recovery. When design work is completed, the transaction enters the validation phase. During this phase programs check that the modified design data are self-consistent, and that all relevant constraints have been enforced before a transaction is allowed to enter completion. During completion a new version is added.

The browser/chip assembler are the interactive interfaces to the design management system. The browser allows a designer to navigate through the complex design structures, making extensive use of graphics to present the data structure and menus to direct the navigation. The chip assembler is the most data-intensive portion of the design task. Subsystems are constructed from component compositions.

The efforts by the Wisconsin research group were directed toward providing design management services to an ensemble of design tools in order to create an environment in which tools are integrated to form a coherent design system. We consider these efforts to be significant in the quest for the realization an integrated CAD system.

2.4.3 ADAM

Researchers at the University of Southern California(USC) [Afsa85b] are currently developing an extensible object-oriented framework suitable for modeling VLSI design environments. The 3 Dimensional Information Space(3DIS) [Afsa85a] is an information management framework intended for applications that have dynamic and complex structures, and whose designers, manipulators, and evolvers are non-database-experts. 3DIS unifies the view and treatment of all kinds of information including the description and classification of data. This model has been developed for use with the Advanced Design AutoMation (ADAM) system. ADAM is intended to provide a unified system for VLSI design, starting with functional and timing specifications and proceeding to circuit layout.

In 3DIS, databases are collections of interrelated objects, where an object represents any identifiable piece of information, of arbitrary kind and level of abstraction. The 3DIS model supports atomic, composite, and type objects. Atomic objects serve as the symbolic identifiers for atomic constants in the database. These objects carry their own information content in their objects-ids. Composite objects describe entities and concepts of application environments. The information content of these objects can be interpreted meaningfully by the 3DIS database through decomposition into further objects. Type objects contain the descriptive and classification information of a database. Every type object is a structural specification of a group of atomic or composite objects. The 3DIS model has been extended to accommodate other kinds of abstraction primitives such as the definition of recursively defined entities.

The 3DIS model also supports a simple and multi-purpose geometric representation. This geometric framework graphically organizes both structural and non-structural database information in a 3-D representation space. Relationships among objects are modeled by triples that represent specific points in the geometric space.

Basic relationships among objects are defined through the three fundamental abstraction mechanisms of classification, aggregation, and generalization. Classification represents member/type relationships by relating atomic/non-atomic object to it generic type object. Aggregation represents member-mapping/type relationships by relating a type object to the mappings that define its members. Generalization represents subtype/supertype relationships by relating a type object to a more general type object.

The fundamental structure of the conceptual schema is the component. A component is described in terms of four models and a set of relationships (bindings) across the models. The models are the dataflow model, which describe the data transformation operations performed by the component, the timing and sequencing model, the structural model and the physical model. All relationships between models are explicitly represented by means of bindings. There are two types of bindings: operations bindings, which relate dataflow elements to structural elements and time ranges, and realization bindings, which relate structural elements to physical elements.

Claims are made that this approach to providing database support results in ease of addition of application packages and ease of access by non-database experts.

2.4.4 GRADAS

The GRADAS system is an approach that concentrates as many functions as possible in an application-independent system, while attempting to minimize the programming work required for implementing a new application [Enca83]. GRADAS provides not only a dialogue module based on a graphics module, but a database and a logics module as well. The logics module knows as much about the semantics of the application as can be specified in an application-independent way, it may be regarded as a subsystem which checks the input against a conceptual schema and context-sensitive semantic restrictions. The major concern related to the management of design data is the storage of graphical and non-graphical data in a uniform way. The designers of GRADAS felt that this required a complex data base management system. This resulted from the conflicting requirements which the DBMS had to meet in order to manage and store both the objects of the user world on one hand and the graphical representations on the other. The need for fast response during interactive CAD system applications called for simple, preferably sequential storage structures. This need was emphasized by the large amount of data that is generally required to produce a display. CAD systems call for the redundant storage of information in forms suited to both the user's complicated structure, and the graphics structure. This approach complicates the modification and deletion of objects and shows all the usual consistency problems of a redundant data representation. An integration of the conventional data base and the graphics file is desirable.

The following is a description of the DBMS design (CORAS) which the designers felt provided a solution to the problems stated above. The basic idea is to give the DBA the possibility of telling the DBMS which pieces of information are critical with respect to the interactive dialogue process. In addition the DBMS is told what other information is likely to be needed together with the piece that has just been retrieved. Thus, time-critical data structures and time-critical references have been defined.

CORAS has simple entities, list entities, relation entities and set entities. Each entity can be accessed by one or several names. Each entity can have attributes that describe it in more detail. Relation,set and list entities may contain entities. CORAS has functions for the generation, production, manipulation and deletion of logical system structures on a basic level. The following are modules above the management of physical storage:

1. The name storage---contains names of entities with reference to the entity data;
2. The data storage;
3. The relation storage--contains tables expressing the relations between

entities in the form of triads "relation,entity1,entity2".

The entity class is attached to the entity name--ID pair. An entity consists of a structure, along with four lists. The first list contains the synonyms of the name; the second list contains user-specific data; the third and fourth lists contain ID-pair references to lists and sets.

CORAS offers the user the required data base functions for conventional application (storage, retrieval, and modification of data). The DBMS deals with procedures as elements as well as data. This means that primitives are be described in procedural form. CORAS, therefore, supports two kinds of data retrieval mechanisms:

1. a normal retrieval where entities are retrieved using information that is stored in the form of data.

2. a retrieval that calls for the application of an expansion procedure.

GRADAS is one of the recent approaches to providing an integrated system within which all types of design processes can be placed.

2.4.5 MITEL

Development of an architecture built on a relational data base to support an engineering design management system for producing printed circuit boards, is a current project at MITEL [Blai85] Corporation in Canada. The primary goal of this project is to produce an integrated system which will facilitate process management.

The architecture consists of three layers: the interface level; the system level; and the database level. Design process constraints are built into the components found in the system level so that the interface level consisting of the applications programs need only be concerned with the design itself. Design process constraints include; update propagation, versioning, and revisions. The system level provides the interface routines to the database so that the designer is removed from the tedious tasks of determining appropriate calls to store the data. The advantages of such an architecture are: the reduction of errors due to processing; the capability to monitor the design progress; the increase in the efficiency and reliability of the design system; and the reduced cost resulting from the use of a commercially available system.

2.5 Programming Environment Systems

Programming support environments (PSE's) are similar to those environments associated with CAD systems. Although most CAD systems are primarily concerned with the design of hardware , both systems are concerned with *objects* , their *relationships* and *manipulations*. Therefore, understanding problems encountered within a PSE and analyzing approaches taken toward solving such problems is useful when developing support environments for CAD systems.

Two systems, UNIX and Interlisp, have made substantial contributions in the area of PSE's [Wass81] and to the general understanding of tools and features that must be provided by a PSE. However, their use of a central information repository is limited. Significant advances of PSE's might be in the use of such a repository. The Department of Defense's publication of the requirements for such a system - ADA Programming Support Environment (APSE) - has as one of the three principal features, a database which acts as the central repository for information associated with each project throughout the project life cycle.

2.5.1 ADA Environments

The Stoneman document [Defe80] specifies the requirements for an ADA Programming Support Environment. The purpose of such an environment is to support the development and maintenance of application software throughout its life cycle. The other two features of an APSE are the (user and system) interface and the toolset. The interface includes the control language which presents an interface to the user as well as system interfaces to the data base and toolset. The toolset includes tools for program development, maintenance and configuration control supported by an APSE. One of the layers in the representation of an APSE is the Kernel ADA Programming Support Environment (KAPSE), which provides data base, communication and run-time support functions to enable the execution of an ADA program. The purpose of the KAPSE is to provide a virtual support environment for ADA programs. A major component of this support environment is the KAPSE data base.

The specific KAPSE database requirements are:

1. Each object in the database shall be uniquely identified;
2. There is no restriction on the format of the information stored in an object;
3. Relationships between objects shall be recorded;
4. Objects shall have attributes:
    a. History attribute
    b. Categorization attribute
    c. Access attribute;

5. There shall be an archiving facility;

6. Access shall be provided to both the information content of objects and attributes and for the traversal of networks formed by relationships;

7. ADA tools can read/write database objects.

These requirements illustrate the expansion of the functionality of a data base support system. The need for a history attribute portends the use of tools which will analyze the process of the design of software so that software design might be better understood. Understanding of the structure of the software design is inferred by requiring that the data base system be capable of traversing networks formed by relationships. That is, more semantics are introduced into the data model.

2.5.1.1 KAPSE Database Implementation

The introduction of ADA, and its environment, led to the development of a Library Manager [Narf84] as the basis for an ADA Programming Support Environment. One objective of the Library Manager (LM) is to give good support for the management of separately compiled ADA units. However, the LM can allow for tools to introduce and retrieve objects and relations between such objects. There is also support for the controlled sharing of objects between programmers. In order to realize this type of support the LM is based on binary functional relations, which form a directed graph. Provisions are made for high level operations for analyzing relationships. There is also the capability to store and retrieve versions of the developing ADA software.

The implementors envision that the programming environment consists of different tools which take different views of the data base while accepting the basic data model and the conventions it provides.

This data model consists of a hierarchy of components. Each component consists of attributes and relations. Attributes describe the component , whereas relations describe the relationships of the component to other components. Attributes can be user-defined or system defined (consisting of a time_stamp). Relations are named properties with values that are collections of references to other components. When there is a need to retrieve information about the structure, operations which perform closure retrieval are used.

2.5.1.2 Arcadia: A Software Development Environment

The Arcadia project [Tayl86] is a research activity which is exploring issues in the areas of environment architectures and software development support tools. This experimental environment is being implemented in Ada.

The environment architecture is intended to reconcile extensibility with the often conflicting goal of integration. A central premise of this project is that Arcadia would appear to the user to be an environment for creating and managing numerous and diverse software objects such as object code, design elements, test data, and graph representations.

The support tools are made up of very small, modular tool fragments; any substantial task involves potentially complex interactions between fragments. The user is shielded from the complexity by allowing the user to describe an object which can be derived from existing objects, leaving it to the system to determine the correct application of tool fragments to produce the desired object.

The objectives and approaches being taken by the Arcadia project are of particular interest to this research effort, since the main approach to the development of the SARA/IDEAS system is similar to that used in the development of the Arcadia system.

2.5.2 Smalltalk Environment

In the early 1970s, at the Xerox Palo Alto Research Center, the Learning Research Group embarked upon the ambitious Smalltalk project. The proposed project had as its goal the design, test, and implementation of abstract notions conceived within the human mind and represented in computer hardware. The interface was to be natural and should require a minimum of translation from human thought to the computer representation of that thought. The Smalltalk project exemplifies object oriented systems and provides a vocabulary for further discourse. Fortunately, members of this group have published widely [Gold83, Gold84, Kras83].

Smalltalk-80 is an integrated programming environment. In addition to a programming language and an interactive graphics system it provides functions normally associated with an operating system. These functions include memory management, a file system, processor scheduling, display handling, and compilation.

The entire Smalltalk system is modeled on a form of object-oriented programming. At the highest level the user's interaction with the system can be viewed as an interaction between two *super-objects*, the User and Smalltalk. Delving deeper into Smalltalk reveals a multitude of smaller object types that serve a variety of roles. All datum in Smalltalk are objects. Each object type can be further classified into more specialized subtypes. It is the communication and interaction between the various objects of the system that determine the functionality of Smalltalk.

The vocabulary of Smalltalk is quite small. The most important concepts are classes, objects, methods, and messages.

In the Smalltalk programming language, the primary unit of data organization is the *object*. Everything in the system is represented and manipulated as an object. An object is a data structure consisting of local private memory and a set of operations, called *methods*, to manipulate information stored in the private memory or perform actions based on that information. The only way to access the private memory of an object is through the methods defined for that object. An object is similar to the concept of an abstract data type in other programming languages. Other examples of objects include numbers, character strings, queues, rectangles, file directories, compilers, text editors, and programs.

A *class* is a description of a set of objects of the same type. The individual objects described by a class are its *instances*.

Smalltalk is a world of communicating objects. The methods of an object are its interface with the other objects in the system. The medium of communication between objects is the *message*. Whenever an object A (the *sender)* wants to get another object B (the *receiver)* to do something, A sends B a message which is interpreted by B's message interface. The message initiates execution of one of B's methods which calculates a response, and B then returns this response to the sender. This is the primary way of making things happen in Smalltalk.

Another important concept of object oriented programming in Smalltalk is the *subclass*. A subclass is a class of objects possessing all the variables and methods of some other class, called its *superclass*, except for certain explicitly stated additions that extend or override the variables and methods of the superclass.

A simple analogy to the subclass can be found in traditional dictionary definitions of English words. For example, a *man* can be defined as a male adult human. A *father can be defined as a man who has one or more children.* The class father is a subclass of the class man, and the class man is a superclass of the class father. Notice that father is defined in terms of man but is more specialized. In order for an object to be a father it must not only be a man but it must also have one or more children. All fathers are men but not all men are fathers. A subclass is thus a proper subset of its superclass.

Another idea important to subclasses is *inheritance*. A subclass is said to inherit all the attributes of its superclass. A father possesses all the attributes of a man, with some additional attributes not required of the man class. A Smalltalk subclass inherits all the variables and methods of its superclass but it also adds new variables and methods in its implementation description, or it may override a method of its superclass with a new definition. Inheritance is transitive. If class B is a superclass of class C and class A is a superclass of B then by the definition of inheritance class C inherits the attributes of class A is well as B.

Smalltalk is an interactive programming environment with a graphical interface. Designing a Smalltalk program requires an implementation of each of the program's objects including a visualization of that object. To support the desired graphical interaction, Smalltalk expects a high-resolution graphical display screen and pointing device such as a mouse.

2.5.3 GEMSTONE

The GEMSTONE [Maie86] database system is the result of a three year development project at Servio Logic Development Corp. of Beaverton, Oregon. The project's main goal was to merge object oriented language concepts with those concepts of a database system. To achieve this goal an object oriented database language, OPAL, is provided by the GEMSTONE system.

Further requirements for the GEMSTONE system were: support for an extensible data model that captures behavioral semantics; no placement of artificial bounds on the number and size of data objects; support for all of the database amenities (concurrency, transactions, recovery, associative access, authorization); support for an interactive development environment.

Smalltalk (Section 2.5.3), an object oriented language, provided the answer for some of these requirements, such as the provision of a powerful user interface along with many application tools for a development environment. However, it did not meet the requirements of a database system. The Servio group added the following enhancement to Smalltalk: support of a multi user disk based environment; support for the enforcement of integrity constraints; support for recovering the database to a consistent state after program, processor or media failures; and virtual memory implementation for large object spaces. These enhancements resulted in an architecture composed of two basic components: Stone and Gem. These two components correspond respectively to an object manager and a virtual machine. Stone provides the secondary storage management, concurrency control, authorization control, transaction management and support for associative access. Gem sits atop Stone and elaborates the Stone storage model into the full Gemstone model.

As with all computer based systems, the perennial performance efficiency problem arises in GEMSTONE. To address this issue, the Servio group is attempting to design more efficient algorithms for search and sorting of the data. Another approach is to move GEMSTONE into the realm of distributed systems. The future research areas include interfacing GEMSTONE to other languages such as LISP and PASCAL.

2.6 Summary

Sections 2.2 - 2.4 of this chapter have presented many approaches toward solving the design management problem. We classified these approaches based on the type of DBMS used within the CAD System. The three categories; commercial DBMS, special application DBMS, and integrated DBMS, are subclasses of a class we call the *environment*. Environment is a super class encompassing the overall data management facilities found in the CAD environment.

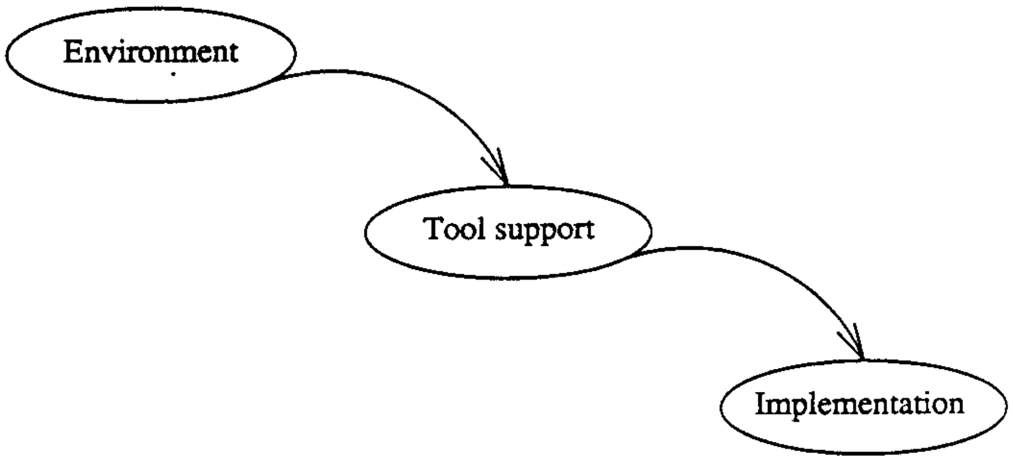


Figure 2.9 Views of CAD Systems

Figure 2.9 includes two other categories we formulated during this investigative phase. One such category, tool support, is the grouping which differentiates between that which is a design-specific system, one capable of handling tools created to manipulate a specific design, or a design-general system, one which is capable of incorporating various types of design tools. Subcategories of the tool support category are: those systems which support only one tool; those systems which support a well defined set of tools and is a complete, closed system; those systems which support multiple tools and is an evolving open system (extensible).

Table 2.1 is a detailed summarization of a particular approach's data base concerns at the environment level. This table is restricted to those environments specifically involved with CAD system support. Therefore, we have not included in this table are the programming support environments such as KAPSE and Smalltalk.

	Database Research Interest				
	MCS	IET	DIM	DTS	KBDP
TORNADO(81)					
FLOREAL(81)		X		X	
IBM(82)				X	
INGRES(82)					
IPAD(82)	X	X			
SARA(83)					
WiSS(83)		X	X	X	X
GRADAS(83)			X		
DIS(85)		X			X
MITEL(86)	X	X	X	X	

Database Research Areas: Column Key

- MCS: managerial control support
- IET: integration of existing tools
- DIM: design integrity maintenance
- DTS: design transaction support
- KBDP: knowledge base of design procedures

Table 2.1 Related Approaches: Environment Level

	Database Research Interest					
	DDS	ADD	UDS	TDM	APS	CL
TORNADO(81)	X	X	X	X		
FLOREAL(81)	X				X	
IBM(82)	X	X	X	X		X
INGRES(82)	X	X		X		
IPAD(82)	X	X		X		
WiSS(83)	X	X	X			
SARA(83)			X	X		X
GRADAS(83)	X	X			X	
DIS(85)	X					
MITEL(86)			X	X	X	X

Database Research Areas: Column Key

- DDS: design data structures
- ADD: low level access to design data
- UDS: unstructured data support
- TDM: use of traditional data models
- APS: appropriate language support
- CL: component library

Table 2.2 Related Approaches: Implementation Level

The third grouping, or category, is the Implementation level. As indicated at the beginning of this chapter, subclasses of this class include the use of different data structures, different ways of access the design data, appropriate use of traditional data models, appropriate languages to support the design process, and provision of libraries with appropriate access to be used throughout the design process. Table 2.2

indicates those approaches that were particularly concerned with this level.

Combining the two tables and looking at the chronological sequence, we can see that there has been an evolution towards concerns around developing an integrated management support system with increased functionality. This is evidenced by the shift from the implementation level to the environment level over the past ten years. It is also evident from the research efforts that there is an increase in concern over the usability of a system, that is, the extent to which designers will use the CAD system.

Although each approach has considerable merit and contributes to the realm of knowledge about CAD DBMS support, we have found that each has its limitations which preclude the implementation of a fully supportive system for the design process. These limitation are :

1. there is little regard for flexibility - the data management system is not concerned about new uses of the data and does not emphasize a flexible design which can accommodate inclusion of new tools, which may view the data is a slightly different way.

2. there is a clear separation of database interactions and design interactions - most systems require that the end-user access the database via a separate subsystem, the data management system.

3. there is no distinction between supporting system defined policies and implementing system policies - the data management concerns have not focused on

defining a nucleus on which to build design management facilities.

4. any work concerning design process and knowledge base support is a separate issue - with the exception of FLOREAL, the work surveyed is primarily concerned with support of the design process without regard to interfacing to knowledge bases which might generate new and enhanced uses of the system.

We propose an approach in the next chapter which draws on the results of the previous work and at the same time provides for a flexible, extendible support system which is tightly integrated into the CAD environment. This support system provides DBMS functions appropriate within such a CAD environment. We have incorporated many of the characteristics described in the programming environments section (2.5). In particular, the object oriented flavor of Smalltalk and GEMSTONE is a prominent feature of this research work.

We also realize that the toolset which will be brought into this system is not static and that the functions of new tools cannot be determined a priori. Therefore, we must provide as much knowledge about design objects as possible to allow for new and novel use of these objects. However, at the same time we are careful to provide an environment which can customize its toolset so that the present user population will readily accept such a system.

CHAPTER 3

Design Environments

3.1 Introduction

The term, Computer Aided Design Information Systems (CADIS), infers a symbiotic relationship between CAD Systems and Information Systems. Providing appropriate data management support requires that the enterprise, in this case the CAD System, be well defined. The purpose of this chapter is to provide a solid foundation toward the understanding of CAD Systems. Similarly, the capabilities and features of the Information System need to be well understood. These capabilities and features were presented in Chapter One.

The first sections of this chapter describe and classify present day CAD Systems. This classification scheme emphasizes the differences between CAD systems. The similarities found in the structure and behavior of CAD systems are described later in this chapter. These similarities spawn a definition of Integrated CAD Systems and an associated description of the development of such an Integrated CAD System. Preceding this description is a description of System's ARchitect Apprentice (SARA). This is both a design methodology and a set of tools to support the methodology. The Integrated CAD System described in this chapter is built using both the tools and the methodology of SARA.

3.2 Computer Aided Design Systems

A high level definition for Computer Aided Design (CAD) is the usage of computer hardware and software for the design of products that are needed by society. CAD is viewed to be the integration of computer science methods and engineering sciences within a computer-based system (CAD system), which provides a communication subsystem, a data base, and perhaps a program library. A CAD system ensures many basic functions such as the storage of information, computation facilities, communication management, and problem solving assistance.

3.2.1 Classifications

Presently CAD systems are used for many different engineering and scientific tasks. These tasks involve design of both physical and non-physical entities. An example of a physical entity might be an integrated circuit, whereas, a non-physical entity is exemplified by the design of a piece of software. Classification of present day CAD systems is based on the type of tasks performed. The following list briefly describes six classes of CAD systems.

1. Drafting board systems: systems which lend graphical support in order to generate drawings [Prei82].

2. Electronic design systems: systems which support the design of electronic components. These systems may support one or more of the following levels found in the electronic design process [Blai85, Hutc85].

    a. specification level
    b. functional level
    c. logic level

d. layout and placement level

e. physical circuit level

Tools in such systems provide for the generation, the simulation and the analysis of these descriptions. In fact systems may exist for different tasks in the design process.

3. Mechanical design systems: systems which provide capabilities to design mechanical parts [Fisc81].

4. Computer design systems: systems which provide for the generation, simulation, and analysis of computer system models. We will refer to these systems as CADOCS (Computer Aided Design of Computer Systems) Systems [Davi83].

5. Architectural design systems: systems which provide for the development of architectural plans, the checking of specifications and the checking of proper structures [Glas82].

6. Manufacturing systems: systems known as CAD/CAM systems which provide design generation capabilities for various products, numerical control operations, process plans and even inspection plans. These systems support the detailed tasks required to produce a physical design [Hosk81, Madd81].

7. Software design systems: systems which provide for software development. Support is given for version control, configuration management, code analyzers, test generators [Tich82a, Boeh84].

The classification of CAD systems presented in this section is based on the differences between the tasks performed. However, CAD systems are similar, particularly in two domains. Such similarity is found in the structure and behavior of these various CAD systems.

3.2.2 Structure of CAD

Two major environments define the structure of CAD systems, the hardware environment and the software environment. The hardware environment is characterized by: one (or more) input devices; graphic display devices; computer; mass storage; and output devices. The software environment is characterized by: a host language (usually FORTRAN); several software tools for access to non-standard features of the computer installation such as graphics devices; data bases etc.; one or more software tools which allow for the generation, analysis, and/or simulation of entities, and languages to drive the tools. This collection more than likely consists of a multitude of tools which have been synthesized from a collection of separately developed tools. The individual tools are written in various dialects of FORTRAN, PASCAL, C, LISP, and assembly code.

3.2.3 Dynamics of CAD

CAD operations can be performed on a batch basis or in an interactive mode. Today, nearly all CAD systems are interactive.

A batch CAD system requires that the end user submit data and/or programs which have subroutine calls to various CAD procedures. The system uses this information to produce a design in some form, or to perform an analysis on the data or simulate the model which was input. Long turnaround times characterize these systems. In fact, it is not uncommon to submit such batch jobs to be run at night or over the weekend. Results of these jobs are stored in files or returned to the user. If such results are to be used by another tool, then the end user might have to reformat the resultant data and submit this to be used by the other tool.

Interactive systems are usually a collection of tools which allow the user to more rapidly explore design alternatives. The user of such a system usually is faced with completely different user interfaces when accessing each individual tool. Some tools may provide help facilities, but the system itself generally has no integral help facilities. Additional tools used in this interactive environment are those which are needed to reformat the data of one tool so that another tool might use it.

3.2.4 Integrated CAD Systems

An integrated CAD system is viewed as a set of tools and a nucleus (kernel) whose facilities are shared by all the tools. This integrated system is to provide a user-oriented environment for formulating designs and also to support the development of tools to be added to this system. True integration in a CAD system is needed to reduce the potential complexity that may result from bringing a multitude of tools under an umbrella and to enhance the ease of use of such systems so that it will be accepted and used by all potential users.

A CAD system's integration is achieved throughout various areas of the system; data integration, user interface integration, tool integration, and task integration are the major areas of concern. Data integration requires that a consistent internal representation and a consistent storage representation be provided. User interface integration calls for a common interaction handler and also a common execution environment. These areas of integration might be regarded as design-space invariant. That is, they are areas marked for integration which are independent of the design space (class of CAD system) in which they are embedded.

The third and fourth areas of integration, tool and task integration, are design-space specific. Tools are brought into the system which support the entire life cycle of the design. This life cycle constitutes the integration of various tasks. The life cycle may include:

a. requirements
b. structure
c. behavior
d. analysis
e. simulation
f. verification
g. ancillary support functions

Extensibility is also required of an integrated CAD system. CAD systems are continually evolving as new modeling tools are created. Thus, an integrated system must be able to be modified or augmented as new needs or applications arise or as new tools are developed. This modification needs to be done easily with little impact to the system and its users.

We now describe a system which is integrated across the life cycle of the design. This system employs a unique methodology of design and supports this methodology through the various tools found in the system. The tools are used as examples throughout this dissertation while describing the work done.

3.3 SARA/IDEAS - A Computer-Based System Design Methodology

System ARchitect's Apprentice, SARA, is a requirement-driven top-down and bottom-up design method for concurrent digital systems [Camp78]. SARA supports the design process for complex concurrent digital systems. Both hardware and software design are supported. The SARA method was earlier supported by an extensive body of software designed at UCLA and implemented in the PL/1 language on the MULTICS system at MIT. It provided separate tools for the structural and the behavioral modeling of systems. The history of SARA is given in [Estr78].

The following sections first introduce the SARA design methodology and then describe, in separate sections, each major tool or subsystem that makes up the SARA tool system. A lock mechanism to support concurrency control in a multiuse database system is employed as a running example to clarify the design procedure and its supporting tools.

3.3.1 Design Procedure

This section describes the SARA design procedure as depicted in Figure 3.1. The design process is *initialized* by insisting that the designer partition the universe of design discourse into a *system module* and an *environment module* in which the system will operate. This first step may seem rather mechanical, but its omission is the cause of many faulty designs. The environment module is made explicit so that the designer is forced to focus attention on what assumptions are being made about stimulus and response conditions affecting desired system behavior. Those documented assumptions constrain the conditions under which designer-specified requirements can be expected to be met by the system module.

Neither the environment assumptions nor the system requirements are yet supported by a formal language and a corresponding language analyzer. Although Winchester [Winc81] has proposed a SARA requirements definition language and a requirements analysis technique, it has not been implemented as a SARA tool.

The next step in initialization is the development of a high-level behavioral description of the system module. Behavior is described in three different domains: the control flow, the data flow, and the interpretation domains. Each is supported by a language and language analyzer. Collectively, they are supported by a simulator.

Both *top-down partitioning* and *bottom-up composition* are supported. If the system being designed is simple it may be described immediately in the three languages already mentioned.

Designers are rarely faced with the task of starting from scratch. Complex systems are composed of many subsystems and it may be possible to re-use what another designer has already provided. A power supply is an obvious example of a re-usable subsystem.

If, for example, the system being defined is a variant on a well established product line, it may be possible to search an existing library of previously designed and tested modules. These *building blocks* can be collected to form a *composition.* If the product line is special-purpose digital controllers, the building block library might contain descriptions of TTL DIP chips or gate arrays that typically comprise major portions of the product line.

However, a system is likely to require the design of a some subsystem or component. If the new subsystem is large, divide-and-conquer is employed. The system module is *partitioned* into smaller, more manageable modules. Initialization is repeated for each new module thus identified. Each new module becomes a system that exists within a containing environment.

Regardless of the tactic taken, partitioning or composing, the resultant design is tested using the many tools in the SARA complement. These tools are generally one of two types, analyzers or simulators. The results of analysis or simulation are tested against the requirements. If requirements are met, then the designer may turn attention to another module. If requirements are not met, a new partition or composition is attempted in a search for satisfaction of requirements.

In the following sections, each major step in the methodology will be discussed in greater detail using a lock_mechanism example.

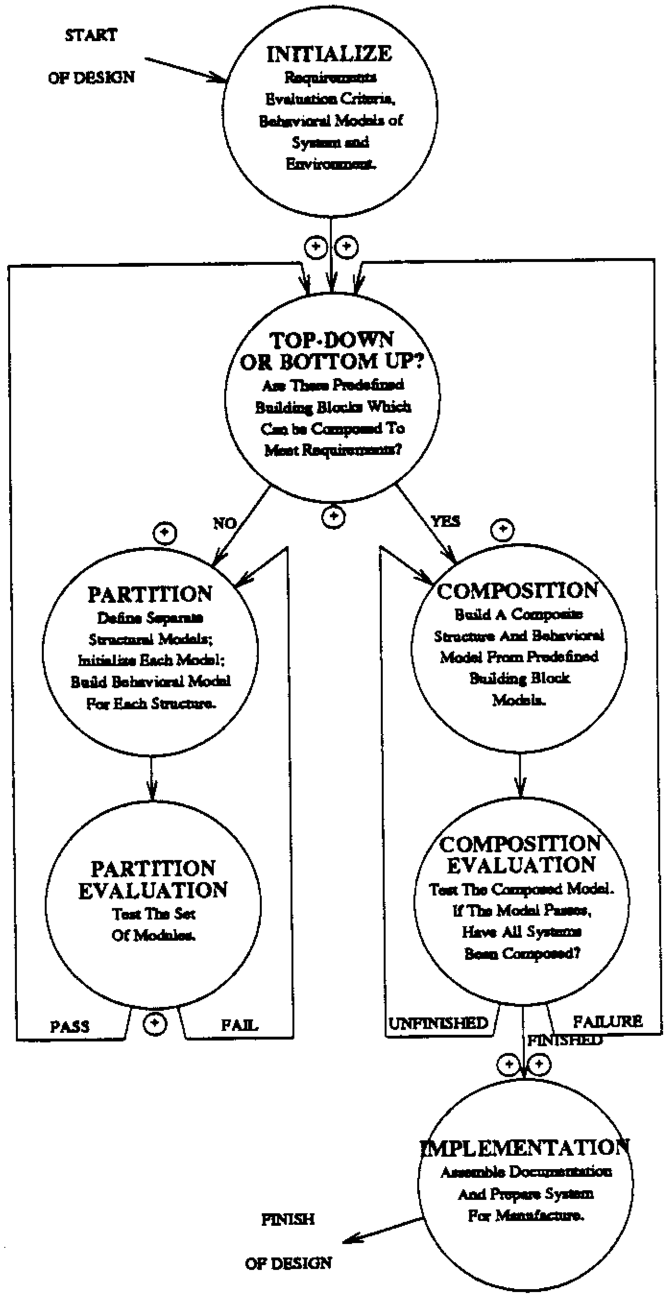


Figure 3.1 The SARA Methodology

3.3.2 Requirements Definition

The SARA methodology is requirement-driven, yet it has no supported requirements definition language nor language analyzer. Winchester [Winc81] has proposed such a language and has defined a set of analysis techniques, tools, and procedures that fill this requirements definition subsystem gap.

The Requirements Definition Language, RDL, is used to separately specify the functional, process, and attribute requirements that comprise the semantic model of the computer system being specified. The semantic model is composed of six primitives that describe the structural and behavioral components of the system. Winchester describes a correspondence between these six primitives and those of the extant SARA system. Given this correspondence, it is possible to generate SARA models from RDL and to apply SARA analytical and simulation tools in the verification of the specification.

In lieu of an RDL specification, the next two sections describe the environment assumptions and the first definition of the system module for the lock_mechanism example.

3.3.2.1 Environment Assumptions For A Transaction Lock-Mechanism

The high level assumptions for the environment are that the environment will define the type of lock required and will manage to issue one request at a time. No "write" will be permitted during a "read" operation.

Environment Model

A specific model of the environment contains three processes: **`requestor`**, **`releasor`**, and **`transaction`**(Figure 3.3). The **`requestor`** process behaves as follows:

- It receives requests from transaction processes for read/write access to a record.
- It determines the appropriate type of lock to request (exclusive, read-only, etc.).
- It sends a lock request to the lock_mechanism system through the **`request`** procedure.
- It sends only one request at a time.

The **`releasor`** process behaves as follows:

- It receives a record to release from the transaction process.
- It sends an update message to the lock_mechanism system **`release`** procedure.
- It does not send an update message for a record that has not been granted to the transaction.

The **transaction** process behaves as follows:

- It processes transactions as they are received.
- As a database record is needed it suspends processing until the record has been acquired.
- After update of a record it releases the record to the database.
- It signals termination to the environment upon completion of the process

3.3.2.2 System Requirements For A Transaction_Lock Mechanism

The high level system requirements are that the system will determine whether a request to access a data item can be granted. If access is granted, then a read operation on the database item is allowed as long as no write operation is simultaneously occuring. If access is not granted then the request is placed in a queue until such time as the database item is available to access.

System Model

The system will contain four processes: **manager**, **holder**, **databaseread**, and **databasewrite**. The **manager** process behaves as follows:

- It receives requests for locks one at a time.
- It determines if the request can be granted or not.
- It sends the request to the **databaseread** process, if the request can be granted.
- It sends the request to the **holder** process if the request cannot be granted.

The `holder` process behaves as follows:

- It maintains queues of transactions on data records.
- It receives messages from the `databasewrite` process when data records are released.
- It sends messages to the `manager` process when a particular previously queued transaction is able to request a lock.

The `databaseread` process behaves as follows:

- It receives messages to read a data record.
- It sends the data record to the appropriate transaction.
- It cannot access a data record if the record is simultaneously being updated.

The `databasewrite` process behaves as follows:

- It receives a data record to write to the database.
- It sends a message to the `holder` process that the record has been updated.
- It cannot update a data record if the record is simultaneously being accessed.

3.3.2.3 Evaluation Criteria For The Transaction_Lock Mechanism

A common form for expressing evaluation criteria is to define the initial control graph state and data state; conduct an analysis to show that no deadlock or undesirable cycles will appear; and design a simulation experiment to observe the system behavior.

3.3.3 Structure Model

The structure of a system is expressed in terms of the Structure Model, SM. The SM has three primitives: *modules, sockets, and interconnections*. Modules can be connected with other modules by an interconnection connecting two sockets, one socket in one module and one socket in the other module. Thus, sockets are communication ports for modules.

The interconnection is not directed; it models a communication line but does not reveal which way the information flows. An interconnection always connects two and only two sockets. Furthermore, a socket can have only two interconnections attached to it: one going out and one coming in. Hierarchical decomposition is achieved by refining a module into submodules.

There is a top level module called universe which has no sockets. Hierarchical decomposition is achieved by refining a module into submodules. This process can be repeated until the system has been decomposed into small enough modules, whose behavior can be directly mapped to an existing behavioral model stored in the Building Block Library or whose behavior is simple enough to be understood and expressed using the behavioral primitives.

In our lock_mechanism system, we would decompose our universe module into the lock_mechanism system and its environment. The environment and the lock_mechanism system would communicate through the **`request`**, **`release`**, and **`access`** operations. Figure 3.2 shows the SM for the lock_mechanism system.

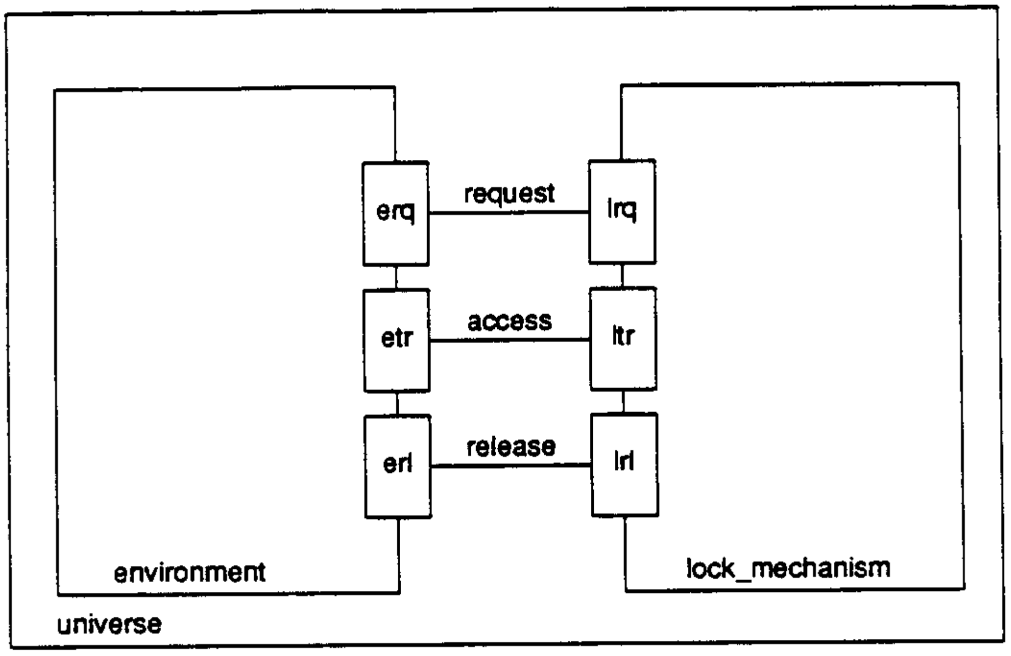


Figure 3.2 Structure Model of the Lock_Mechanism System

The environment module has three sockets: erq (for environment request), etr (for environment transaction), and erl (for environment release). These sockets are connected through interconnections request, access, and release, respectively, to sockets lrq ,ltr, and lrl in the lock_mechanism module.

The environment module or the lock_mechanism module could be partitioned further into submodules if needed.

3.3.4 Behavioral Modeling

In SARA, the behavior of the system is modeled using the Graph Model of Behavior, GMB [Razo77]. The GMB offers the designer three different but related modeling domains: control, data, and interpretation. The designer focuses on one of these domains at a time. After developing independent systems descriptions in each domain, the designer insures that they are consistent with each other.

3.3.4.1 The Control Domain

The control flow model describes concurrency, synchronization and precedence relations in a graph using an underlying theoretical model equivalent to Petri-Nets [Pete81].

The control domain of the GMB is a directed hypergraph, i.e., a graph in which the edges may have multiple sources and/or multiple destinations. *Control nodes* (the vertices) represent events and *control arcs* represent precedence constraints, or a partial ordering, among the events.

Each node has an *input logic expression,* which is a boolean expression on the input arcs, that expresses the condition under which that node can be initiated. An OR, ''+'', in the input logic means any of the operand arcs can initiate the node. An AND, ''*'', in the input logic means that all operand arcs must pass control before that node can be initiated.

Each node has an *output logic expression,* a boolean expression on the output arcs, which shows where control is passed upon termination of that node. An OR here implies control is passed to one of the designated arcs. An AND implies control is passed to all of the designated arcs.

Both input and output logic expressions can be arbitrary functions using ANDs and ORs. Control flow in the control graph is represented by the passing of *tokens* through control arcs. When a node is initiated, it consumes the tokens which enabled it. Upon node termination, tokens are created and placed on output arcs according to the node's output logic expression. The semantics of a control graph are dictated by an underlying machine, known as the *token machine*, which performs state-to-state transformations on the graph, starting from an *initial token distribution* and terminating if and when no further transformations are possible.

Continuing with our lock_mechanism system, we define a control graph for each of the modules defined in the SM. To show this mapping, each control graph can be drawn on its corresponding SM module:

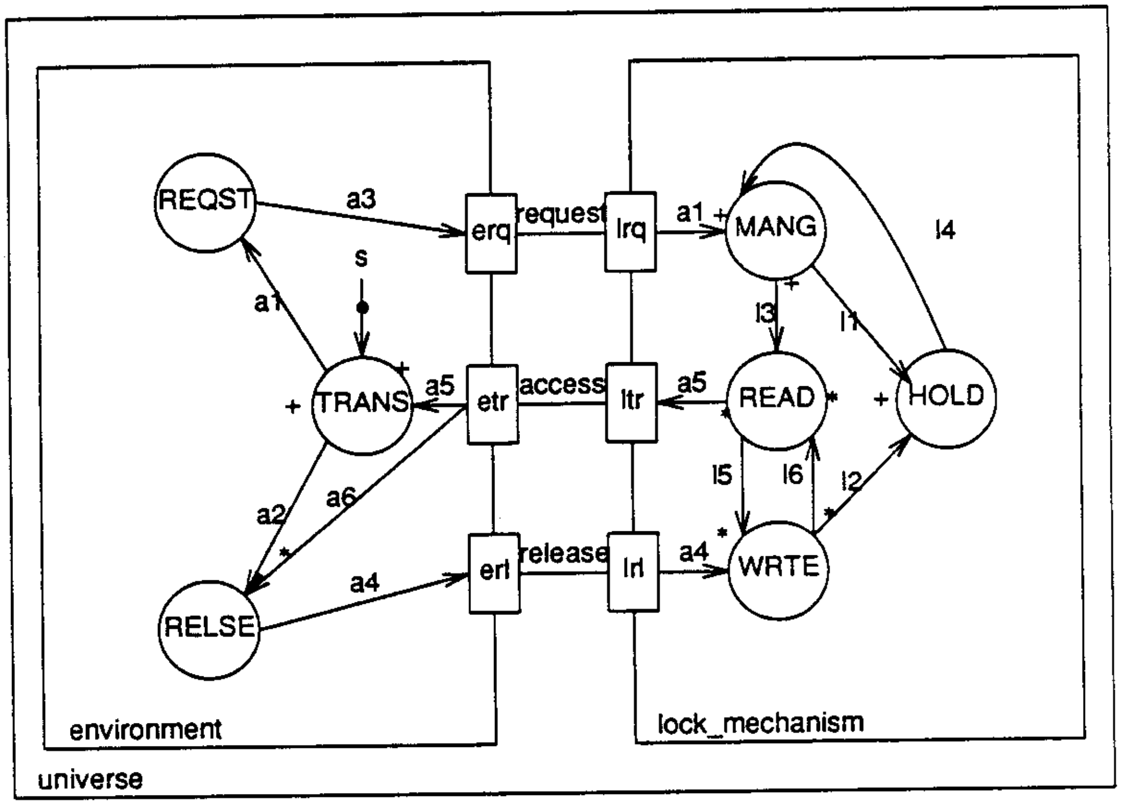


Figure 3.3 GMB Control Graph for the Lock_Mechanism System

The following tables describe the function of the various components in the control graph:

Control Nodes

TRANS	Transaction process, initiates requestor or releasor.
REQST	Requestor process

RELSE	Releasor process, sends update message to the lock_mechanism
MANG	Manager process, receives lock request and sends appropriate message
HOLD	Holder process, manages queues of waiting requests
READ	Receives message from the manager, accesses the requested data record
WRTE	Receives update request, performs the update operation, and sends lock release to the `holder` process

Control Arcs

s	Arc to initiate the system.

Tokens

o	Initial state of tokens.

3.3.4.2 The Data Domain

The data domain of the GMB is a bipartite directed graph, i.e., a graph in which there are two kinds of nodes (*datasets*, represented as rectangles and *data processors*, represented as hexagons) and in which arcs (called *data arcs*) are used to connect datasets with data processors. Thus, every data arc goes from a data processor to a dataset or viceversa. This graph represents the data flow of the system by defining its data paths. Datasets model static collections of data. Data processors are data transformers which can read from and/or write to datasets. Data arcs define the read and write accesses of a data processor to a dataset.

Continuing with the buffer example, we draw the data graph over the SM and show the mapping existing between the data graph and the control graph.

The following table describes the function of the various data processors and datasets in the data graph:

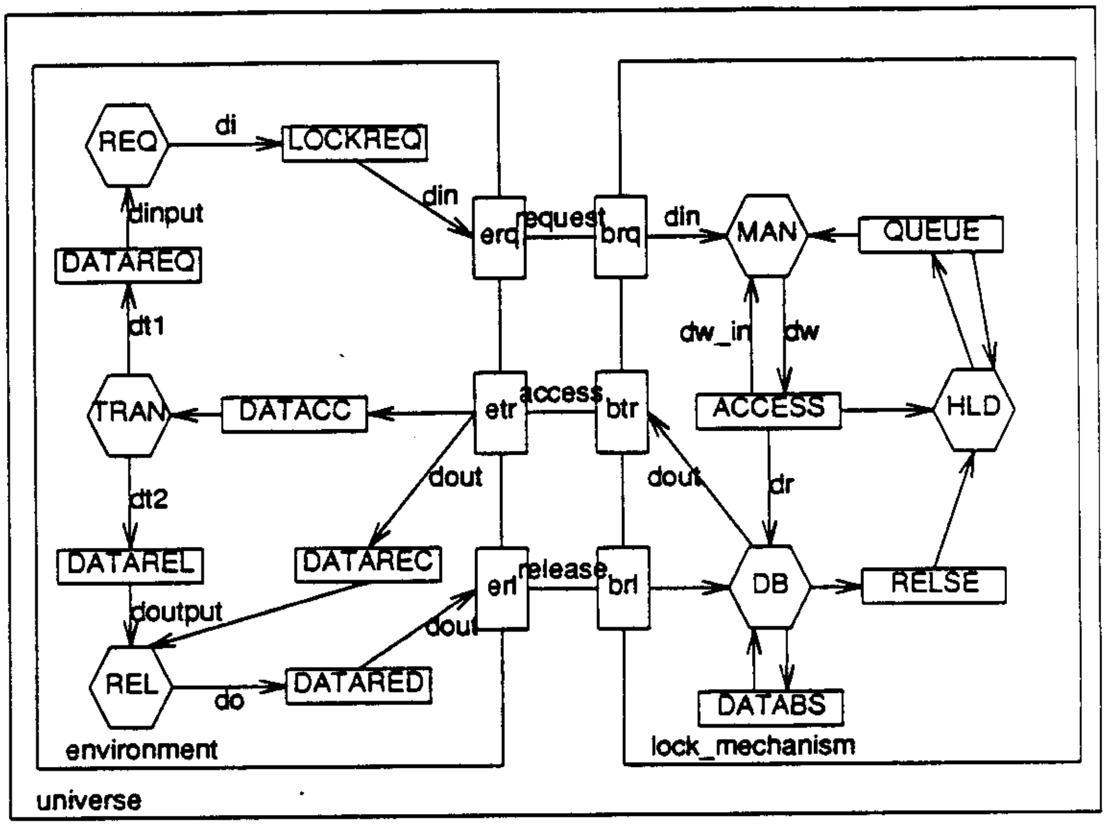


Figure 3.4: GMB Data Graph for the Lock_Mechanism System

Controlled Processors

REQ	Mapped to control node REQST, processor which receives request for a data record, determines the appropriate lock type and gives the request to the lock_mechanism
TRAN	Mapped to TRANS in the control graph. It executes a transactions, as data records are needed it requests such a record. Execution is suspended while waiting from the record. After use of the record, it releases the record by writing it to the DATAREL file.
REL	Mapped to RELSE in the control graph. It reads released records from DATAREL, and compares them against DATAREC records. If the record has been received by the transaction requesting release then a message is sent to update the database.

MAN	Mapped to MANG in the control graph. It receives requests from the environment or the HLD processor and determines if the request can be granted or must be held. It places the request in the dataset ACCESS.
DB	Mapped to READ and WRTE in the control graph. It reads messages from either DATAREC or ACCESS and writes to DATABS. It may also write to dataset RELSE.
HLD	Mapped to HOLD in the control graph. It reads data requests from ACCESS and places the request in the appropriate queue. It reads data releases from the dataset RELSE and removes a waiting transaction id from the queue and writes to the dataset QUEUE

Datasets

DATABS	Initial database of records
ACCESS	Individual records identifiers paired with transaction identifiers to be read by the DB processor or the HLD processor
RELSE	Identifiers of records to be released
QUEUE	Dataset with identifier of transaction waiting to be granted lock
LOCKREQ	Dataset with identifier of datarecord, transaction and lock type
DATAREQ	Dataset with with database access request
DATACC	Dataset with transaction identifier, identifier of datarecord and datarecord contents
DATAREL	Dataset with identifier of datarecord, identifier of transaction and new datarecord contents
DATAREC	Dataset with identifier of datarecord and identifier of transaction
DATARED	Dataset with identifier of datarecord and new datarecord contents

3.3.4.3 The Interpretation Domain

The Interpretation Domain defines the format of the data stored in datasets and defines the transformations of data performed by the data processors. Many interpretation languages can be used for this domain. The original SARA system used PLIP (an extension of PL/1) as its interpretation language. The current system, being implemented in a Lisp dialect, supports a Lisp-like interpretation language, "T" and is able to call upon any Unix function.

3.3.5 Building Block Library

In order to support bottom-up design, it is necessary to have a collection of previously designed and tested models, appropriate for the design domain, stored in a design database. The SARA design database is called the Building Block Library by Drobman [Drob80]. His work concentrates on hardware building blocks but the procedure is also applicable to software modules.

The primary hypothesis of Drobman's work is that

> "a set of models of hardware and software building blocks can be created and utilized as primitive elements in a computer-aided design system and methodology such that the composition of requirements-satisfying, partially correct, microprocessor-based digital systems is dramatically enhanced."

He demonstrated satisfaction of the hypothesis by defining building block descriptions of the Am2901 bit-sliced microprocessor, the Am29775 PROM, and other similarly complex devices, and then using those building blocks to design a 16-bit microprogrammable microprocessor.

Drobman's building blocks are prefabricated simulation models of physical building blocks. The simulation models are defined in the previously mentioned SARA languages, the Structure Model Language and the Graph Model of Behavior Languages.

Other SARA researchers have studied the requirements and organization of a design library [Land83, Mars83].

3.3.6 Socket Attribute Modeling

During research on the Building Block Library and the SARA simulation tools, it was felt that many of the errors detected during simulation could have been found much earlier by analysis of some as-of-yet undefined static description of the building blocks. This observation spawned Sampaio's research into the Socket Attribute Model, SAM [Samp79], and Penedo's research into the Module Interconnect Description, MID [Pene81]. While both deal with a description of a building block at its interfaces, sockets or interconnects, SAM concentrates on hardware building blocks and MID concentrates on software building blocks.

Sampaio provides a language to describe the behavioral attributes of a hardware module's sockets, for example, electrical characteristics (fan-in, fan-out), timing (set-up and hold times), bandwidth, and perhaps physical characteristics. With these descriptions attached to a module's sockets it is possible to detect inconsistencies occurring during composition of two or more modules. The detection of socket mismatch errors occurs at the time the socket connection is attempted, not later during an expensive and time consuming simulation that may not detect the error at all.

3.3.7 Module Interconnect Description

Penedo attacks the same problem as Sampaio, but on the software front. She describes software modules as they appear at their interfaces. Most type checking compilers detect some of the errors that Penedo is after, for example, procedures called with the wrong number or type of arguments. The product of her research is the Module Interconnect Description, MID [Pene81, Pene79]. Berry later shows that Ada package specifications meet the needs of Penedo's MID [Berr84]. Krell [Krel86] continues this line of reasoning by researching the suitability of Ada as the language for the interpretation domain of the GMB.

3.3.8 Extensibility and User Interface

The extant SARA system implementation at MIT was not constructed in an ad hoc manner. From the beginning, the implementors knew that no matter how complete their tool kit was, there would be inevitable pressure to add new tools. They therefore established a procedure for constructing a new tool and for eventually integrating it with the existing tool kit. This procedure is described in [Vern78]. To insure consistency between existing and newly defined tools, Fenchel [Fenc80] defined a user interface construction tool that promotes sharing of grammatical constructs between tools. By following the procedure and by using the user interface construction tool, the end product is *self-describing,* offering syntactic and semantic help to the end user. Fenchel's tool [Fenc81, Fenc78] is summarized in the following paragraphs.

Each tool initially is partitioned into user interface dependent and independent parts. The user interface independent part is partitioned into a collection of PL/1 routines that comprise the tool's functionality. The syntax of the user interface dependent part is described in an SLR(1) grammar. Upon recognition of certain syntax rules, a user interface independent routine is called.

Once the tool is fully constructed the user interface independent routines are merged with those of any pre-existing tools. The tool's syntax specification is added as a subdialogue to the tool system's grammar.

The underlying support tools use the grammar to provide integral help to the end user. This insures that the user gets help information that is in agreement with the implementation. It also alleviates the burden of providing help from the tool implementor.

3.3.9 The SARA/IDEAS Environment

The SARA methodology is supported by automated tools in an integrated, interactive environment. These tools allow the user to create and modify SM and GMB models. There is also a design data base from which models can be stored and retrieved.

The GMB Simulator [Razo79] is one of the tools in the SARA environment. Another tool is the Control Flow Analyzer, used to analyze control graphs for control flow anomalies such as deadlocks and to assure some control flow properties such as proper termination.

The SARA methodology and tools described in the preceding sections provide us with a framework to develop an integrated design environment. This research included the redesign of some of the basic tools described in these sections. This development provided us with much insight into the problems and solutions one might encounter while providing an extensible, integrated system. The resulting interactive modeling environment is the SARA/IDEAS system. The following sections describe the building of this interactive, integrated system.

Throughout the remainder of this dissertation the SARA tools, in particular the Structure Model and the Graph Model of Behavior will be used to provide examples when presenting the various components of the design environment developed during this research.

3.4 Intelligent Design Environment for Analyzable Systems (IDEAS)

IDEAS is an integrated interactive environment currently being developed at UCLA. It is the second generation of the SARA system and thus is refered to as the SARA/IDEAS system. The SARA/IDEAS system is intended to run on Apollo, Sun, Vax, HP, IBM, and MacIntosh workstations and consists of various tools needed by the class of CADOCS systems. This author and Duane Worley collaborated in the development of a tool building methodology and run time system described in this section. Worley's focus was on the user interaction involved in building tools and using them, whereas the focus of this author is on the database interaction with the system.

The following list of high level goals represent a portion of those imposed on the development of the SARA/IDEAS (Intelligent Design Environment for Analyzable Systems) system so as to provide for a readily extensible and tightly integrated CADOCS system which can more effectively support complex design efforts [Worl86].

1. Support strong partitioning of user interface issues from tool modeling issues

2. Automatically provide a consistent, friendly user interface directly from the tool's specification

3. Provide a large body of reusable software that is common to CADOCS tools, including interaction handler, data base definition and manipulation, work space, and library of logical and physical device drivers

4. Automatically generate as much code as practical

These goals follow from observations that tool developers are primarily interested in the modeling capability and have insufficient appreciation for human interface issues, database management and amplification achieved through the integration of tools [Worl86]. An underlying support kernel, which provides user interaction handlers, graphics and data base management functionality, provides the means to meet these goals.

The provision of such an interactive integrated CADOCS system is embodied in a multi-phased method and specification system. The first phase, conceptual by nature, focuses on the semantics of a particular tool followed by a description of the syntax of a command language. Subsequent phases in the development of a tool consider the logical devices, and eventually the physical devices, that support the command language support. This approach provides a clean partitioning between a tool's semantics, syntax, and the underlying interaction hardware.

Worley's methodology is multileveled and object oriented. That is, a new tool is brought into this environment first by considering its semantic definition. The tool designer creates a new modeling capability by defining objects, their interrelationships and operations performed upon them. This designer must first identify the objects that the end user of this tool will manipulate or create. Next, the relationships between these objects must be defined as well as the relationships between objects belonging to the environment and/or to other tools. Finally the operations upon the object must be determined.

It is the function of the kernel to provide high level support for these defined objects as well as to provide environment objects. Environment objects are those objects which are present across the broad spectrum of CADOCS tools. A display screen is such an environment object. Another most important environment object is the CADOCS data base.

The Entity-Relationship Model (ERM) [Chen77] provides a natural mechanism for communicating many of these concepts to both human implementors and system facilities. The objects and their inter-relationships, as defined by the tool developer, are modeled respectively as ERM entity sets and relationship sets. This model forms the basis of the tool designer's view of the underlying database structures.

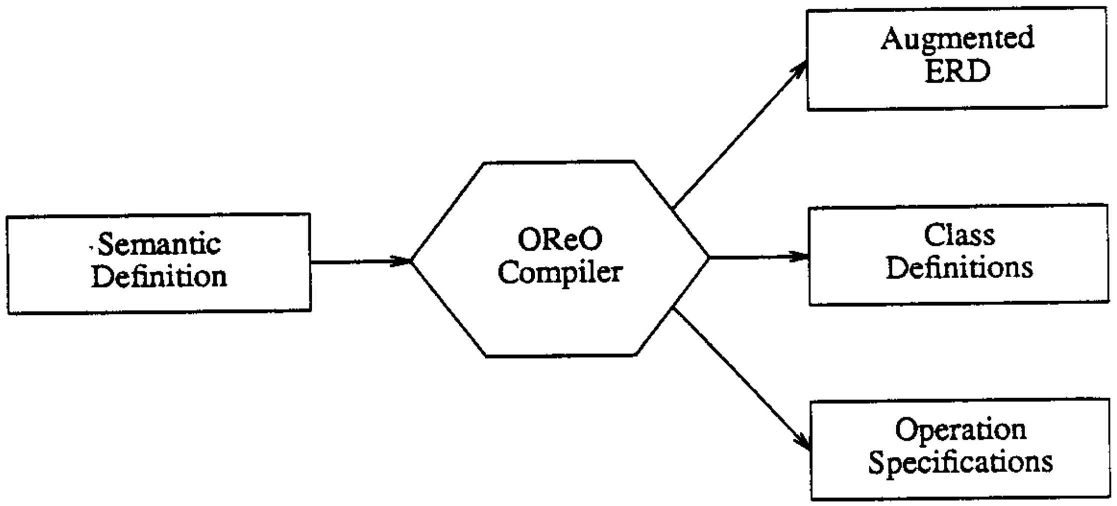


Figure 3.5 Semantic Compiler

Figure 3.5 shows that upon completion of the tool development phase, resulting in a Semantic Definition, a semantic compiler (OReO Compiler) produces a set of semantic Operation Specifications, a set of Class Definitions, and an ERD, augmented to satisfy the needs of a CAD Information System (see Chapter Five). The set of semantic Operation Specifications contain a specification for every operation identified in the operation step. The set of Class Definitions incorporate the information extracted from the object and relation specification. They include the necessary data structures to represent an object, its attributes, and its relations with other objects. The ERD is a graph which is derived from the object, attribute and relation descriptions. This graph (Augmented ERD), along with augments necessary to represent the design environment, is then processed by a data base compiler to produce the schema for this particular tool.

A portion of the output from the OReO Compiler, namely Class Definitions and Operation Specification, is used as a template by the Tool Implementor, who happens to be human, in order to flesh out the details of the code needed to implement the operations of the tool (Operation Implementations).

The remaining output from the OReO Compiler, the Augmented ERD, is used as input to a Data Base Compiler. The Data Base Compiler (Chapter Six) produces a data base schema which will provide for the storage and access of the objects and relationships defined in the concepual definition phase.

The syntactic description phase provides for specification of the human-machine interaction in terms of a grammar. The tool developer associates each sentence with a semantic operation defined in the previous phase, resulting in a grammar definition. Upon completion of the syntax definition phase, Figure 3.6 shows the Operation Specification from the semantic phase and the syntax definition (Grammar Definition) being input to a syntax (Grammar Compiler) compiler to produce an augmented transition network (ATN Graph) [Wood70] and a Token List. The ATN Graph is a graphical representation of the language, used to guide the execution of the command-response language interpreter. The Token List is a collection and categorization of the terminal symbols which appear in the syntax and will be resolved in a later phase.

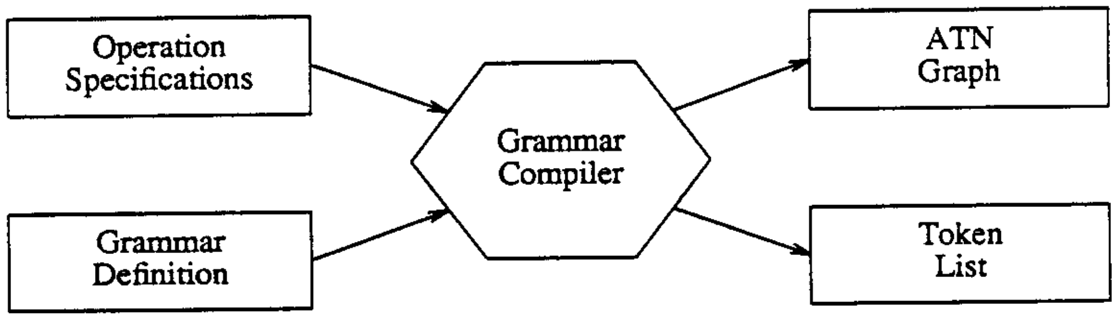


Figure 3.6 Syntax Compiler

The final two phases concentrate on supporting the low-level aspects of a tool's user interface. These two phases deal exclusively with interaction techniques and devices. The logical device description phase, resulting in a lexical definition, allows the developer to bind tokens from the previous phase to logical devices. These tokens consist of input tokens associated with interaction tasks and output tokens associated with shapes. A logical device definition buffers the developer from the specifics of physical devices by expressing the more general interface characteristics of logical devices. The physical device description phase focuses attention on the available physical devices to support interaction. This definition bridges the gap between the tool concept and the physical devices with which the user comes in direct contact.

Figure 3.7 shows the output from these two phases, Lexical Definition and Physical Definition, as being input to the Device Compiler. Other input to the Device Compiler consists of: the Token List; the set of Interaction Tasks; the set of Device Drivers; and the set of Shapes. The Token List is the same as described as output from the Grammar Compiler in Figure 3.6. The Device Drivers are software routines that implement the logical devices on the physical devices. The Shapes are a body of routines which draw primitive shapes. The Interaction Tasks are a set of software routines which implement the logical interactions performed by a specific physical device, such as locating, picking and selecting. Rather than create new software the Device Compiler calls on the System Librarian (Figure 3.9) for predefined software packages of device dirvers, interaction tasks, and shape routines. Output from the Device Compiler is an I/O Handler responsible for the execution of the tool.

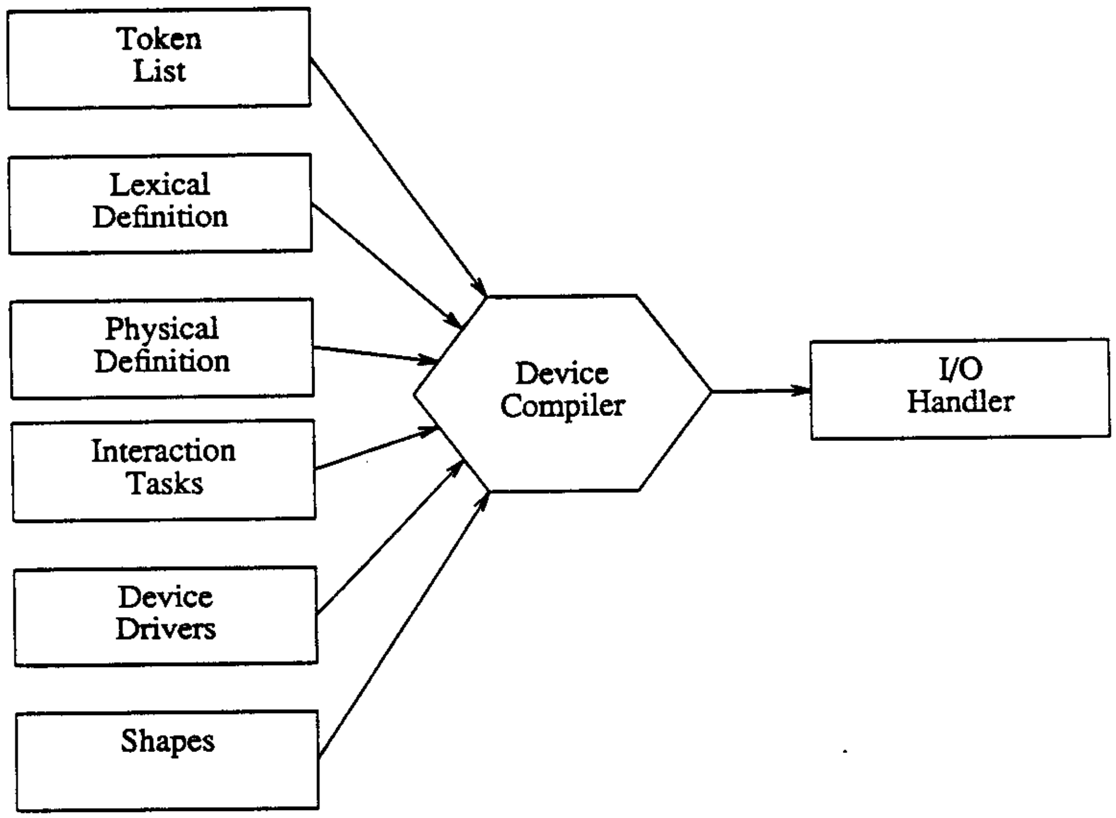


Figure 3.7 Device Compiler

Integration of a new tool into SARA/IDEAS is accomplished in two steps. Figure 3.8 details the complete set of processes involved in step one; the implementation of a tool invariant interaction handler and data base management system tailored by the tool's specification subsystem. The processes, hexagonal figures in Figure 3.8, provided by the system to perform this step are the ORe0 Compiler (Figure 3.5), the Grammar Compiler (Figure 3.6), the Tool Implementor (not separately illustrated), the Data Base Compiler (Figure 4.1), and the Device Compiler (Figure 3.7).

These processors, with the exception of the Data Base Compiler, have been described in the preceding paragraphs. Step two, resulting in an executable system tailored to a particular hardware environment, uses as inputs to the Integration Facility (processor), the outputs from the processors used during step one. This Integration Facility contains subprocessors, one of which is a Data Base Integrator (Chapter Four and Seven). This Data Base Integrator is responsible for the inclusion of all data base relevant information within the design database.

All processors found in Figure 3.8, collectively called the SYSGEN environment, have access to a stand alone processor, the System Librarian. The Librarian accepts requests (Library Requests) to search various libraries which are input to it and output source or object level descriptions in response to requests. As indicated in Figure 3.9, the input to the System Librarian consists of libraries of Logical Devices, Device Independent Shapes, Device Dependent Routines, description of Data Base Kernels, and description of various Interaction Kernels. We have previously described how the Device Compiler calls on the System Librarian in order to access predefined software routines. The Integration Facility might also call on the System Librarian to provide a Data Base Kernel and Interaction Kernel to be used as the support level for the executable CADOCS system.

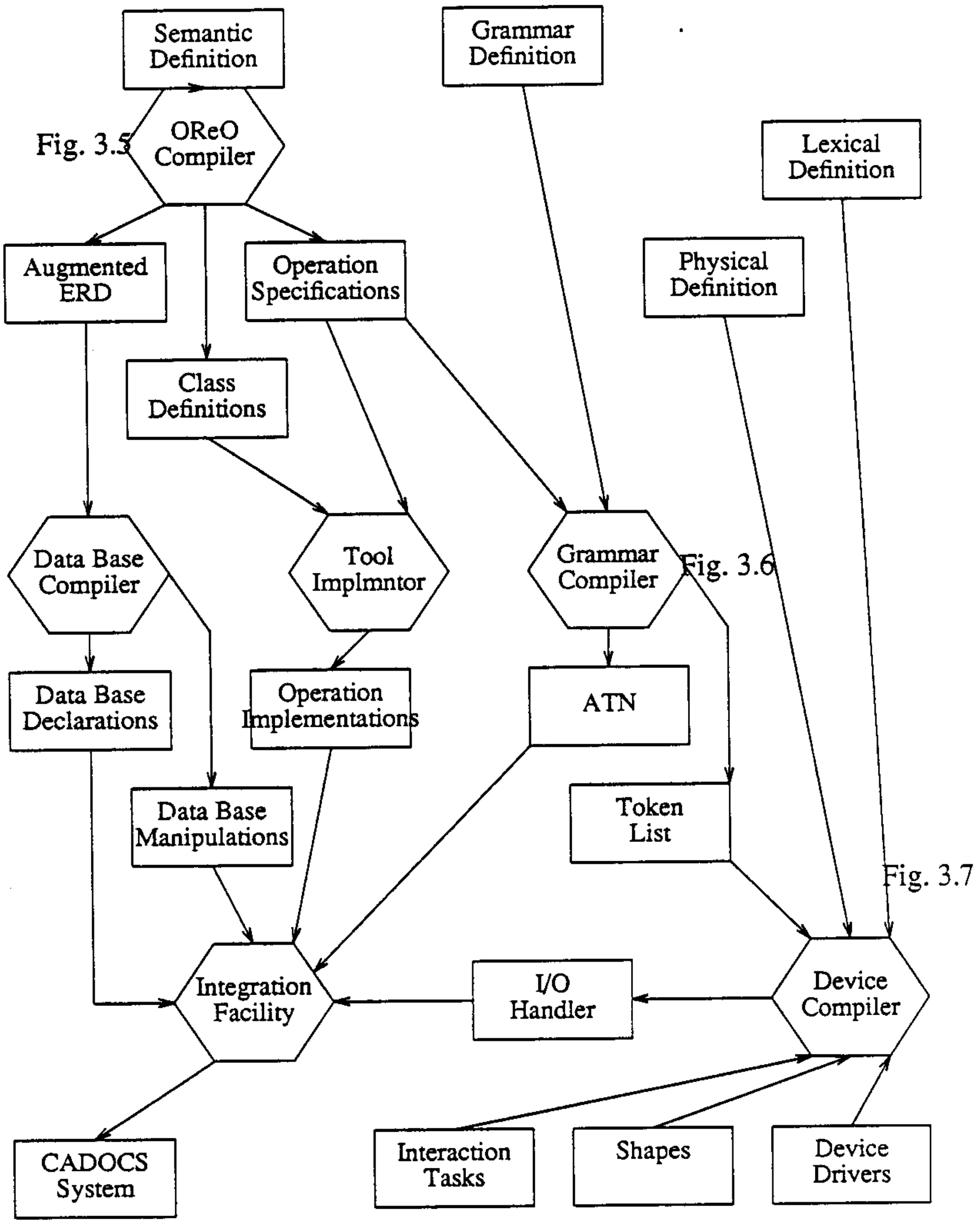


Figure 3.8 System Generation Data Flow Diagram

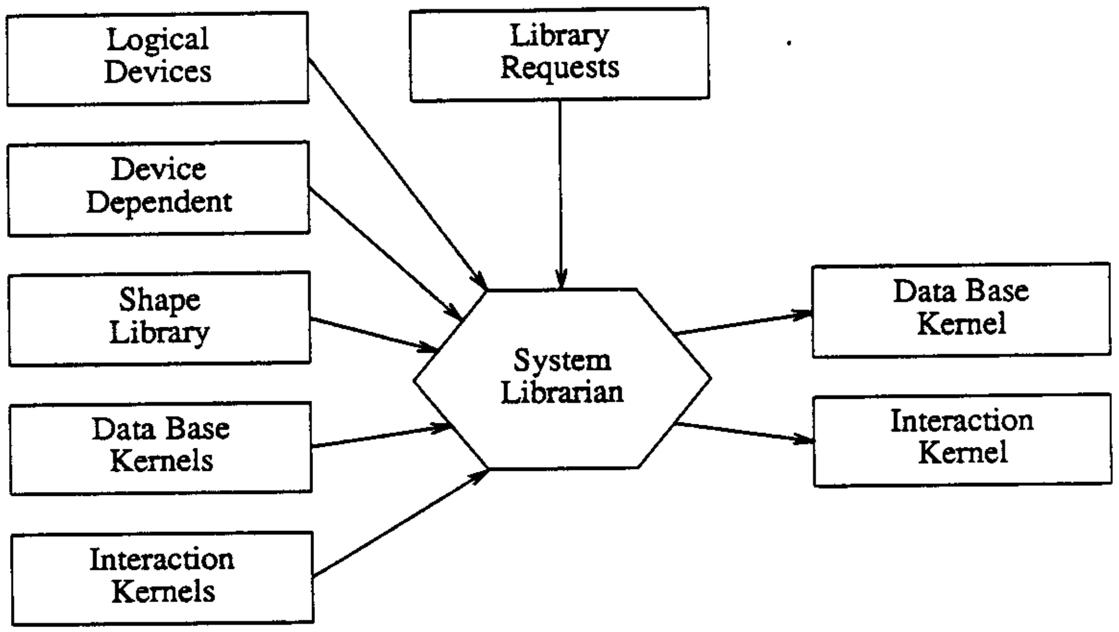


Figure 3.9 Library Facility

The methodical use of the tool building system, defined by Worley, produces a procedural interface to the modeling objects appearing in the user interface. Also defined by Worley is a kernel supporting the user interface management, the user interface management system. This kernel provides another, though more primitive procedural interface to objects. The end user of a tool is never aware of this level. A portion of this kernel includes the operations to support a design database system. This portion of the kernel, the DB_KERNEL, will be described in detail in Chapter Five.

3.5 Summary

This chapter first developed the concept of an integrated design system. A description of such an integrated CAD system, the SARA/IDEAS system, was then presented. This description included a definition of a tool building methodology (TBS) and the resultant run time system (SARA/IDEAS). We use these definitions and guidelines to formulate an approach to the intelligent data management within integrated CAD systems. An overview of this approach is presented in the following chapter. The realization of such a data management facility takes place within the integrated environment, the SARA/IDEAS system and is described in detail in the following chapters.

CHAPTER 4

Computer Aided Design Information Systems

Computer Aided Design Information System (CADIS) is a comprehensive integral component within the SARA/IDEAS system. CADIS is designed to support the tool building methodology as well as the run time system resulting from the integration of these tools.

The chapter begins by first defining CADIS. This definition incorporates the four major requirements of our design information system. The descriptions of the major components of CADIS follow this definintion. The concluding sections present the overall architecture of the database system. This architecture, in the ANSI/SPARC tradition, is multi-leveled. A brief descriptive overview of each level is then given. Detailed expositions of each level are presented in Chapter Five through Chapter Eight.

4.1 Definition

A Computer Aided Design Information System (CADIS) is defined to be a generalized support system which provides standard data management features found in commercial DBMSs, and yet is capable of supporting the design process found within a specific design environment. The design of CADIS is such that integration across the design tools (Tool Integration - see Section 3.2.4) is supported as well as integration across the various design phases (Task Integration - Section 3.2.4). The concept of an extensible CAD system is also supported by CADIS. CADIS is capable of absorbing new tools whose degree of integration into the system is defined by the tool designer. Thus, the four primary objectives of CADIS are to support and facilitate data integration, tool integration, task integration and extensibility. A secondary, though important, objective is the economical construction of such a system.

Provision of standard GDBMS features, found in commercial environments, can be accomplished by extending existing GDBMSs. This approach is attractive from a practical viewpoint, since it preserves the considerable know-how acquired in the building of these systems. Also, the vast amount of software which is required for general purpose operations need not be reinvented. The design of CADIS is based on the extension of a standard GDBMS (Chapter Six).

Supporting the design process involves not only the support of designs during invocation of tools, but also support for the building of the tools. Design is an evolutionary process, and likewise the design of a CAD system is an evolutionary process. Various versions of tools and designs exist during the lifetime of a CAD system. Software developers make use of some type of version control mechanism and configuration control manager. Many such systems exist and can be extended for use in a design system. The development of CADIS includes the use of a standard version control program (Chapter Seven).

Integration of tools and tasks within a CAD system necessitate an underlying common view of data. Although some tools may view data as text, such as an editor, other tools view data as code or even as single valued attributes. These different views constitute what is *variable* between CAD tools. This variability is described as tool dependent. However, the need to interact with long term storage (a database) is *fixed* regardless of the CAD tool. These independent portions of a tool require a policy independent mechanism to support them. These portions are collectively referred to as the DB_KERNEL (Chapter Five). The DB_KERNEL is provided to the Integration Facility processor (Figure 3.8) so that the tool dependent portions of the CAD tool (specified by the tool implementor) might be integrated with the tool independent portions to produce an executable system.

An extensible, flexible CAD system requires support from the data management system for dynamically modifiable storage structures. Such modification involves the ability to define new structures and the ability to extend existing structures. CADIS is an integral part of the tool building system (TBS) as defined in [Worl86]. The unifying concept in CADIS is that of an object oriented world. CADIS itself is viewed as an object as are the constituent tools of the CAD system. Mechanisms for the definition and manipulation of these objects are provided by CADIS. As new tools are added to the CAD system, CADIS also evolves. This object oriented view, coupled with the Worley TBS, allows new tools to be easily integrated into the existing system.

4.2 Design and Development

The design and development of CADIS is sectioned into two major tasks: the design of the database, and the design of the management system which manipulates the database.

The database design, the schema, consists of two levels: the top level, which is invariant, and the underlying level, which is evolving as new tools are introduced. The unifying structure containing these two levels is an object. That is, the database is considered to be an object with operations defined on that object, allowing one to access its internal objects. Two internal objects found in the database are the system object (invariant level), and the tool object (variant level). Operations are associated with each of these two sub-objects. Descriptions of the database design employ definition of the operations and sketches of the structure of the object.

The design of the management system involves design of the various components needed to support the tool building system, which we have previously defined, and the run time system. The design process employs many techniques. This dissertation describes these components using such techniques as HIPO lists, data flow graphs, pseudo code, and ADA packages.

HIPO (Hierarchical Input Processing Output) lists [Shoo83] require that the designer list all inputs to a process, list the processing that takes place and list all expected outputs of the process. The processes are defined in a hierarchical order starting with the top level of the system being designed. We use this technique to detail the Data Base Compiler and the Data Base Integrator in Section 4.3.

Data flow graphs, defined in detail in Chapter Three, hide the processing that takes place. They are a graphical means of representing the inputs and outputs of various processes.

Pseudo code [Rood85] is a structured paragraph form for describing logic associated with the implementation of procedures and/or functions. Pseudo code allows a program designer to quickly sketch the broad outline of the design and complete the details later. It is a popular method for detailing the algorithm without regard to some particular syntax. The processing which is performed within the components is defined in detail using pseudo code (Chapter Six).

ADA packages [Berr84] are used for design to perform data abstraction and detail design before beginning implementation. An ADA compiler allows these packages to be compiled so that interface inconsistencies can be detected. ADA packages are used to specifiy the operations performed on objects and relationships for the Browser tool (Chapter Eight).

4.3 Components

The major components comprising CADIS are: the underlying kernel of the data management system; the supporting mechanisms to create a database; the tools of the CAD system with additional mechanisms needed to support the evolution of designs.

CADIS Kernel

The underlying kernel of CADIS is that software neccessary for the creation and manipulation of the unified data model. It consists of an interface between the users' view and the logical view of the database, as well as another backend interface between the logical level and the implementation level. This kernel represents the invariant aspect of CADIS. Chapter Five of this dissertation defines the kernel, called the DB_KERNEL.

Support Components

The database definition facility is comprised of two major subsections. A Data Base Compiler and a Data Base Integrator. The contents of this component vary only as the underlying physical composition of the CAD system changes. Chapters Six and Seven develop these facilities in greater detail.

Figure 4.1 is a data flow representation of the data base compiler. This processor is taken from Figure 3.8, the SARA/IDEAS System Generation illustration. The HIPO list following Figure 4.1 provides more detail as to the inputs, outputs, and processing functions of this component.

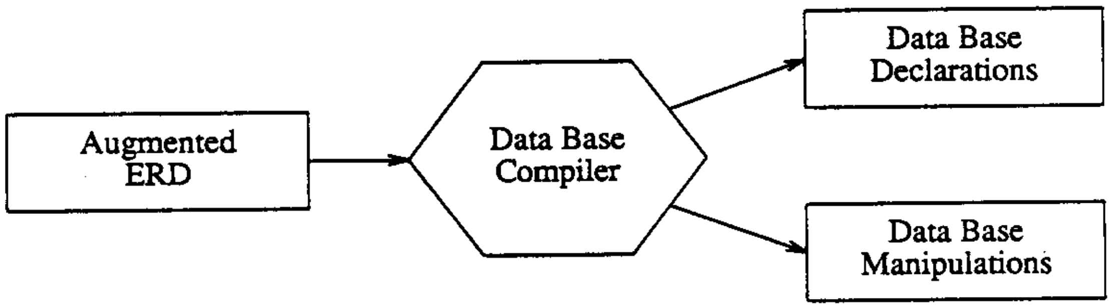


Figure 4.1 Data Base Compiler

Data Base Compiler HIPO list:

I. NAME:

Data Base Compiler (DB_COMPILER) - Part of the SYSGEN

(Figure 3.8) system component. The DB_COMPILER is the processor within the Integration Facility (Figure 3.8) which is responsible for the coupling of the tool and the data base management facilities.

II. INPUT:

The Augmented ERD (Figure 1.2) consisting of:

- defined entities
  - a. properties of entities
  - b. relationships between entities
  - c. roles entities assume
  - d. entity origin

Further ERD textual augments (Section 5.5) consisting of:

- constraints on entities:
  - a. type constraint
  - b. value constraint
  - c. structural constraints
  - d. procedural constraints
- access definition

III. PROCESSING:

The inputs defined above are sufficiently rich in information so that both the logical data base model can be defined as well as the tool's physical data base. The following actions need to be performed:

- traversal of the augmented ERD and ERD textual augments to identify types of entities, relationships and properties,
- mapping of entities, relationships and properties into an extended relational model, the RM/T data model [Codd79],
- mapping of the RM/T data model into the relations supported by the underlying relational data base, and
- mapping of the constraints, options list, defined unit and defined methods into catalogs and procedures supported by the DB_KERNEL.

The RM/T has been chosen as the logical data model for this research as a result of a comprehensive review of the extended relational model indicating that the RM/T provides:

- a natural correspondence with the ERM, the conceptual model for SARA/IDEAS system;
- structural representation for both the atomic and molecular semantics of complex designs;
- high level operators to support storage and retrieval

of complex units;

- support for system generated id's, necessary for management of the complex associations found in designs.

IV. OUTPUT:

Data Base Declarations consisting of:

- catalogs defining the RM/T data model,
- catalogs defining the external links to other tools and the system

Data Base Manipulations consisting of:

- augments to the data base procedures provided by the DB_KERNEL - these are in the form of T code, the implementation language for SARA/IDEAS.

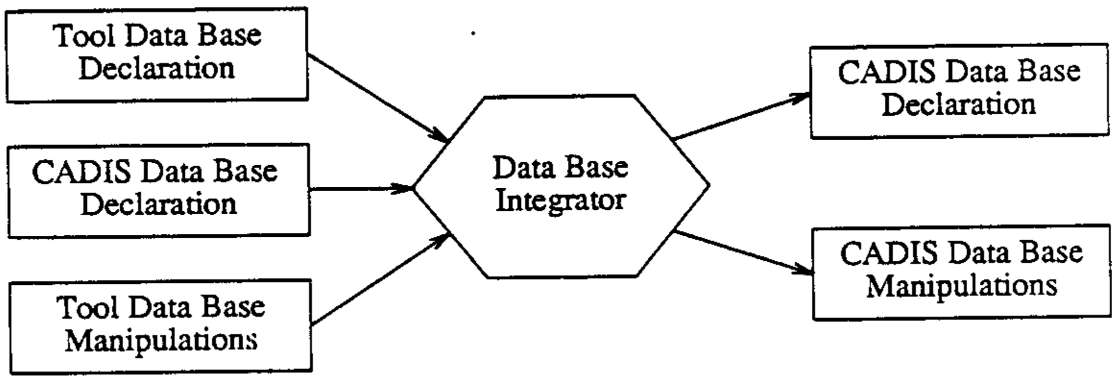

Tool Data Base Declaration
CADIS Data Base Declaration
Tool Data Base Manipulations
Data Base Integrator
CADIS Data Base Declaration
CADIS Data Base Manipulations

Figure 4.2 Data Base Integrator

Figure 4.2 is a data flow representation of the Data Base Integrator followed by a HIPO list describing it in more detail. This processor is a sub processor found within the Integration Facility Processor illustrated in Figure 3.8

Data Base Integrator HIPO list:

I. NAME:

Data Base Integrator (DB_INTEGRATOR) - Part of the SYS-GEN (Chapter Three) system component. The DB_INTEGRATOR is the processor which is responsible for integrating the tool database into the existing CADIS database.

II. INPUT:

Tool Data Base Declaration consisting of:

- catalogs defining the RM/T data model
- catalogs defining the external links

Tool Data Base Manipulations consisting of:

- augments to the data base procedures provided by the DB_KERNEL

CADIS Data Base Declarations consisting of:

- catalogs defining the existing tools' schema

III. PROCESSING:

The following actions need to be performed:

- processing of the RM/T catalog information to identify meta data for use at the system level
- processing of the external catalog definitions to identify origin of objects
- binding of external object references at system level

IV. OUTPUT:

The DB_INTEGRATOR produces the following

- CADIS Data Base Declaration - catalogs containing the system schema
- CADIS Data Base Manipulations - augmented methods to access the tool schemas

Tools

The tools of the CAD system include not only tools that define and manipulate CAD objects, but also for example, allow the user to browse through objects found in a component library - a librarian. Chapter Eight describes the structure of a data intensive tool, a browser. These tools constitute the variant aspect of CADIS. As new tools are added, the character of CADIS changes.

Additional Components

Additional mechanisms necessary to support the design environment are those mechanisms which are often found in software development systems. Often these mechanisms are considered software tools. However, we regard tools to be those designed to carry out a specific task directly related to the CAD system's primary goal. A mechanism is more general in nature. An example of one such mechanism is the RCS program [Tich82b]. RCS (Revision Control System) manages software libraries. It provides the following functions for such management:

- RCS stores and retrieves multiple revisions of program and other text.
- RCS maintains a complete history of changes.
- RCS manages multiple lines of development.
- RCS can merge multiple lines of development.
- RCS flags coding conflicts.
- RCS provides release and configuration control.

The authors state that this program is compatible with existing software and is unobtrusive so that existing tools can be used as before. This mechanism would be combined with the data management facilities to provide versioning of the database information. As such its place is within the underlying system components and not as an independent tool.

4.4 CADIS Architecture

The CADIS database architecture (figure 4.3) is multilevel. The four levels of CADIS have the indicated correspondence with the ANSI/SPARC three level architecture.

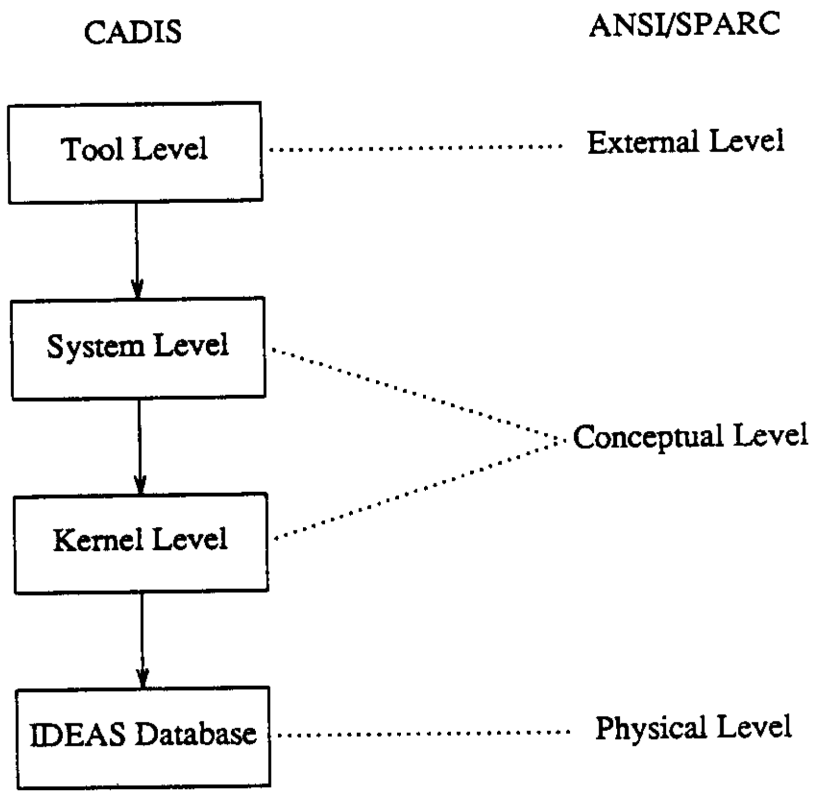


Figure 4.3 - CADIS Database Architecture

4.4.1 CADIS Tool Level

The tool level represents the external level of the IDEAS database. There exist at this level multiple tools, each with their own view of the database. An end-user (designer) invoking a particular CAD tool, such as the SM Editor, is concerned primarily with that tool's conceptual view of the design representations associated with it. This conceptual view is originally defined by the tool implementor and is based on the Entity Relationship Data Model (ERM) [Chen77] which has been augmented by this author. Unlike view definition found in traditional data base management systems, these tool views are not necessarily defined as a subset of the original data base schema, but can be defined independently of the data base schema and then integrated into the existing model.

CADIS tools include all of the tools described in Section 3.3. Chapter Five describes in detail two of these tools, the SM Editor and GMB Editor, and their ERM design representations. Chapter Eight also details a tool, the SI Browser, and its related ERM design representation.

4.4.2 CADIS System Level

We use an Entity-Relationship Diagram (ERD) (Figure 4.4) in order to describe the semantics of the system level's position within the CADIS architecture.

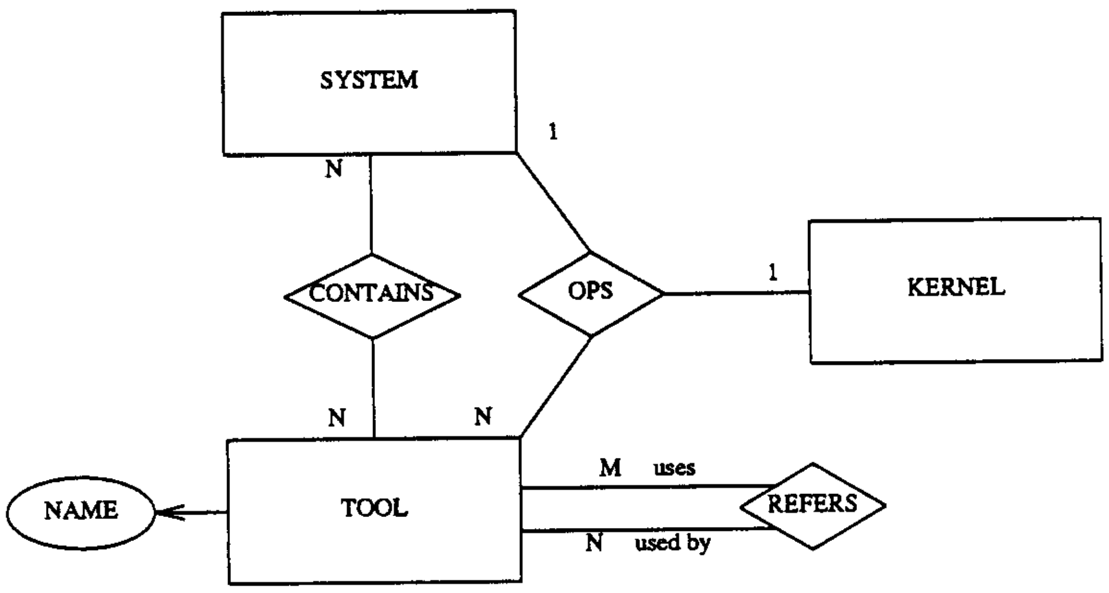


Figure 4.4 - ERD of the CADIS Enterprise

The tool developer, while in the process of defining a tool's objects, relationships, and operations, has a view of CADIS as a system object with constituent entities - system, kernel, tool. The tool developer's role is that of creating a new entity of type TOOL and also creating any new relationships of type REFERS, defining the role of the tool in this relationship. Instances of the system level relationship type REFERS allow a tool to use objects defined in other tools. Another system level entity of relationship type OPS is created if the tool developer needs to reference any of the objects found within the system. Such objects may be a protection object or configuration object. An instance of the system level entity of relationship type CONTAINS is created during the process of integrating the tool's view into CADIS. The tool developer may also classify tool objects as belonging to particular system-defined classes, objects. These system generic objects provide the tool developer with database access operations and update operations.

4.4.3 CADIS Kernel Level

The kernel level of CADIS represents the conceptual level of the IDEAS data-base. The kernel level supports an underlying extended relational data model [Codd79] which has been modified so as to accommodate design-specific require-ments such as the support of version control mechanisms and storage/retrieval of unstructured data. This data model (RM/T) is described in detail in Chapter Six.

The kernel level supports the semantics of the ERM through basic operations defined on that model - selection, deletion, update and insertion. Most information retrieval and storage requests can be considered as a combination of the following basic types of operations found in the kernel:

Selection

1. Selection of a subset of values from a value set.

2. Selection of a subset of entities from an entity set.

3. Selection of a subset of relationships from a

relationship set.

4. Selection of a subset of attributes from an attribute set.

Deletion

Deletion only involves deletion of an entity or a relationship. The consequence of deletion is that all properties are deleted. The separation of properties from value sets however, allows for the deletion of properties without affecting similar properties found in other entities. Another consequence of entity deletion is that dependent entities will be deleted as well as relationships involving the deleted entity.

Update

Updating only changes the value of attributes of entities or relationships.

Insertion

Insertion requires only three operations:

1. Insertion of an entity into an entity set.

2. Insertion of a relationship into a relationship set.

3. Insertion of properties of an entity or relationship.

Chapter Five defines the set of operations necessary to define and manipulate the augmented ERM, using these basic operations.

The partitioning of the ANSI/SPARC conceptual level into two distinct levels (system-kernel) is necessary in order to achieve the objectives of providing for an extensible system, ease of use of the CAD system, and ease in developing new tools. The schema associated with the kernel level is less dynamic than that of the system kernel. It too is modified as new tools are brought into IDEAS. However, the modification involves basically the addition of new relational tables. The tool-independent functions necessary to support the kernel view are integrated into the SYS/KERNEL which provides the CAD system with a nucleus of common functions necessary for the development and integration of CAD tools.

4.4.4 CADIS IDEAS Database

The physical data base used for this research is built by the PLAIN Data Base Handler [Kers81]. However, the design of the SARA/IDEAS system is such that a designer of a tool could pick another database (assuming it existed in the Kernel Library). The final binding of the tools' database activities does not occur until the run time system is generated. The PLAIN data base handler's interface language is Troll. For this reason, we refer to the data base management system as Troll. The primary reason for Troll being chosen over some other system, such as INGRES, is that Troll is a compact relational database handler available under the UNIX operating system. In fact, the goals underlying the development of Troll are primarily compactness and procedurality.

Troll was designed to be small; as such it provides a minimum set of facilities and few of the "extras" present in other systems. For example, the output of Troll is very straightforward and as such no policy has been predefined which dictates the format of such output. In this regard Troll meets the requirement of the kernel concept by providing only the mechanisms but not the policy.

Troll was also designed to carry out steps one at a time in a user specified order. That is, no query optimization has been determined. This is a beneficial feature when extending Troll to operate on models other than the relational model. Again, the use of Troll with various implementation models enables experimentation with other ways of representing data.

Troll was designed to provide information about system performance especially the time required to carry out certain operations and I/O activity. This facilitates experimentation with database design and selection of organizations that perform well for the desired set of operations.

Troll was designed to accommodate a variety of interfaces to a shared database. This strategy yields great flexibility in providing the most suitable interface for different user classes and different applications.

Disadvantages found while using Troll might lead system developers to later choose another underlying data base management system. Three disadvantages encountered during this research endeavor were: a lack of support for concurrent usage of the database; lack of an indexing capability needed to improve response time to frequently accessed information, and an insufficent and inconsistent recovery procedure.

4.5 Summary

This chapter has defined an information system for support of the design process (CADIS). Unlike other efforts at developing such a system, the data management facilities are tightly integrated into the design process. That is, there are facilities to manage the evolving system as well as an evolving design. The methodology presented by [Worl86] provides a means by which the CAD system developer is able to specify the database structure and operations while designing a tool for use with the CAD system. CADIS is designed to be an integral part of this process.

CHAPTER 5

The Kernel Level

The nucleus of CADIS, the DB_KERNEL, consists of mechanisms necessary for the management of the design database. These mechanisms are considered the tasks of the DB_KERNEL. The DB_KERNEL's primary task is support of the user's conceptual model. This conceptual model is based on the Entity-Relationship Model (Chapter One). Although the ERM is a substantial and powerful design tool which can aid in the database design process, the complexity of design objects and the iterative, recursive nature of the design process necessitates augmenting the model. These augmentations, we claim, increase the semantic expressiveness of the ERM for use in a design environment; provide a flexible means for describing the semantics; and allow the tool developer to specify pertinent database access requirements. The focus of this chapter is on developing a definition for each of five classes of augmentation needed to support a design environment. The resultant ERM and its graphical representation, the Entity-Relationship Diagram (ERD) is then used by a tool designer for expressing the structure and semantics of any particular CAD tool.

This chapter begins by defining the objectives of the DB_KERNEL. Descriptions, using the basic ERD, of the tools needed in the SARA/IDEAS system are presented. These descriptions reveal weaknesses and serve as examples when defining the five classes of augmentations. The concluding sections of this chapter complete the DB_KERNEL design with a presentation of salient features of the augmented ERM's definition and manipulation operations.

5.1 DB_KERNEL

The primary hypothesis of this research is that a flexible and hence, more powerful data base management system can be developed to serve as a component of the kernel of a CAD environment. The development of such a system is based upon the separation of *mechanism from policy*. This separation provides the framework for flexibility within a data base management system. Since we have found many similarities between the design of such a system and the design of operating systems, many guidelines have come from the operating systems field. Separation of policy from mechanism is a widely supported principle in operating system design, as indicated by the following quotation from Peterson and Silberschatz [Pete83]

> One very important principle is the separation of *policy* from *mechanism*. ...
>
> The separation of policy and mechanism is very important for flexibility. Policies are likely to change from place to place or time to time. In the worst case, each change in policy would require a change in the underlying mechanism. A general mechanism would be more desirable. A change in policy would then only require redefining certain parameters of the system.
>
> Policy decisions are important for all resource allocation and scheduling problems. Whenever it is necessary to decide whether or not to allocate a resource, a policy decision is being made.

Likewise, we find that policy decisions can affect the design of a data base management system. These decisions are handled by the data base administrator (DBA), who also is responsible for the implementation details. A separation of policy issues from mechanism issues however, is never clearly defined by designers of data management systems. Since we can point out specific areas within these management systems which are similar to those found in operating systems, it is reasonable to suggest that such a separation of policy from mechanism is necessary for the development of a flexible management system.

The philosophy underlying the design of CADIS clearly delineates between mechanism and policy. The resultant architecture of CADIS supports this separation. Although policy features can be implemented within CADIS, we emphasize that the DB_KERNEL provides only the mechanisms. The SYSTEM level provides the capability to implement policy decisions. A description of the SYSTEM level is provided in Chapter Seven.

5.2 DB_KERNEL Objectives

The DB_KERNEL provides the mechanisms needed to support data management in a design environment. Major objectives have been established for the DB_KERNEL and serve as high level requirements of this kernel. These objectives are:

- to provide operations to create a representation of the user model in the tool database.
- to provide operations to store the user representation so that tool semantic information can be contained in the database
- to provide operations on instances of the user model
- to allow for the automation of data definition so as to ease the burden of the tool designer
- to provide access to system objects which the tool developer can use to ease the burden of tool design and integration

The first two objectives can be met by requiring that the DB_KERNEL contain data definition facilities for defining storage structures capable of representing the semantics associated with the design. These semantics, represented by an augmented ERM structure, plus additional semantics, such as are found in explicit constraint specifications, need to be supported by the DB_KERNEL. Section 5.4 defines this augmented structure, whereas section 5.5 defines the operations needed to build this augmented structure. The third objective is met by defining operations on the ERM such as insert, retrieve, modify entities and/or relationships. The last two objectives are expanded upon in Chapter Six (automation of data definition) and Chapter Seven (system objects) of this dissertation.

The following section (5.3) digresses in order to describe the graphical representation of the ERM. This graphical component of the ERM, the ERD, is used in section 5.4 to describe, in terms of objects and relationships, two of the SARA/IDEAS tools; the SM Editor and GMB Editor.

5.3 Entity-Relationship Diagram

Recall from Chapter One that the Entity-Relationship Model, as originally defined by Chen, [Chen77] provides the analyst with three main semantic concepts: entities, relationships and attributes. Designers of database systems make use of the ERM by describing the enterprise in terms of

- entities, which are distinct objects within a user enterprise;
- relationships, which are meaningful interactions between the objects;
- attributes, which describe the entities and relationships

This section defines the basic ERD so that we can use this as a base to build an augmented structure. The augmented ERD will provide a non database designer with the capability of generating a database schema for use in the storage and manipulation of objects produced by the tool being developed.

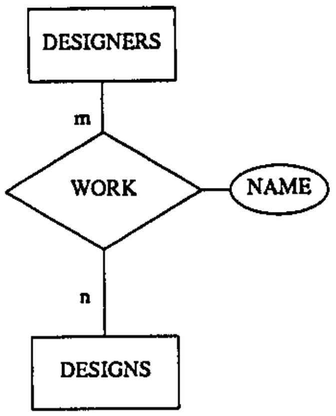


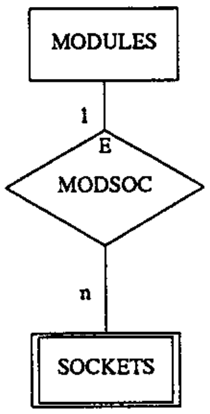
Figure 5.1 Example ERD

Basic ERD

Entities and relationships can be represented diagrammatically by an Entity-Relationship Diagram (ERD). Figure 5.1 illustrates an ERD that depicts two entity sets labelled DESIGNERS and DESIGNS and an m to n relationship between them labelled WORK. The semantics associated with this ERD indicates that many entities in the set labelled DESIGNERS work on many entities in the set labelled DESIGNS and also that a particular designer working on a particular design belongs to a named entity labelled *WORK*. The ERD allows a designer to also represent 1:1, 1:N relationships as indicated by the numbers associated with the arcs.

ERD Constraint Specifications

Figure 5.2 and 5.3 illustrate the provisions found in the original ERD for the representation of existence and identification dependency constraints. An existence-dependency is a constraint requiring that an instance of an entity type depend upon the existence of another specific entity. Figure 5.2 indicates that the entity type SOCKETS is existent-dependent upon the entity type MODULES via the relationship MODSOC. An identification constraint, on the other hand, merely serves to restrict the primary key of an entity to the concatenation of the parent's key and the child key. This is semantically equivalent to saying that the identification dependent entity is not able to be identified through its own unique key, but requires a combination of keys in order to be uniquely identified. Figure 5.3 indicates that the entity type INTERCONN can only be uniquely identified through the entity type MODULES which it is related to in the relationship set MODINT. An entity which is existent-dependent is not necessarily identification dependent upon another entity. Both the existence and identification dependencies are defined only in the case of a 1:n mapping.



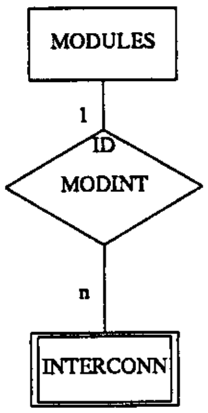


Figure 5.2 Existence Dependency   Figure 5.3 Identification Dependency

An existence-dependency is graphically noted by

- a double box around the dependent entity, and
- an E in the relationship through which this dependency holds, and
- a 1:n mapping.

All three conditions must hold in order to express such an existence-dependency.

ERD Recursion Representation

The ERD also allows a user to represent recursive relationships. This capability is especially important in the design domain since the design process itself is recursive in nature. That is, a design is a refinement of a design which may be a refinement of yet another design. Figure 5.4 illustrates the ERD representation of a recursive relationship. Roles, which define a subset of entities taking part in the relationship, are indicated on the lines connecting the entity set to the relationship.

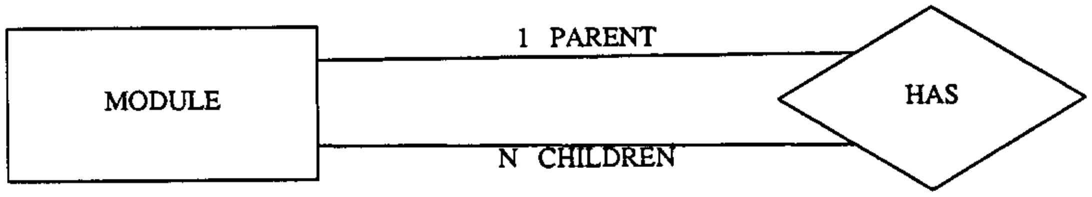


Figure 5.4 Example ERD

5.4 SARA/IDEAS Tools

5.4.1 SM Editor Description

The Structural Model is one of the original SARA tools (Chapter Three). When faced with the prospects of moving SARA from MIT to UCLA, the SARA research group decided upon a complete redesign of the tool. Following the Worley methodology, [Worl86] as defined in Chapter Three, the designers of the SM Editor [Cai85] designated three primitives as objects; modules, sockets, and interconnections. Appendix A contains the T [Slad87] implementation of this tool in terms of these primitives. The SM Editor is the SARA/IDEAS tool which provides for the construction of a Structure Model. This model is relatively simple. There are only a few constraints placed on the objects. The explicit semantics of the SM (Chapter Three) are the following:

1. There is a top level module *Universe* which has no sockets and only contains other modules;

2. Modules themselves can contain other modules;

3. Modules have sockets which provide ports to the environment outside the module;

4. Modules connect with other modules via an interconnection;

5. An interconnection is not directed and can connect only two sockets;

6. A socket can have at most one outside interconnect and one inside interconnect; (A more recent version removes this constraint. For purposes of this chapter we will retain this constraint and then demonstrate the effect the change has on the model in chapter Seven.)

Implicit semantics of the SM are:

1. A module can not exist if there is no universe;

2. If a socket is removed, the corresponding interconnections are deleted thus modifying the information regarding the other affected sockets;

3. If an interconnection is disconnected from one socket it is disconnected from its other socket;

4. Interconnections cannot exist if not attached to two sockets;

5. Identification of a module is through a fully qualified path name, the concatenation of the names of all of the module's ancestor's names;

6. If a module is deleted then all of its sockets and hence the corresponding interconnections are deleted. This also can affect other sockets.

The corresponding ERD for the Structure Model Tool (Figure 5.5) shows four objects, their relationships to each other and also the semantic constraints on the objects and relationships. The following list points out the semantics expressed in the

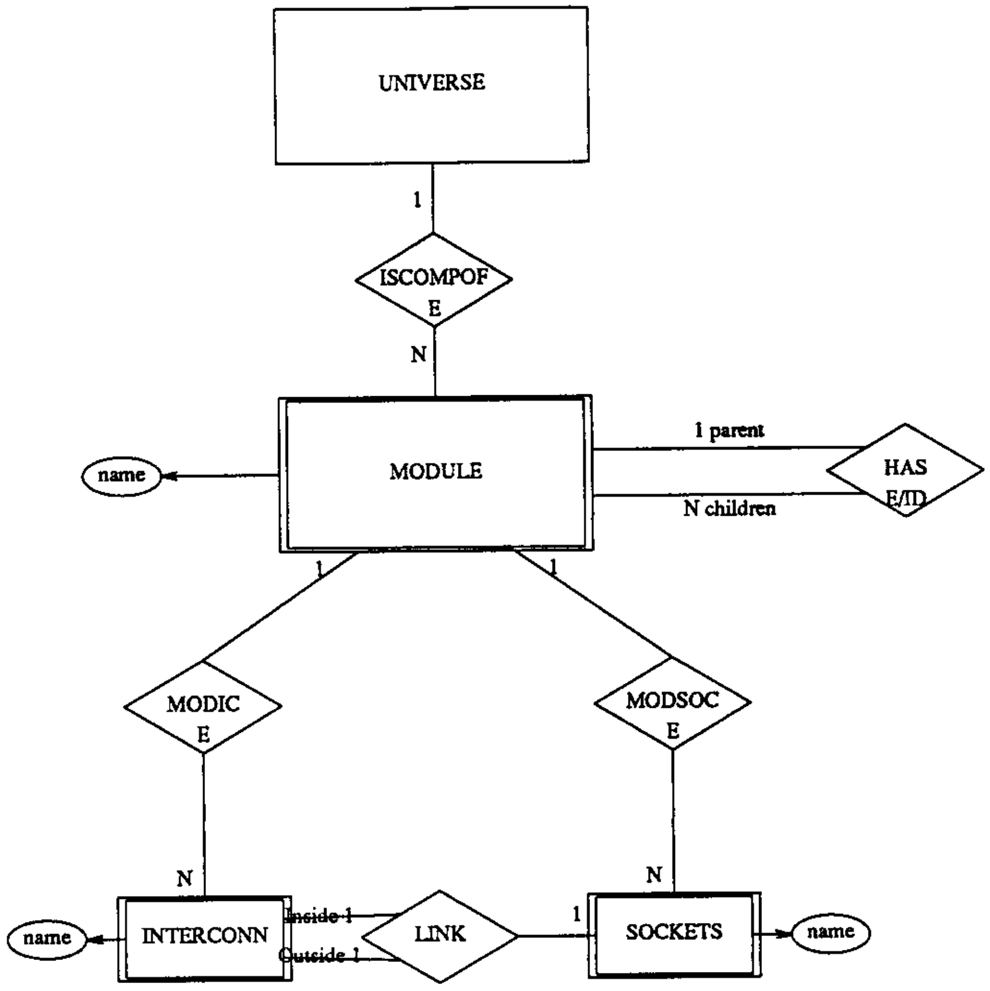


Figure 5.5 ERD for the Structure Model Tool

ERD.

1. The ERD entity labelled UNIVERSE represents an object which participates in a relationship labelled ISCOMPOF with the entity labelled MODULE. The tool designer has a need to represent the condition that the Universe is composed of modules. In order to indicate this condition she/he has chosen the label ISCOMPOF as a symbol representing the condition *is composed of*. The remaining portions of

the editor ERD representations will consistently use this label to indicate composition. References to the objects modules, sockets, and interconnections will use the ERD labels MODULE, SOCKET, and INTERCONN.

2. MODULE is existent dependent on the UNIVERSE. This constraint is indicated by the placement of "E" in their relationship ISCOMPOF, the double box surrounding MODULE, and the 1:N relationship mapping.

3. A MODULE is shown as participating in the recursive relationship HAS, and takes on either the role of *parent* or *child*. The 1:N mapping and the presence of an E/ID in the relationship indicates that the *child* can only be uniquely identified via its *parent* and that the *child* can not exist in the database without a *parent*.

4. SOCKETS are indicated as being existent dependent on their parent MODULE. INTERCONN also have this constraint.

5. SOCKETS can have at most one outside interconnect and one inside interconnect associated with them (indicated by the mapping 1). SOCKETS and INTERCONN participate in a 3 - nary relationship LINK.

6. MODULE, INTERCONN, and SOCKETS each have the property *name* defined for them.

Figure 5.5 represents an initial attempt at describing the semantics of the SM Editor using the basic ERD. Some of the limitations with this representation are:

- The composition of MODULE, SOCKETS, and INTERCONN creating the aggregate object UNIVERSE is not represented explicitly. Although the relationship between MODULE and UNIVERSE is clearly marked as being ISCOMPOF, there is no means of easily representing the fact that SOCKETS and INTERCONN are also included. Connecting SOCKETS and INTERCONN directly to the relationship ISCOMPOF only serves to complicate the representation, because the question would arise as to the status of their relationships MODSOC, MODINT, and LINK with the object UNIVERSE.

- There is no means of representing the constraint that prohibits the existence of an INTERCONN if it is not connected to a SOCKET. The placement of an "E" within the relationship LINK would be ambiguous since there is no means, using the basic ERM, of indicating the direction of a 1:1 existence dependency.

- The constraint which limits the uniqueness of names to sibling MODULE(S); MODULE(S) at the same level in the hierarchy, is not able to be unambiguously expressed. The existence dependency notation in the relationship HAS, indicating that a child MODULE is uniquely identified via its *parent's* identifier does not express this constraint. If the designer designates that the property of MODULE, name, is a unique identifier, then an inconsistency is introduced in the model. That is, the representation expresses that a MODULE'S name is unique as well as its uniqueness being dependent upon another MODULE.

- There is no means of expressing that retrieval of a MODULE should include its SOCKETS and INTERCONN as well. In the basic ERM, entities and relationships are the most likely candidates to serve as the unit of retrieval. Most ERM-based systems access these units. However, the designer needs to be able to express the desire that a more complex structure is to be considered as the unit of access.

Augments to the basic ERM are required in order to overcome such limitations.

5.4.2 GMB Editor Description

Recall that the Graph Model of Behavior (GMB) describes both the data flow and the control flow of a hardware/software system. The GMB tool developer conceptualizes the model as comprised of the gmb object, which contains a control-graph object, and a data-graph object. These objects also contain other objects such as nodes, control arcs, data arcs, data sets, processors and interpretation code. Control arcs are associated with the sockets defined in the SM Editor tool(see 5.4.1). Appendix B contains a listing the T code description of this tool's objects. It is apparent, from the amount of code needed to describe the GMB, that the model of the GMB is of greater complexity than that found in the SM Editor. The following list represents the complete description of the GMB object:

1. GMB is an object (Figure 5.6)

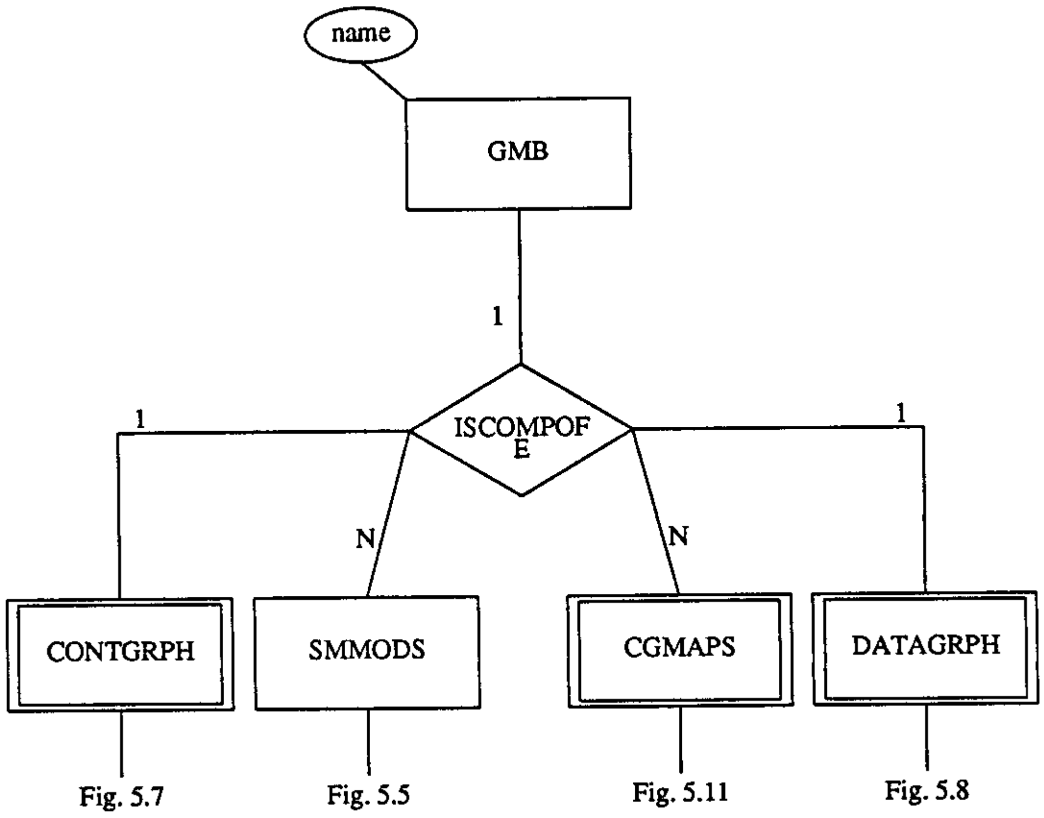


Figure 5.6 ERD for the GMB Tool

It consists of (ISCOMPOF) the following attributes and objects:

a. name: an attribute identifying the GMB Object;

b. CONTGRPH (Control Graph): an object which contains the control flow information;

c. SMMODS (Sm Modules): a set of objects defining the module objects mapped to the gmb object;

d. CGMAPS (Control Graph-Data Graph mapping): a set of objects defining the mapping of control nodes to data processors.

e. DATAGRPH (Data Graph): an object which contains the data processing information;

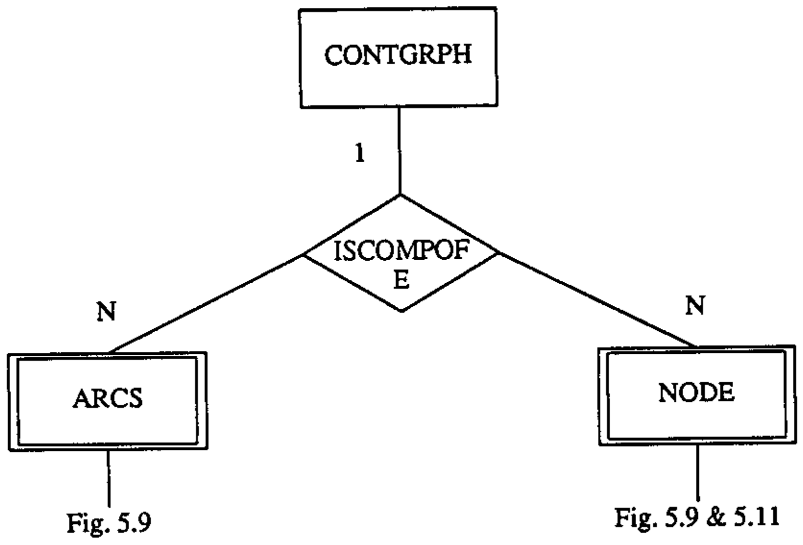


Figure 5.7 Control Graph ERD

2. A CONTGRPH (Control Graph) is an object (Figure 5.7):

It consists of (ISCOMPOF) the following objects:

a. ARCS (control arcs): a set of control arc objects associating control nodes;

b. NODE (control nodes): a set of control node objects;

3. A DATAGRPH (Data Graph) is an object (Figure 5.8)

It consists of (ISCOMPOF) the following objects:

a. PROCESSOR (data processors): a set of data processor

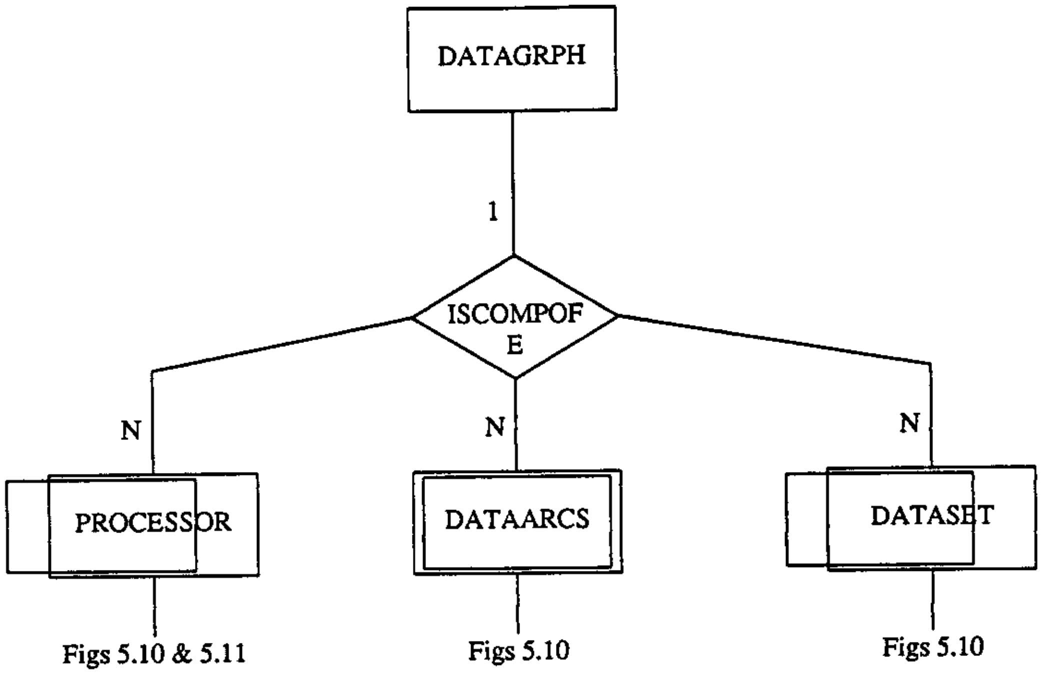


Figure 5.8 Data Graph ERD

objects;

b. DATAARCS (data arcs): a set of data arc objects associating data sets and data processors;

c. DATASET (data sets): a set of data set objects;

4. A NODE (Control Node) is an object (Figure 5.9)

It consists of the following attributes:

a. que (queue): an attribute specifying the queuing distribution;

b. name: an attribute identifying a unique instance of a node;

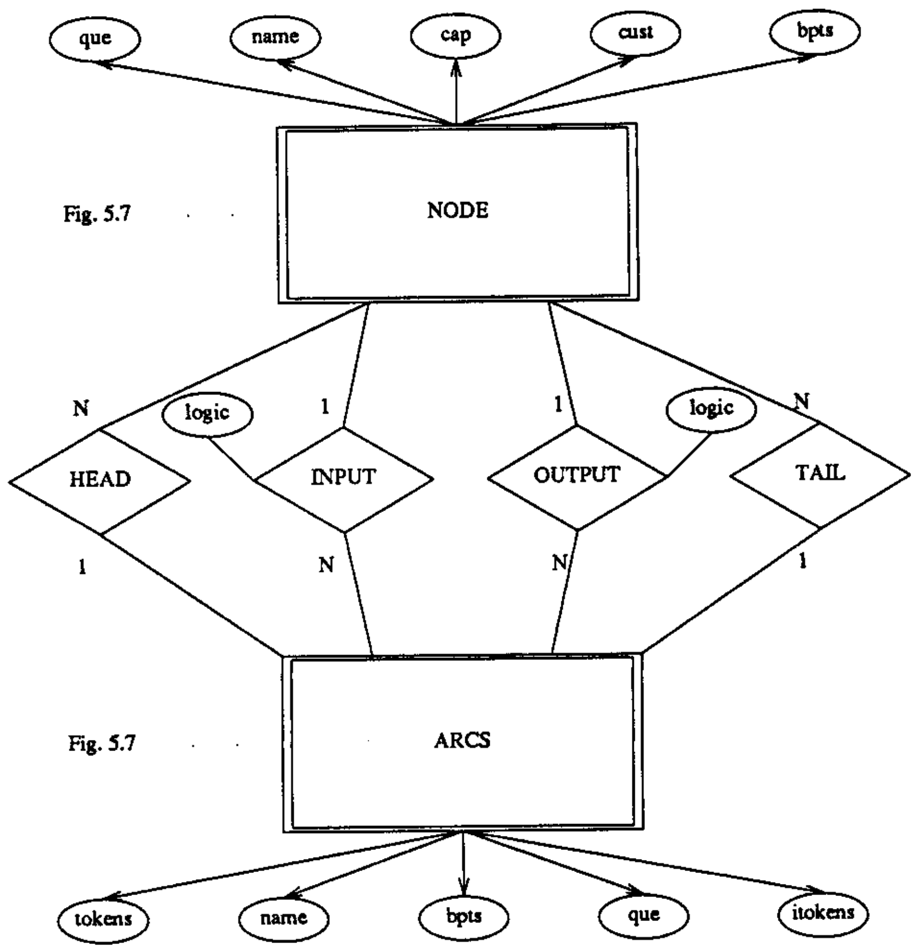


Figure 5.9 Control Nodes and Control Arcs

c. cap (capacity): an attribute specifying the number of concurrent initiations of the node;

d. cust (customer): an attribute specifying the current number of tokens in the queue;

e. bpts (breakpoint): an attribute specifying if this node is a breakpoint or not.

The Control Node enters into the following relationships (Figure 5.9):

a. INPUT (input arcs): the relationship entity describing the set of control arcs which enter the control node;

b. OUTPUT (output arcs): the relationship entity describing the set of control arcs which exit the control node.

The property LOGIC is defined for both the relationship set INPUT and the relationship set OUTPUT. This property represents the logic expression (Section 3.3.4) associated with the relationship between a control node and its input/output arcs.

5. ARCS (Control Arc) is an object (Figure 5.9)

It consists of the following attributes:

a. tokens: an attribute specifying the number of current input tokens;

b. name: an attribute identifying the specific instance of an arc;

c. bpts (breakpoints): an attribute specifying if placement of a token causes a break in simulation;

d. que: an attribute specifying queuing information;

e. itokens: an attribute specifying the number of initial input tokens;

The Control Arc enters into the following relationships (Figure 5.9):

a. HEAD (headset): the relationship entity describing the set of control nodes at the head of the arc;

b. TAIL (tailset): the relationship entity describing the set of control nodes at the tail of the arc.

6. A DATASET (Data Set) is an object (Figure 5.10)

It consist of the following attributes:

a. NAME: an attribute identifying the object;

b. VAL: an attribute specifying the current value of the dataset.

c. IVAL: an attribute specifying its initial value;

The Data Set enters into the following relationships (Figure 5.10):

a. DARCS: the relationship specifying the set of data arcs which are mapped to this set;

7. DATAARCS (Data Arc) is an object (Figure 5.10):

It consists of the following attribute:

a. NAME: an attribute identifying the object;

The Data Arc enters into the following relationships (Figure 5.10):

a. DSETS: the relationship specifying the set of datasets the arc is mapped to;

b. DPROCS: the relationship specifying the set of data processors

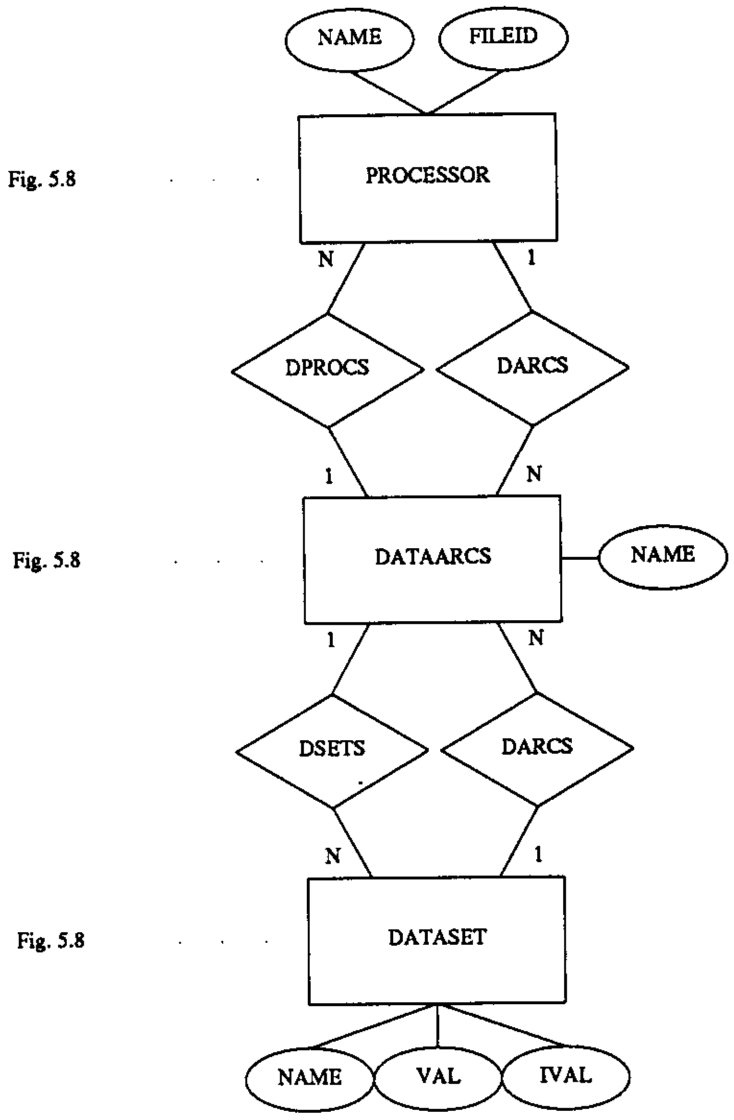


Figure 5.10 Data Sets, Data Arcs and Data Processors

the arc is mapped to;

8. A PROCESSOR (Data Processor) is an object (Figure 5.10):

It consists of the following attributes:

a. NAME: an attribute identifying the object;

b. FILEID: an attribute specifying the file identifier of the interpretation code.

The Data Processor enters into the following relationship (Figure 5.10):

a. DARCS: the relationship specifying the set of data arcs the processor is mapped to.

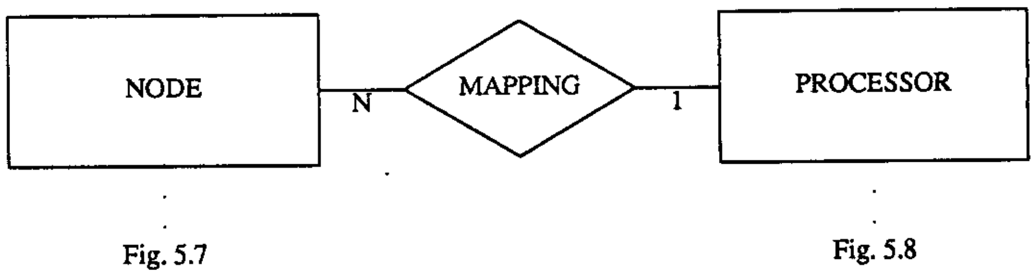


Figure 5.11 Control Node - Data Processor Mapping

Additionally, a relationship MAPPING (Figure 5.11) is defined between the NODE of Figure 5.7 and the PROCESSOR of Figure 5.8. The explicit semantics of this relationship are:

- A Control Node is mapped to zero or one Data Processor (Figure 5.11);
- A Data Processor is mapped to a set of Control Nodes (Figure 5.11);

Additional explicit semantics associated with the GMB, not found in the graphical representations of Figures 5.6 - 5.11 are:

- The input/output logic expression attributed to the relationship set LOGIC needs to reference the arcs according to their position in the relationship. For example, 1 * 2 + 3, is the logic expression for the AND of the first and second arc in the relationship ORed with the third arc;
- Data Arcs are directed. The source of a Data Arc can be a Data Processor with its destination a Data Set and/or Socket. The source can also be a Data Set with its destination a Data Processor and/or Socket.

Implied semantics of the GMB which need to be represented are:

- If the set of control nodes a data processor is mapped to becomes empty, the data processor is deleted. This will cause deletion of the arcs into and out of the affected data processor and hence modification of the data sets and sockets which are the source or destination of those deleted arcs.

Figures 5.6 - 5.11 represent an initial attempt at describing the semantics of the GMB editor using the ERM. Some of the limitations with this representation are:

- Again the composition of objects creating such aggregate objects as the GMB, the Control Graph, the Data Graph, the Control Graph-Data Graph Mapping, is represented using the cumbersome technique which requires that the designer define a separate relationship, ISCOMPOF, for each instance needing to express composition and/or decomposition.

- There is no means in the ERM for representing an existence dependency on a set of entities. The basic ERM is not able to represent the constraint that a single (1) entity is dependent upon a set (N) of entities. This constraint must be handled explicitly through additional programming statements. In the GMB model we note that this limitation occurs in Figure 5.11 since the PROCESSOR (the 1 entity) object needs to be dependent upon the N entity, NODE, according to the implicit semantics of the GMB editor tool.

- There is no provision in the ERM for representing the order of the entities participating in a relationship. Therefore, we are unable to represent that a subset of entities must be ordered in some way. This limitation is needed to express the complete semantics of the relationships INPUT and OUTPUT (Figure 5.9).

- There is no provision which supports primitive types other than integer, real, boolean, string, and character. The property FILEID (Figure 5.10) refers to a body of interpretation code. The designers place in this attribute a file identifier. However, this knowledge is restricted to the domain of the tool and is not known outside of this domain, thus limiting the semantic expressiveness of the model.

- There is no provision for the representation of directed relationships. The basic ERM assumes that relationships are bidirectional, when in fact, there are many instances when the direction needs to be restricted to one way. Direction is implied in the representation of existence dependencies and identification dependencies. For example in Figure 5.6, the ISCOMPOF implies the direction goes from top to bottom. In the basic model, the burden of implementing a unidirectional relationship is placed on the developer of the tool. We find in Figure 5.10, we are not able to represent the directionality of the Data Arcs which are associated with the Data Processors and Data Sets.

The following sections describe in detail the ERD augments needed so as to overcome the limitations presented in 5.4.1 and 5.4.2.

5.5 Augments to the ERD

We have previously stated that the goals of the tool developer are to provide, via some model, a complete description of the objects which the tool will create, modify and/or use (Chapter Three). This description is enriched if it includes all possible constraints that may be placed on the objects and their relationships during the design session. Constraint specification is one of the means by which the semantic understanding of the tool can be enhanced. Therefore, we have augmented the ERD by enlarging the constraint representation capability which the basic model supports. A tool developer also needs to be able to represent the various processes which a particular tool employs. We have seen that the design process makes use of both the process of composition and decomposition of objects, as well as the process of classification. Abstraction is a concept which allows for the representation of composition and decomposition of objects and/or relationships and the classification of the same objects and/or relationships. We have incorporated within the augmented ERD these abstraction capabilities as well. While developing a tool, the designer, although not a data base expert, has some perception of the level of granularity necessary for the retrieval and storage of the objects and relationships being defined. That is, the tool might need to have an object along with associated objects retrieved as a unit. Thus, we have provided to the tool developer the ability to specify some aspects related to storage parameters via ERD objects and relationship representation.

The following three subsections describe in greater detail the concepts of constraint specification, abstraction representation, and storage specification.

CONSTRAINTS

Constraints are considered to be three dimensional. That is, a constraint can be classified as to whether it is:

1. a static constraint or dynamic constraint
2. an explicit or implicit constraint
3. a structural constraint or value constraint.

A static constraint is one which defines the allowable values an attribute may have at any time during its existence, whereas a dynamic constraint specifies restrictions only during transitions of database states.

Explicit constraints are those constraints specified within the ERD such as: *an instantiation of type SOCKETS* may not exist unless it is related to an instantiation of type MODULE. Implicit constraints, however, are constraints implied by the semantics of the diagram. The basic ERD has very few implicit constraints associated with its structure. One possible implicit constraint would be the requirement that entity set membership be specified via a predicate.

Value constraints are those constraints specified with domains or relationships, such as requiring a domain to be a subset of the integers, or a relationship to be a 1 to n mapping class. Structural constraints are constraints on the interrelationships of entities. One such constraint found within the ERD is the explicit constraint requiring an instance of one member of an entity set be identified via an instance of member in another entity set (Identification dependency).

The basic ERD supports minimal constraint specification within these three dimensions. The augmented ERD expands the constraint specification capability of the tool designer by providing additional abstractions and constraint capabilities built into the model.

ABSTRACTIONS

Abstraction is one means of structuring and visualizing information. It is basically the ability to hide detail and concentrate on the general, properties of a set of objects. Abstraction is used in two ways: generalization and aggregation. Generalization enhances understanding by allowing classification of objects, whereas aggregation is the abstraction by which an object is constructed from its constituent objects. We have seen (section 5.4.1 and 5.4.2) that these abstractions are useful concepts employed within a design environment. The augmented ERD provides for the representation of generic and aggregate objects.

STORAGE SPECIFICATION

Storage specification capabilities are not found at all in the basic ERD. However, in the design environment it is necessary to allow at least the tool designer to designate the unit of access, whether it is an entire object or only a part of an objects. The designer may also need to indicate that certain parts of the design are to be considered a unit, even though the part may consist of multiple objects; and/or cluster a group of objects, so that they are stored in a common area for faster retrieval. The ability to represent such a clustering can be represented within the augmented ERD. Access paths are also indicated via the introduction of restrictions on relationship directions. Similarly, the capability to designate subsets of objects within the augmented ERD allows the designer to implicitly define indices, which provide faster access to groups of information. Other storage considerations would be the responsibility of a data base expert who would later fine tune the database after repeated use of the CAD tool. The flexibility found in the CADIS design is such that it supports intervention by a data base expert after the specification of the tool and generation of the tool's database.

AUGMENT CLASSES

We have augmented the ERD in order to meet the high level requirements:

1. to provide semantic capability more closely related to the design domain;

2. to provide a flexible means of tool specification;

3. to allow the model to be closer to the implementation level.

Augmentation of the ERD which is necessary to meet these requirements for support of a design environment is grouped into five (5) types:

- Augmentation to support the abstraction primitive *generalization*. Support for the generalization primitive increases the semantic expressiveness of the ERM by providing the designer with the means to represent the classification and categorization of objects and/or relationships. Through the use of generic objects, the designer can reference predefined objects and relationships which contain complex database operations. This provides the designer with an alternate way of defining database access functions and the integrity constraints which need to be associated with the tool's objects and relationships.

- Augmentation to support the abstraction primitive *aggregation.* Support for the aggregation primitive also increases the semantic expressiveness of the ERM by providing the designer with the means to model such common design notions as composition and decomposition. Flexibility in the use of the model is introduced by providing the designer with simple alternatives for expressing such relationships as ISCOMPOF or ISPARTOF or CONTAINS. Designation of such aggregate objects indicates these entities as those which are to be physically clustered within the database, so that they may be more efficiently retrieved as a unit.

- Augmentation to support expanded role concept for objects. The expanded role concept provides the designer with a means of expressing the semantics associated with a subset of entities, without having to define a completely new subclass for a given entity. Identification of these subsets of objects results in the specification of an index to be built using the key properties of the role sets. Specification of an index can be used to increase the efficiency of access to these sets.

- Augmentation to support detailed and expanded specification of properties associated with design. Expanded support of properties provides the designer with alternate ways of representing text and code. These built in types provide input/output operations for use in a manner similar to the input/output operations of primitive types such as integers. Support for the type *reference* provides the designer with an alternate means of representing referential relationships between objects. This supports an easy way of establishing hierarchical relationships, so often found in designs. [Gutt82]

Support for these new types of properties has been previously proposed and implemented by designers of CAD data base systems [Gutt82, Hask82] and is included in the ERD augments in order to have the full complement of primitive design types.

- Augmentation to expand implicit constraint specification, via the mapping representation, in order to represent more completely the semantics of design. The expansion of the mapping representation allows design specific phenomena such as order and direction of relationships to be easily modeled. Also, classification of mappings into one of three categories, can be expressed so as to enhance the constraint specification facility by allowing for dependencies on subsets of object types. Through use of the expanded constraint augmentation, the designer is able to designate access paths when defining the direction of a relationships and specify additional operational constraints by defining the ordering of relationship participants.

5.5.1 Type I Augment - Generalization

We have augmented the basic ERD to provide for the graphical representation of the abstraction primitive, *generalization*. Generalization is the abstraction concept whereby a set of similar objects is regarded as a single generic object. This abstraction is considered to be an important dimension for forming larger meaningful units, and thus represents a powerful tool for use within the design process. Providing the designer with the capability to generalize, means that classification of objects and relations as subtypes of some previously defined type of object or relation can be permitted. We have previously noted that classification occurs throughout the design process. An example of such an occurrence would be placing a developed SARA/IDEAS model into either the class of hardware or the class of software designs. Within each of these classifications, this model might be classified either as a sequential system or concurrent system. Drobman [Drob80] makes use of the process of classification in building a taxonomy of hardware and software design building blocks. A second characteristic associated with generalization is inheritance. Subclasses inherit properties and operations defined for superclasses. Using generalization in this way allows the designer to define subsets of entities which can then inherit the operations previously defined for some other entity. The SARA/IDEAS tool designer, therefore, can use one of two methods when employing the abstraction primitive generalization.

Method I

The first method is one in which the the designer defines an entity, say entity "A", using conventional ERD notation. Later another entity can be defined and designated as a subtype of entity "A". This is achieved by enclosing the name of the supertype entity A in parentheses. Use of the generalization primitive guarantees that properties and operations defined for the super entity are inherited by the subtype entity, and thus, do not need to be redefined. The system automatically produces a relationship type "ISA" for each subtype - supertype declaration. This relationship is used by the system as an access path when searching for instances of a particular entity type.

Method II

The second method, using this primitive, involves the classification of user defined entities and relationships into predefined entity and relationship types supported by the system. That is, certain generic types are already defined at the system level and may be referenced for use at the tool level. In the SARA/IDEAS system we permit the developer of an ERD to classify entities or relationships as to their type of *instantiation* and to their type of *access*. *The System* provides these generic objects *instantiation and access*. The properties and semantic constraints associated with these *super* objects are inherited by the descendents in the hierarchy tree. Providing the supertype *access* relieves the tool developer of the task of specifying object operations for storage and retrieval. The supertype *instantiation* provides the database with linking information used during database integration, as well as semantic constraints concerning modification of instances of the objects.

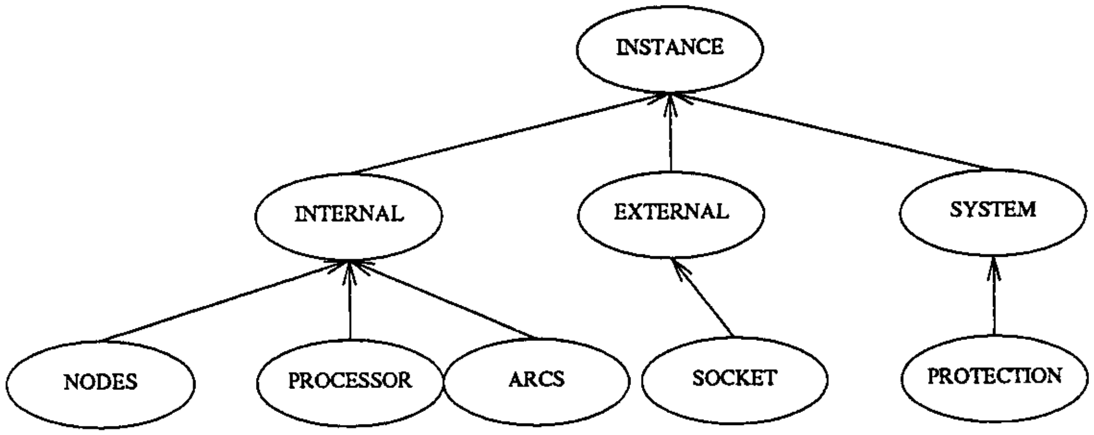


Figure 5.12 Generalization Hierarchy Tree

Figure 5.12 illustrates the subclasses of the predefined object INSTANCE (instantiation object). The leaves of the tree are user defined entities which form a relationship ISA with the node above. For example, NODES ISA INTERNAL object and as such inherits properties and operations defined for INTERNAL. A detailed description of these specific system generic objects is given in Chapter Seven.

The SM MODULE shown in Figure 5.13 illustrates the way in which the designer classifies this entity as to its access type and instantiation type. The subclass *Complex* provides an access mechanism which retrieves the MODULE entity , as well as its SOCKETS and INTERCONN. The subclass *Internal* provides system defined update policies for the entity MODULE. Chapter Seven describes these classes in detail. The remaining entities and relationships found in Figures 5.5-5.10 can be classsified using the same type of notation. In order to eliminate unnecessary repetition in classifying each entity and relationship, the system defines default values.

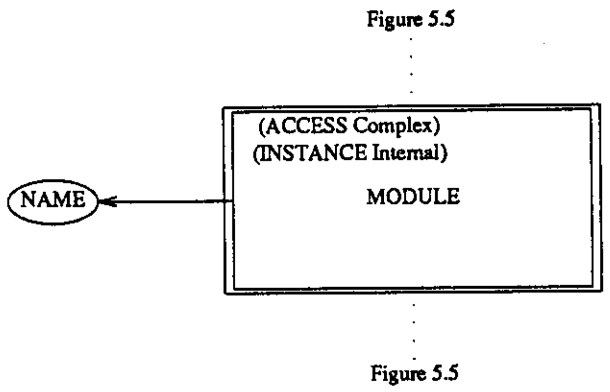


Figure 5.13 Augmented ERD from Figure5.5

Support for the generalization abstraction capability allows the system to provide the generic objects for use by the tool designer. Without this augment the tool designer would not only be forced to repeat numerous sets of attributes and operations for each object defined, but would also be required to provide access operations for each object. This redundancy could only serve to obscure the semantics of the tool description.

5.5.2 Type II Augment - Aggregation

We have found it is necessary to augment the ERD so as to allow for the graphical representation of the abstraction primitive *aggregation*. An aggregation is an abstraction which allows a relationship between entities to be thought of as a higher named object. This relationship is either explicitly defined or implicitly defined. Implicitly defined relationships are those relationships which aggregate (collect or unite into a sum) various entity sets. These implicit relationships are commonly referred to as *Is_Part_Of or Is_Composed_Of.* For example, a *wheel* IS_COMPOSED_OF a rim, lug nuts, and a tire.

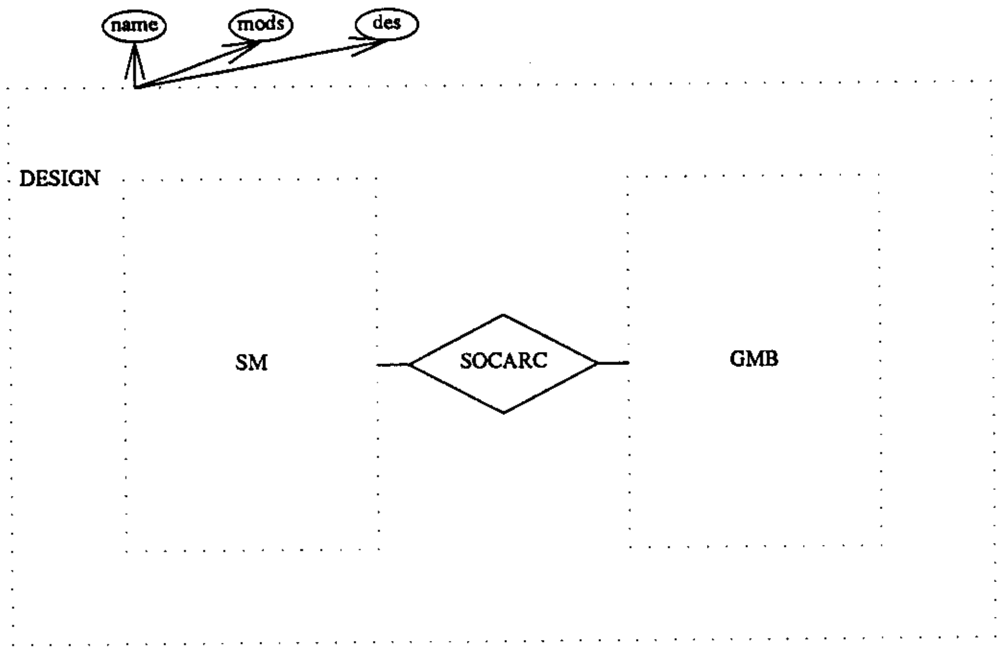


Figure 5.14 Aggregate Object - DESIGN

The aggregate object DESIGN (Figure 5.14) is composed of the objects GMB and SM. The dotted box labeled *DESIGN* is the graphical means of representing such an aggregate in the ERD. We refer to this graphical notation as the *dotted aggregate*. This higher level object may then participate in relationships, and have properties associated with it. In the SARA/IDEAS system a DESIGN might have attributes of a name, designer(s), date of last modification. This type object can then be used to provide catalog information to be associated with a design for use by a Library Browser tool. Figure 5.14 the SM object and the GMB object are related via a relationship defined as SOCARC. This relationship is an entity also found in the object DESIGN.

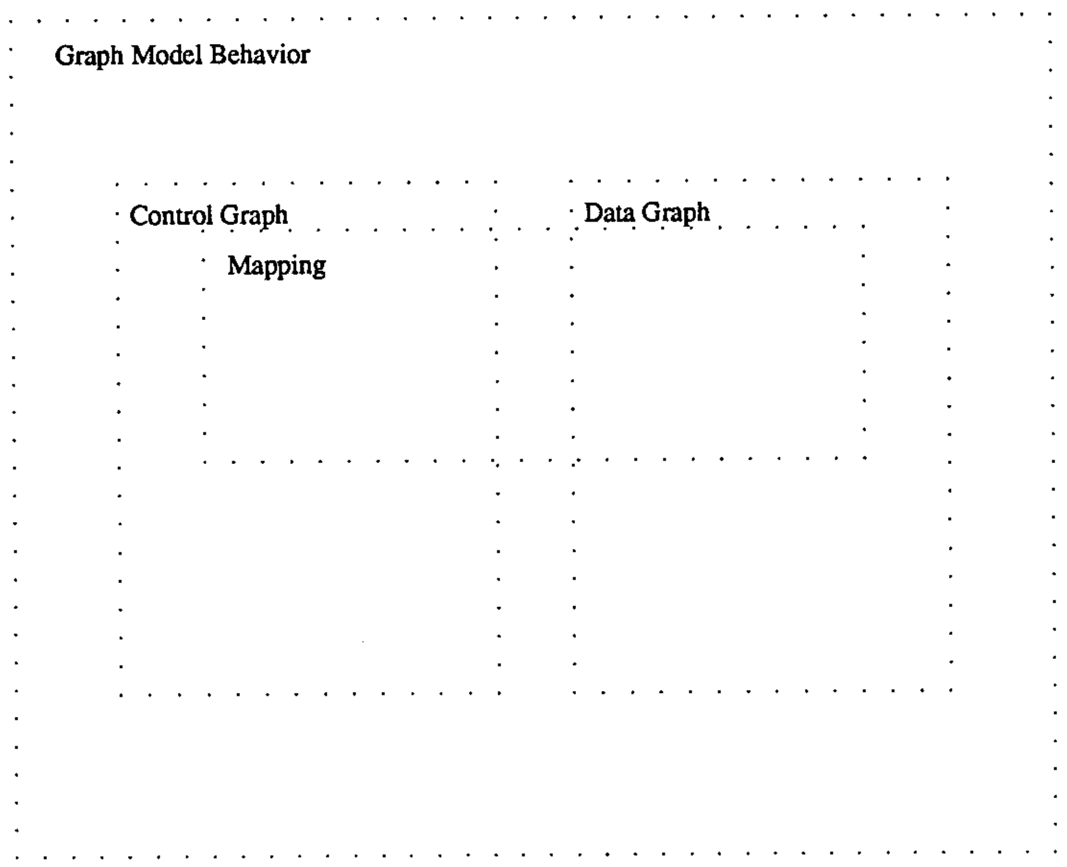


Figure 5.15 Augmented ERD for Figures 5.6-5.8

The refinement process found in design environments (top-down design methodology) necessitates the presence of this capability for representing aggregation. A designer is primarily concerned with the composition of some object. This composition is an object which is created from other predefined objects which might be considered building blocks. These building blocks are often used in a multitude of designs. That is, they can participate in relationships with other designs. The composed object must be able to have properties associated with it and must be able to participate as an entity in other relationships. Supporting the concept of aggregation provides the tool builder with the ability to compose.

Another use of the aggregate primitive is to be able to more clearly represent simple composition. Figure 5.15 is a redrawing of Figure 5.6, 5.7, 5.8 and 5.11. This model clearly represents the aggregate objects GMB, Control Graph, Data Graph, and Control Graph-Data Graph Mapping and their relationships to each other. The dotted aggregate notation provides a clearer representations of these aggregate objects. Figure 5.15 is a considerable improvement over the original ERDs.

Declaration of an aggregate object, using the dotted aggregate notation, associates the object with its constituents via both the system defined relationship types ISCOMPOF and ISPARTOF. In Figure 5.16 we see that the objects; modules, sockets, and interconnections, are part of the SM_design, as well as the relationships has, modsoc, modint, and link. Furthermore, this aggregation primitive is used by the DB_COMPILER to cluster objects and relationships which are part of the same object. The SARA/IDEAS system permits the designer to use either the dotted aggregate notation or to use the system defined relationships in order to model aggregation.

Without support for the aggregation primitive, the tool designer would have to define aggregate relationships for each composition in the design. This definition would need to include access emthods for each such relationship. Having the system

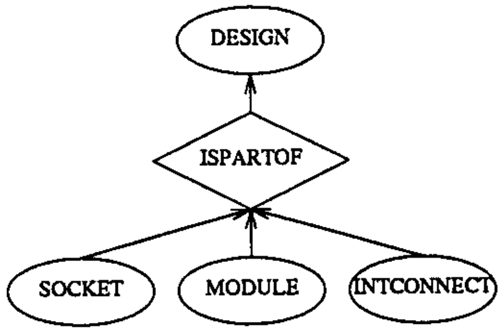


Figure 5.16 Is_Part_Of Relationship

provide the types relieves the tool builder of having to determine database access paths. Again, this redundancy of relationship types would add to the complexity of the tool description and obscure the semantics of the tool.

5.5.3 Type III Augment - Roles

The ERD augment expanding the role concept, provides the tool developer with the capability of specifying properties and/or constraints for subsets of entities. Subsets of entity types are defined in the basic ERD via the roles attached to the mappings from entity sets to relationship sets. We see this role specification being used primarily in describing subsets of entities which take part in recursive relationships. However, subsets of different entity types may participate in non-recursive relationships. In both instances the basic ERM does not elaborate upon these subsets any further.

We have seen a need for not only defining subsets of entity types, but also associating properties and constraints with these subsets. Such a capability needs to be available to the tool developer for use in a design enviornment.

As an example, if a tool developer defined an entity of type NAILS and an entity of type SCREWS, there would be defined for these types such obvious properties as

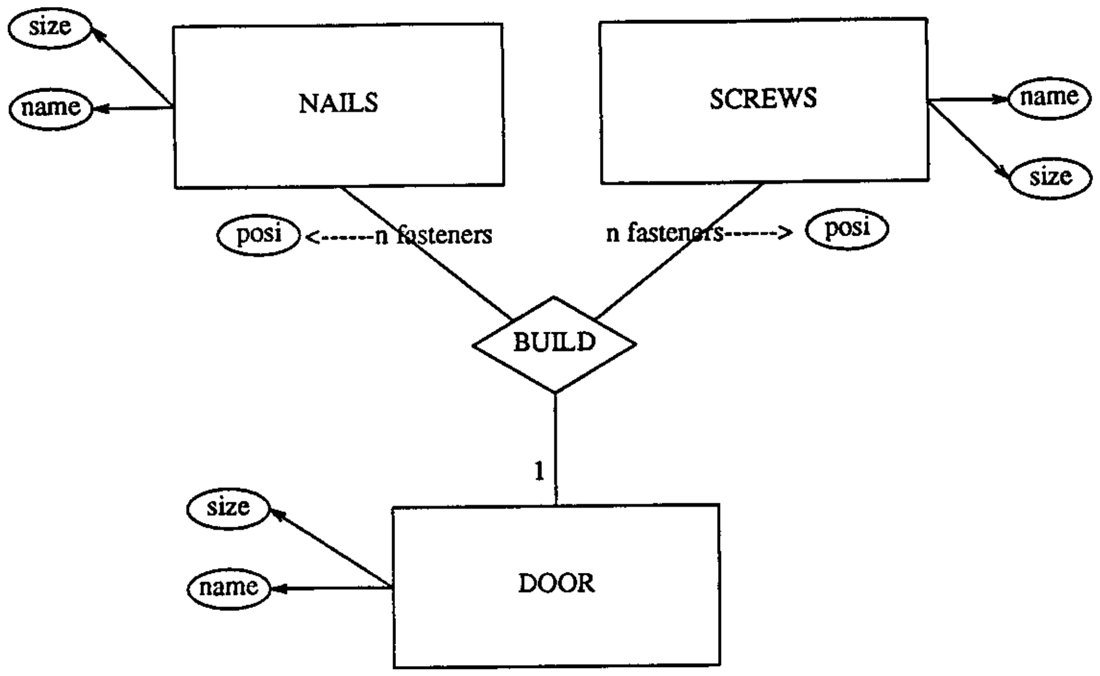


Figure 5.17 Role Augmentation

size, name, etc. Furthermore, if a relationship type BUILD were defined which associated these entities with another entity of type DOOR, then a property of each instance of NAILS and SCREWS would be the position where it is used when building a door. However, this property is not needed for NAILS and SCREWS which do not participate in this relationship but might be associated with DOORS in another relationship, say ISCOMPOF. The entities, nails and screws, do not need a property of posi (position) for purposes of describing solely the composition of the door. The role *fastener,* as illustrated in Figure 5.17, defines the subset of instances of type NAILS and type SCREWS associated with the DOOR entity and also has this property - position attached to it.

Additionally, we see that constraints need to be defined for only a subset of a type of entity. This need has arisen in the design of the SM_Editor tool. Figure 5.18 illustrates the use of such a role constraint specification. Existence constraints can be placed on roles as is indicated for the role of Inside INTERCONN or Outside INTERCONN. Recall that a semantic constraint associated with INTERCONN was that they could not exist unless connected to a SOCKET. This constraint is able to be represented by placing the Existence dependency on the roles. Placement of the existence dependency within the relationship LINK is ambiguous, in that the relationships are 1:1:1, and the original ERM defines dependency only for 1:n relationships. This role constraint specifies that an INTERCONN can not exist unless it participates in a LINK relationship with a SOCKET.

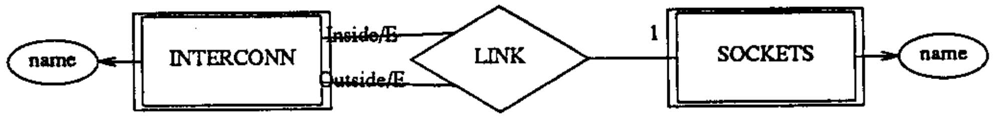


Figure 5.18 Augemented ERD for Figure 5.5

Besides providing a means for defining subsets of entity types, another effect of this type of augmentation is to provide an alternate way of conceptualizing an object. Figure 5.5 showed that a MODULE took on the roles of *child* or *parent. An equivalent way of designing the SM ERD would be to* create an entity called CHILDREN as a subtype of entity MODULE and then attach constraints and properties and relationships. However, the fact that CHILDREN is a subtype of MODULE might be obscured to the designer due to the fact that before participating in any relationship all of the properties of both are identical. The support for roles increases the ways in which equivalent concepts may be modelled. Hence, this augmentation increases the flexibility of use of the model.

5.5.4 Type IV Augment - Properties

The ERD is augmented to expand the representation of value-sets to include the types: text, code and references. These additional types are needed to fully support the design process.

The basic ERD provides the mechanisms to represent attribute value pairs. These pairs are the defined properties of entities, relationships and roles. Values are classified into different value-sets (domains) and the attributes are mapped into these value sets. This mapping allows the designer to associate an attribute with a single value or multiple values. An attribute can also map into the Cartesian product of value-sets to produce a form of a multiattributed attribute. As an example, NAME can map into the value-sets FIRST-NAME and LAST-NAME. Value-sets may be represented sets of the primitive types of integer, character, real, boolean or string. The augmented ERD allows value-sets to also include text, code, and reference.

TEXT

There are many times in a design environment when the need arises to associate text with objects of a model. One example would be the documentation of a design. This documentation is considered a property of that object and needs to be accessed as an information bearing part of that object. The operations associated with this type of value-set would be the reading and writing to a file of this ASCII code. Another example would be the association of descriptive text with a relationship or object. The SARA/IDEAS Browser (Chapter Eight) defines the property of type *text* for each tool object. This description is displayed to the end user, in textual form, whenever the tool is accessed.

CODE

Code is another type supported by the augmented ERD. A distinction is made between the types *text and code* so that meaningful operations can be defined to operate on type *code*. In the present implementation of the SARA/IDEAS system the objects GMB and SM each store an ASCII representation of the T code necessary to regenerate the designed model. When the designer retrieves this T code the tool merely loads the file into the T system. This causes execution of the T code represented in the file. We might consider the contents, since it is in ASCII format, to be an example of a property defined for the objects GMB and SM of type *text*. However, this property is defined as type *code* and as such is loaded for execution and not read as an ASCII text file. The interpretation code associated with the GMB is accessed by the Simulator tool this way also.

REFERENCE

Type *reference* provides the designer with a second mechanism which can be employed to refer to other objects. Recall that the designer can define associations between objects and thereby reference other objects through use of this association. Instances of a defined relationship *refer* will enable the user of the model to ascertain all of the objects which refer to and are referred to. Indirectly then, the instances of the objects which do not take part in the relationship can be determined. Unless constraints have been placed in the relationship, deletion and insertion of objects can occur so that some instances of the relationship are deleted as one of the participants is deleted. However, the use of a property of type *reference* will guarantee that an object will not be deleted if it is referenced to by a property. If some instances of the object having this property do not reference another object then the value of the value-set is null.

As previously stated, the enhancement of the primitive type support provides CADIS with a full set of types found in a design environment. Without this augment, the tool developer would not have the necessary mechanisms to handle design specific primitive types.

5.5.5 Type V Augment - Relationships

The original ERD allows a designer to specify the mapping class (1:1, 1:m, m:1, m:n) for a particular relationship set and also to specify the additional constraint as to whether the relationship type is weak or regular. The mapping class of a relationship specifies the number of allowable appearances of an entity in the relationship set. A weak relationship is a relationship in which the existence of an entity (A) in one entity set depends upon the the existence of a specific entity (B) in the other entity set. Augmentation of the ERD provides for additional classes of relationships to be defined and additional integrity constraints to be placed on these relationships.

Relationship Classes

Relationships in the augmented ERD can be graphically classified with respect to their *completeness*. This classification scheme was found to be necessary in work done by [Sche79] on abstraction capabilities. [Sche79] defined the classes of completeness to be: *partial, total or complete.*

Partial: If the entities (A), of an entity set are not required to be represented in a relationship type which relates members of that entity set to another entity set (B), then the relationship is said to be partial with respect to the first entity (B). This class is the standard regular relationship type defined by Chen and represented by a diamond in the ERD.

Total: A total relationship is a relationship in which *all* instances of an entity type (A), must participate in the relationship. The graphical representation of a total relationship is the inclusion of a dot on the edge of the relationship connected with the total entity set. This class may appear to be similar to a weak relationship type, in that a weak relationship type is indeed a total relationship with respect to one of the entities. However, not all total relationship types are weak. For example, if an instance of an entity is deleted in a total relationship this does not automatically deleted the members of the other entity set which were related to it. First, a search for other appropriate members are made and a reconnection might be made.

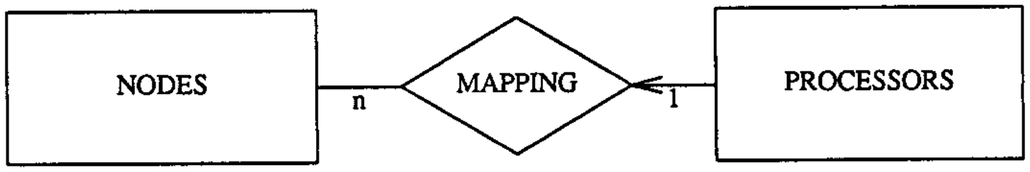


Figure 5.19 Complete Relationship Class

Complete: If each entity in a set is required to be related to all entities in the other set to which it can be related, then the relationship is complete. The graphical notation is an arrow added to the line on the side of the relationship which is required to match all the other entities (Figure 5.19). A complete relationship provides the designer with the capability of expressing the situation in which an entity is dependent on a subset of entities. Figure 5.19 is example of the mapping of a data processor to a set of control nodes. The semantics are that the data processor is deleted only when the set is empty.

Provision for the classification of relationships into one of these three categories, allows the tool developer to specify that system defined data base assertions be activated upon modification of the objects participating in these relationships.

Directed Relationships

Direction of a relationship is implied when a dependent entity is designated. The dependency goes in one direction, from the independent entity to the dependent entity. However, there is a need to express direction other than that which is naturally implied through structural constraints. Graphically, this is simply a matter of adding arrows to the end points of the lines in an ERD. The need for a directed relationship occurs when a designer needs to restrict the symmetry of the semantics of a relationships. An example of symmetry implied in a relationship CONTAIN is that in one direction A CONTAINS B and in the other direction B IS CONTAINED IN A. There are instances in the design environment when this bidirectional relationship quality makes no sense. Modeling of time would be an example of this instance. Direction semantically guarantees that an order to access of the relationship is maintained within the data base.

Ordered Relationships

Often, the need arises in the design environment for an ordered relationship. The entities in an entity set which are mapped via a relationship instance to another entity constitute an ordered set. Such an ordering is illustrated in the mapping of input arcs to control nodes. The attribute LOGIC is represented as boolean expressions of integers such as 1*2+3. This expression refers to the first arc, the second arc and the third arc participating in the relationship INPUT. The relationship can be ordered by key value in ascending or descending order, or by insertion into the relationship; first, second, third, etc. Without this capability, the tool designer would need to create special purpose application programs which would sort and/or keep track of the order of insertion of the obejcts into the relationship. The system provides the necessary operations to maintain ordered relationships.

The DB_KERNEL contains the primitive operations by which the objects and relationships within the augmented ERD are defined and manipulated. The definition operations are used either directly by the tool designer or, as in the case of the SARA/IDEAS system, by a DB_COMPILER. A complete description of the DB_COMPILER is given in the following chapter.

The basic operations which create an augmented ERM are defined in the following partial CADIS syntax. Additionally, there are corresponding operations to destroy entities, relationships or properties as well as to unlink entities from existing relationships. The majority of these operations are similar to those data definition operations which would be found in a standard ERM data base system. However, the augmented ERM defined in the previous sections requires additional syntax in order to define generic and aggregate objects. Therefore, we have included in these operations the specification of *types* of objects in the entity description. This information is used while integrating the ERM into the CADIS system. The N-ary relationships permitted in the ERM must be formed by linking in the other entities after creating a binary relationship and cannot be formed directly, as is the case in some ERM systems. These additional entities must be explicitly linked to an existing relationship. The link operation is necessary so that the tool developer may define entities which then can be linked to existing system or external relationships.

5.5.5.1 CADIS Definition Operations

```
Make_tooldb := <Make_entity list> <Make_relationship list> <Make_property
list> <Link_entity list>

Make_entity list := <MAKE_ENTITY COMMAND> <Make_entity list>  | {}

Make_relationship list := <MAKE_RELATIONSHIP COMMAND>
<Make_relationship list> | {}

Make_property list := <MAKE_PROPERTY COMMAND> <Make_property
list> | {}

Link_entity list := <LINK_ENTITY COMMAND> <LINK_entity list> | {}

MAKE_ENTITY COMMAND:= 'MAKE_ENTITY' <entity_type>
<spec_types_list> <key_list> <econstraint_list>

MAKE_RELATIONSHIP COMMAND:= 'MAKE_RELATIONSHIP' <rel-
type> <relspec_types> <entName list> <role_list> <mapping_list> <key_list>
<rconstraint_list>

MAKE_PROPERTY COMMAND:= 'MAKE_PROPERTY' <host_id>
<attrName> <attr_type> <constraint_list>

LINK_ENTITY COMMAND:= 'LINK_ENTITY' <entName> <relName>
<role> <mapping> <key> <econstraint_list>

spec_types_list:= <spec_types> <spec_types_list> | {}

spec_types:= 'Subtypes' <subtypes_list>| 'Aggregate' <aggregate_list> | {}

rel_spec_types:= <rel_access> <completeness_class>
```

subtypes_list:= <subtype> <subtypes_list>| {}

aggregate_list:= <aggregatetype> < aggregate_list> | {}

subtype:= <instantiation> <ent_access> <entity_type>

instantiation:= 'External'|Internal' | 'System'

ent_access:= 'SIME'|'COME'

rel_access:= 'SIMR'|'COMR'|{}

completeness_class := 'P' | 'T' | 'C' |{}

key_list:= key <key_list> | {}

constraint_list:= <constraint body> <constraint_list> | {}

role_list:= <role> <role list> | {}

mapping_list:= <mapping> <mapping list> | {}

host_id:= <entName> <relName> <role,entName>

attr_type:= <single_attr> | <multival_attr> | <multiattr_attr>

role:= <roleid> <constraint_list>

mapping:= <mapping_type> <mapping_group>

entName list:= <entName> <entName list> | {}

entity_type:= <entName>

entName:= *string*

rel_type:= <relName>

relName:= *string*

attrName:= *string*

key:= <attrName>

single_attr:= integer | real | text | code | reference | boolean | string | char

multival_attr := 'M' <single_attr>

multiattr_attr := <single_attr> <multiattr_attr> | {}

attrNamelist := <attrName> <attrNamelist> | {}

An example of the transcript which was used to define the database schema for the SM tool follows. The operations are simply functions which create internal representations of objects, relationships and attributes. The subtypes for the generic objects, instantiation and access, are specified within the MAKE_ENTITY command. This example show only the use of the subtype Internal. However, a GMB transcript would generate the other generic instantiation type, External. The implementation of the language is in the form of function calls with a list of parameters needed to complete the operation.

```
MAKE_ENTITY(UNIVERSE,
 (Subtypes Internal, COME),
  (Aggregate MODULE,SOCKETS,INTERCONN,MODSOC,MODIC,LINK,HAS),
   (Name))
MAKE_ENTITY(MODULE,(Subtypes Internal,COME),(Name));
MAKE_ENTITY(SOCKETS,(Subtypes Internal,SIME),(Name));
MAKE_ENTITY(INTERCONN,(Subtypes Internal,SIME),(Name));
MAKE_PROPERTY(MODULE,Name,string,());
MAKE_PROPERTY(SOCKETS,Name,string,());
MAKE_PROPERTY(INTERCONN,Name,string,());
MAKE_RELATIONSHIP((MODSOC,(MODULE,1))(SOCKETS,n,EX))());
MAKE_RELATIONSHIP((MODINT,(MODULE,1))(INTERCONN,n,EX))());
MAKE_RELATIONSHIP(HAS,((MODULE,1,parent)(MODULE,n,children(E ID)));
MAKE_RELATIONSHIP(LINK((SOCKETS,1)(INTERCONN,1,inside(E))));
LINK_ENTITY ( LINK,(INTERCONN,1,Outside(E)));
```

SM Editor Data Definition Transcript

5.5.5.2 Data Manipulation Operations

Operations which manipulate the data model are also found in the DB_KERNEL. These operations include insertion, retrieval, and modification of the database objects, relationships and their respective attributes. They too are in the form of a set of functions which are later translated into lower level access functions provided by the underlying database system. For purposes of this work the underlying model, supporting the augmented ERM, is an extension of the relational model, Relational Model Tasmania. [Codd79] Since this model is used as the basic model of the system, a detailed description will be given in chapter Six.

The following sample design would be generated by an SM editor consistent with the augmented ERD requirements. We now illustrate the structure and use of the various classes of data manipulation operations. These operations can be used by the tool builder to provide query facilities for the designer through a tool subdialogue. The basic philosophy underlying the semantics of these operations is that the fundamental units manipulated by the designer are objects and relationships. Therefore, selection, retrieval, and insertion are defined in terms of these objects and/or relationships.

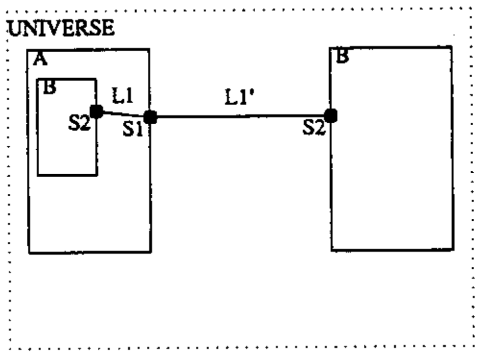


Figure 5.20 Sample SM Design

Figure 5.20 indicates that the user of the SM Editor tool has created an aggregate object, UNIVERSE. Recall that an aggregate object is associated, through the relationship ISCOMPOF, with other objects. These objects are MODULE A and B and INTERCONN L1. MODULE A assumes the role of the *parent* of the child MODULE B via the relationship HAS. MODULE A also is associated through the relationships MOD/INT and MOD/SOC with INTERCONN L1' and SOCKETS S1 .

Likewise, there is another MODULE B, the sibling of A, which has a SOCKETS S2. MODULE B, the child of A, also has a SOCKETS S2.

SELECTION

The first class of manipulation operations defined for the model are the select operations. The semantics of a FIND is to select and return a set of unique identifiers for which some qualification clause is true. As is the case in many database systems, a unique identifier must be found before retrieval or modification of objects or relationships can occur.

The partial syntax of the FIND operation is

```
Select_op := <select_entity> | <select_relation> | <select entrole>
select_entity := FIND <entity> WHERE <entityqual clause>
select_entrole := FIND <relation><role> WHERE <entityqual clause>
select_relations := FIND <relation> WHERE <relatqual clause>
equalexpr := <key|attrName|relName|<relName><role>>
             <boolexpr><valueexpr>
rqualexpr := <key|attrName|entName|role>
             <boolexpr><valueexpr>
entityqual clause:= <Select_op> | <equalexpr>
relatqual clause:= <Select_op> | <rqualexpr>
entity := <entName>
role :=<roleName>
relation := <relName>
boolexpr:= '=','<>'...
valueexpr= <value>|entName|role|relName|< >
```

The result returned by the operation

FIND UNIVERSE WHERE Name = design1

is the set of 1 element whose value is the system identifier of the UNIVERSE. For use in later examples the system identifier returned is referred to as @U1. To access the child of A - MODULE B, the following operation would be formed.

FIND MODULE WHERE HAS parent WHERE Name = A

Note that the query provides for access to the role set parent. In response to another operation:

FIND MODULE WHERE Name = B

the unique identifiers would be that of a set of two elements indicating both MODULES whose NAME is B.

As can be seen from the syntax, the operations provide for return of the identifiers of relationships.

FIND ISA WHERE INTERNAL = MODULE

will return the set of identifiers of all modules classified as internal objects. In the example of the SM this would be all of the modules in the diagram. This example query illustrates the ability to access subclasses of objects.

RETRIEVAL

The second class of operations are the retrieval operations. Retrieval is initiated only after the establishment of a unique identifier. The syntax is relatively simple, yet execution of these operations can be complex. The semantics of these operations are detailed in Chapter Seven.

```
load_op :=RETRIEVE object-id
        | <attrName> OF <object-id>| <relation-id>
```

Execution of the operation RETRIEVE @U1 will result in the entire model being brought into the system. The database retrieves the entity UNIVERSE with the property Name whose value is A. Since the entity retrieved contains the entity MODULE with the property Name whose value is A and the entity MODULE with the property Name whose value is B, these MODULE, their SOCKETS, INTERCONN and *children* will be retrieved.

INSERTION/DELETION

The third class of operations are those that allow instances of the ERD objects to be placed in the database or removed from the database.

```
INSERT <object-list>| <relation-list>
  DELETE <object-list>|<relations-list>
```

The construction of the operations to store an instance of an actual model could be quite tedious. However, identification of MODULE as the unit of access eliminates the need to perform individual insertions and attachments. Following is a sample transcript of an insert of the model in figure 5.20 into the SARA/IDEAS database.

```
INSERT MODULE( Name=A;
               Parent;
               SOCKETS;
               INTERCONN( Name=L1);
               Children( (Name=A;
                     SOCKETS( Name=S1);
                     INTERCONN;
                     Children;)
            (Name=B;
                     SOCKETS(Name=S2);
                     INTERCONN;
                     Children;))
               Inside;
               Outside L1,S1:L1,S2)
```

Since the DB_KERNEL knows that this is a complex object and has classified the relationships, the individual inserts into MODULE of the children and the linking of the children to their parent A are all accomplished automatically.

MODIFICATION

The final class of operations are the modification operations. These operations simply allow the tool builder to modify the values of the properties attached to relationships and objects.

```
CHANGE property = value OF object-id/relationship-id
        TO newvalue
ATTACH property=value TO object-id/relationship-id
```

The CHANGE operation takes an existing attribute value pair and changes it, whereas the ATTACH operation places a new attribute-value pair in the database. Together the INSERT and ATTACH operations are the means of creating instances of those SM objects, relationships and properties.

We have presented a limited selection of operations which can be used to manipulate the database. Chapter Seven details the various access operations, whereas in Chapter Eight we find examples of some of the types of query operations performed on the database.

5.5.5.3 Summary

This chapter has defined an augmented ERD to be used as the conceptual model for a tool builder using the Worley methodology. We have identified those augments needed in order to support the development of tools which will be used in a design environment. These augments have enabled a tool developer to clearly specify the semantics of the tool, have allowed the tool designer to specify complex data base access requirements without being a database expert, and have provided alternate ways for developing the tool's specification model.

We showed that inclusion of specific generic objects provides an extensible easily integratable database. Integration of a tool's data base scheme is achieved via the *external* object mechanism. This generic object allows a tool developer to use objects created by other tools. Use of the generic object *access* provides the tool developer with complex data base access routines. Aggregate objects can be used to represent composition and decomposition of designs, as well as allowing an object to be abstracted to a level more appropriate to use.

We expanded upon the concept of roles so as to provide the tool developer with the capability of focusing on subsets of objects and their attributes. This ERD enhancement affected the storage and access design of the underlying data base. The support for additional primitive types enable the ERD models to be more closely attuned to the design environment, where such phenomena as documentation, executable code, and references to such objects as component lists are common.

Dynamic constraint specification capability is introduced through the inclusion of class representation, directionality and ordering of relationships.

The concluding sections gave a brief introdution to the data definition and data manipulation language of the CADIS system. Operations on the CADIS data base are object oriented and are capable of handling the manipulation of classes of objects. Although these languages are similar to other Entity-Relationship languages in many respects, the introduction of query capability on generic and aggregate objects was necessary to support the augmented ERM. Another major difference in the manipulation language is the ability to access subsets of objects based on the roles they assume in the various relationships.

CHAPTER 6

The Implementation Level

The implementation level of CADIS consists of the underlying data structures and the functions needed to access these structures. The development of this implementation level requires the generation of specifications which later become input to the implementation's technical design. This specification is required to contain sufficient technical design information, present the information in a structured form and be readily converted to the implementation model. The augmented ERM defined in Chapter Five is the major source of such a specification. It includes the data items in the user's environment together with the relationships between them, specifications as to the size and type of data items, and partial specification as to the access requirements. This chapter shows how the augmented ERM is converted to the implementation data model, an extended relational model.

The chapter begins by discussing the issues which arise when mapping from one model into another. The reasons for converting the augmented ERM into the extended relational model, RM/T [Codd79] are then listed. A brief definition of the RM/T is given prior to a presentation of the translation rules needed to map the augmented ERM to the RM/T. These rules form the nucleus of the classifier algorithm, the main procedure found in the DB_COMPILER. Excerpts of the code used in the SARA/IDEAS system are given at the end of the chapter.

6.1 Mapping Schemas

We have seen that the traditional data models; relational, hierarchical, and network, have been proposed independently of one another. More recently, numerous semantic data models have also been proposed. However different they seem, there are some basic underlying concepts which relate them. This relationship makes it possible to determine equivalence relationships and equivalence preserving mappings between data models. Algorithms [Bork78, Lien80] have been developed for mapping from a relational model to a network model or from a relational model to a hierarchical model, and vice versa. The basic ERM has also been mapped to the relational, hierarchical and network models. [Jajo83, Lusk80, Dump81]

Being able to map from one data model to another is of practical importance, since many large organizations have different data bases and DBMSs which have to coexist. The Worley tool building methodology has specified that tools should be able to be built independent of the underlying existing DBMS. As a consequence, this independence permits CAD system developers to experiment with the various DBMS systems which are available. These systems may or may not support different data models. Thus, the exercise of mapping the augmented ERM into an underlying data model is important so as to validate the claim that the augmented ERM supports the Worley tool building methodology. Since both the augmented ERM and other data models, in particular the relational model, are defined in terms of sets and relations, it would appear to be relatively easy to specify structure mappings between these data models. The major difficulties in the mapping process are introduced as we consider the constraints.

The underlying model we have chosen to map into is relational. Mapping from an augmented ERM into a relational schema structures the less formal ERM into a more formal data representation model. The mathematical formality of the relational model allows a data base expert to analyze the resulting storage structures and modify, if necessary, in order to achieve greater performance and also to eliminate anomalies which may have been present in the original model. This process is known as fine tuning the database.

Mapping from a conceptual model (the augmented ERM) to an underlying schema involves three major tasks. First, a mapping from the conceptual model's structure to the underlying model's structure must be defined. Next, the ERM's operations must be mapped to the underlying model's operations. Finally, the constraints expressed in the conceptual model must be mapped to equivalent constraint representation in the underlying model. This chapter defines the structural mappings and included structural constraints. The mapping of operations and operational constraints are considered in Chapter Seven.

6.2 Implementation Model

Early in the design of the SARA/IDEAS system, it was decided to experiment with the use of Troll as the underlying data base system. The reasons for this choice are listed in Chapter Four. However, we note here again that the primary attraction of Troll is that it is a small and simple data base system based on the relational model. The basic relational model is known to lack the capability of representing the additional abstraction primitives and the complex inter-relational constraints such as those added to the basic ERM. Therefore, it is necessary to modify Troll in order to support these augments. [Codd79] defines an extension to the relational model in order to capture more semantics. A review of this extended relational model shows the constructs to be similar to those found in the model used by the SARA/IDEAS tool building system. We have indicated such provisions of the RM/T in section 4.3. The modifications to the Troll code to support this extended relational become obvious as we look into the description of RM/T.

6.2.1 RM/T Description

The following description is a condensed version of the original description found in. [Codd79] The fundamental RM/T constructs are preceded by the notation [C-#] (where # is an integer), so that we can easily reference these definitions.

The basic assumption underlying RM/T is that the real world can be modeled in terms of entities. It is also assumed that entities can be categorized into entity types, so that a simple regular structure can be imposed on the database. Relationships of many types can exist among entities. Entities of the same type have certain properties in common and hence, the RM/T model factors out that commonality and achieves some economies of representation.

In order to illustrate some of the definitions found in Codd's paper, we refer to the following hypothetical model of an EMPLOYEE - PROJECT example data base system.

A PERSON is an entity. Some properties of this entity are that of a name and birthdate. The property birthdate consists of a month, day and year.

Instances of this entity PERSON are:

Entity 1: Jones	March 22, 1934
Entity 2: Smith	April 1,1942
Entity 3: Johl	June 21, 1965

An EMPLOYEE is an entity. EMPLOYEE is a PERSON. Thus some properties of this entity are name, birthdate and also social security number, department assigned to and dependents.

Instances of this entity EMPLOYEE are:
Entity 1: Smith 123-234-455 Dept A
Tom, Mary, Joe
Entity 2: Johl 987-567-345 Dept C
Dick, Joe

A DEPENDENT is also an entity. Properties of this entity are age and name.

Instances of this entity DEPENDENTS are:
Entity 1: Tom 13
Entity 2: Mary 15
Entity 3: Joe 4
Entity 4: Dick 3
Entity 5: Joe 2

PROJECTS is an entity. Some properties of this entity are that of a name, a start date, a finish date, and the department in which this entity takes place.

Instances of this entity PROJECTS are:
Entity 1: StarWars 1/23/84 1/1/99 Dept A
Entity 2: Gemini 6/14/85 1/1/92 Dept B

DEPARTMENT is an entity. Some properties of this entity are a name, building number, number of offices it contains.

Instances of this entity DEPARTMENT are:
Administration 7 86
Engineering 2 43

EMPLOYEE entities are associated with PROJECTS entities to represent the projects an employee might be assigned to.

Instances of these associations are:
EMPLOYEE Entity 1 is assigned to
PROJECTS Entity 1
EMPLOYEE Entity 2 is assigned to
PROJECTS Entity 2
EMPLOYEE Entity 1 is assigned to
PROJECTS Entity 2

Example 6.1

Continuing with the definition of the RM/T, the RM/T supports the concept of the use of surrogates within the database. Surrogates are system generated entity identifiers, which address the problems of user key change, duplicate user keys to identify the same entity, and lack of a user key. Within the database all entity identification and entity referencing is performed via surrogates. The use of an appended '@' throughout the following chapters indicates a surrogate.

Formal constructs of the RM/T are:

- [C-1]: E-domains, domain of all possible surrogate values;
- [C-2]: E-attributes, any attribute defined on the E-domain;
- [C-3]: E-relations, a relation exists for each entity type.

The primary purpose of the E-relation is to record the existence of the entities in question. E-relations accept insertions and deletions but not updates.

[C-4]: A property is an immediate single-valued piece of information that describes an entity. The property types for a given entity type are represented by a set of P-relations. Every E-relation has a corresponding set of P-relations. If a given entity is classified as belonging to different types then it will have properties corresponding to each type. The RM/T convention that P-relations for a given entity type do not have any attribute names in common, allows supertype properties to be automatically inherited by its subtypes.

Figure 6.1 illustrates the tabular representation of the E-relations for the entities EMPLOYEE, DEPENDENT, DEPARTMENT and PROJECT. The tuples in these relations consist soley of an E-attribute whose value is found in the E-domain, EMPLOYEE

E@
E1@
E2@

E-RELATION

PROJECTS

P@
P1@
P2@

E-RELATION

DEPARTMENT

DT@
DT1@
DT2@

E-RELATION

DEPENDENTNAME

DP@	name
DP1@	Tom
DP2@	Mary
DP3@	Joe
DP4@	Dick
DP5@	Joe

P-RELATION

DEPENDENT

DP@
DP1@
DP2@
DP3@
DP4@
DP5@

E-RELATION

DEPENDENTAGE

DP@	age
DP1@	13
DP2@	15
DP3@	4
DP4@	3
DP5@	2

P-RELATION

Figure 6.1 Selected E-Relations and P_Relations from Example 6.1

the domain of surrogate id's. Only the P-relations for the E-relation DEPENDENT are illustrated in this figure. The properties of DEPENDENT are *name* and *age*. The DEPENDENTNAME P-relation is shown to consist of five 2-tuples; surrogate id and name value; for each of the dependents defined in Example 6.1. Similarly, the P-relation DEPENDENTAGE contains five 2-tuples, each one having an attribute value which identifies the ages of its respective DEPENDENT entity.

Additionally, entities are classified into the following four groups :

- [C-5]: Characteristic: A characteristic entity is an entity whose sole function is to qualify or describe some other entity. The characteristic entity is subordinate to and existence-dependent on a superior entity which it qualifies. If an entity has multivalued properties then those properties are classified as characteristic entities. The supporting integrity rule is that a characteristic entity cannot exist unless the entity it most immediately describes also exists.

- [C-6]: Associative: An associative entity is an entity whose function is to represent a many-to-many relationship between two or more entities The associative entity has participants which are entities that are independent and adhere to the integrity rule that a given instance of an associative entity type can exist in the data base only if for that instance each E-attribute in its P-relations either has the value E-null or identifies an existing entity of the appropriate type.
- [C-7]: Kernel: A kernel entity is an entity that is neither of the above.
- [C-8]: Designative: A designative entity is an entity which is used to complete the description of another entity but is also an entity in its own right. The designative integrity rule for this type is that a given instance of type designative can exist only if for that instance each entity it designates is either null or is an existing entity of the appropriate type.

RM/T maintains catalogs, as relations, which describe the relations, attributes, and domains present in the database. These are referred to as the RELATION relation, ATTRIBUTE relation, and DOMAIN relation, respectively. The system tables shown in figure 6.2 are the RELATION catalog and ATTRIBUTE catalog. The tuples found in the RELATION relation consist of a relation name and relation type. The attribute Relname contains the name of every relation in the database and the attribute Reltype designates which class of entities the relation represent. For example, RELATION

Relname	Reltype
PERSON	K
EMPLOYEE	D,K
DEPENDENT	C
PROJECTS	K,D
DEPARTMENT	K
BIRTHDATE	C

ATTRIBUTE

Relname	Attr	Domain	Pkey	Ukey	Null
PERSN	name	string	false	true	false
EMPLOY	name	string	false	true	false
DEPENT	name	string	false	true	false
PROJ	name	string	false	true	false

Figure 6.2 Partial System Tables for Example 6.1

the EMPLOYEE entity is classified as being a Designative (D), as well as Kernel (K) entity. The DEPENDENT entity is classified as a Characteristic entity(C). The ATTRIBUTE relation contains tuples for every attribute present in the database. Although there are no entries in the table, the attributes are: the name of the relation in which the attribute is found (Relname); the name of the attribute (Attr), the domain of the attribute values (Domain); and the three boolean attributes indicating whether the attribute is used as a primary key (Pkey), is used as part of a user key (Ukey), and if nullvalues are allowed (Null).

[C-9]: Entities may also have one or more subtypes defined for them. This subtype classification constitutes a generalization hierarchy which provides for inheritance of properties by subtypes. Entities may also be divided into several distinct categories of subtypes. RM/T supports a subtype integrity rule which states that a surrogate of an entity or type *e* must also belong to the E-relation for each entity-type of which *e* is a subtype.

[C-10]: Aggregation is also supported in the RM/T. Aggregation is broken down into three types:

1. [C-11]: Aggregation of simple properties which yields an entity type - all the properties which comprise an entity type.

2. [C-12]: Aggregation of characteristic entities yields an entity type - all of the characteristic entities of a given entity type, along with the respective properties.

3. [C-13]: Aggregation of any combination of kernel and associative type yields an associative type or nonentity association type - this aggregation promotes a relationship relation to that of a relationship entity so that it may participate in other relationships.

Since RM/T entity types are not related in ways based only on values of attributes, graph relations are maintained by RM/T which specify the way in which the entity types are related to each other. These graph relations provide the ability to specify relationship types, generalization hierarchies and other semantics. The following is a brief description of the graph relations found in RM/T.

- Property graph relation (PG-relation or PropGraph relation): indicates which P-relations belong to which E-relations.

- Association graph relation (AG-relation): indicates which entities participate in which relationship types.

- Characteristic graph relation (CG-relation or CharGraph relation): indicates which entities are characteristic and which entities they characterize.

- Designation graph relation (DG-relation): indicates the designated and designative entity-types together with the name of the designative property.
- Generalization graph relation (SubGraph relation): indicates an entity-type, its generalization and the category to which it belong.
- Aggregation: uses the P-, PG-, CG-, and AG-relations and does not maintain an individual graph relation. Support for the first type of aggregation is found by the P-relations together with the PG-relations; the second type by the characteristic relations with the CG-relations; the third type by the kernel relations, associative relations and the AG-relations.

There are additional graph relations which provide for more semantics to be represented such as a cover aggregation type and entities of type event. As it turns out representation of a cover aggregation type does not necessitate the need of a special graph relation in our database. Rule 7 in the following section requires that the ERM aggregate entity, analagous to Codd's cover aggregation, need only be mapped to a characteristic entity. Entities of type event are not considered in this research primarily because the need for such entities did not arise.

Figure 6.3 shows a partial listing of three of the graph relations generated for Example 6.1. The PropGraph (PG-relation) contains a tuple for each property of the entity EMPLOYEE, DEPENDENT, and PROJECT. For example, the EMPLOYEE entity has a tuple indicating the property, Name (EMPLOYEENAME), and a tuple for the property, Social-Security-Number (EMPLOYEESS). Birthdate is not listed in the PropGraph relation since, according to Codd, it is considered a characteristic entity (Section 6.4) of EMPLOYEE. Therefore, it appears in the CharGraph (CG-

PropGraph

PRelNAME	ERelNAME
EMPLOYEENAME	EMPLOYEE
EMPLOYEESS	EMPLOYEE
DEPENDENTNAME	DEPENDENT
DEPENDENTAGE	DEPENDENT
PROJECTNAME	PROJECT
PROJECTSTART	PROJECT

CharGraph

ChERelNAME	SupERelNAME	RelType
DEPENDENT	EMPLOYEE	HAS
BIRTHDATE	EMPLOYEE	-----

SubGraph

SubERelNAME	SupERelNAME	Status
EMPLOYEE	PERSON	ISA

Figure 6.3 RM/T Graph Relations

relation) relation as a characteristic entity of EMPLOYEE, as well as the entity DEPENDENT. The third field in the CharGraph relation indicates the type of characteristic relationship being represented. The value of "HAS" in the tuple indicates that the relationship is an entity-entity relationship labeled HAS, whereas, the value of "----", indicates the type of relationship is that of a multivalued property relationship. The SubGraph relation represents all of the class-subclass categories defined for the database. The Status field indicates the type of category. For purposes of representing the standard generalization category we have used the value "ISA".

The system relations described in this section constitute the *meta* data used to manipulate an RM/T database. Therefore, any modifications to the underlying relational database (Troll) must include the capability to process these system relations.

6.3 Translation Rules

The augmented ERD, generated from a tool's concept definition specifications, is used to define the underlying database schema. This schema consists of the appropriate RM/T relations described in the preceding section. The schema is defined either by a database designer using the data definition operations provided by CADIS (Section 5.7) or by some form of a data base compiler which automatically translates the ERD into the underlying RM/T data model.

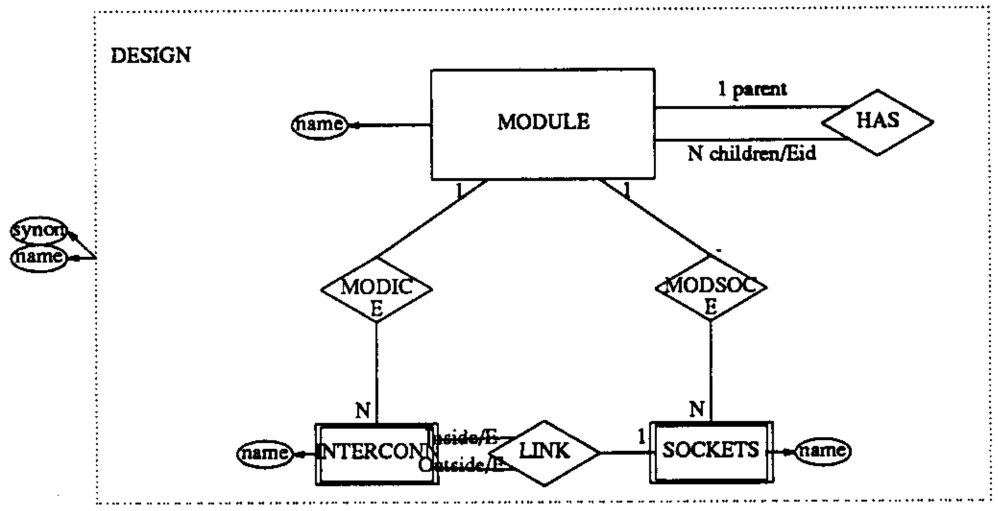


Figure 6.4 SM Editor ERD

Chen [Chen77] originally defined translations of the basic ERM into a traditional relational model, a hierarchical model and a network model. The translation into the relational data model primarily involved mapping entities into entity relations and relationships into relation relations. Others (Section 6.1) have defined mappings of various extended ERM's into relational, hierarchical and network data models. We now define the mapping of the augmented ERD's (Chapter Five) objects and relationships into the extended relational model, RM/T.

The ERD of the SM Editor tool is used to pictorially illustrate the mapping rules. The translation begins with the identification of RM/T entities.

Rule 1

> Each entity in the ERD which has no graphical constraints, such as existence-dependencies, is mapped into a unary E-relation using the entity-type as the name of this E-relation.

This mapping rule is derived from the natural correspondence between Codd's definition of E-relations [C-1] and an ERD entity. The CADIS DB_COMPILER procedure for identification of the E-relation's single attribute is: the composition of the first n unique letters of the relation's name followed by '@' constitutes the E-attribute of all such E-relations. Figure 6.5 illustrates the entity MODULE being mapped into an E-Relation MODULE. The E-Relation's attribute is M@.

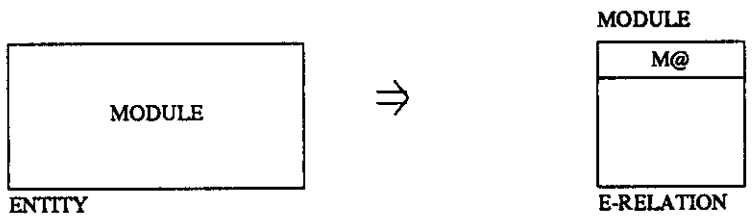


Figure 6.5 Entity Identification

Rule 2

Each *single* valued property of an entity is mapped into a P-relation for that entity's E-relation.

This rule is also derived from the natural correspondence between Codd's definition of entity properties [C-4] and an ERD property. Since the augments for the properties of type *code* and *text* merely involve identifiers for the files in which these types reside, they are viewed as single valued properties. The CADIS naming procedure for P-relations is to concatenate the entity-type with the attribute name. Figure 6.6 illustrates the creation of a P-relation MODULE NAME, representing the attribute Name which is a property of MODULE. This binary relation consists of the attributes M@ and Name.

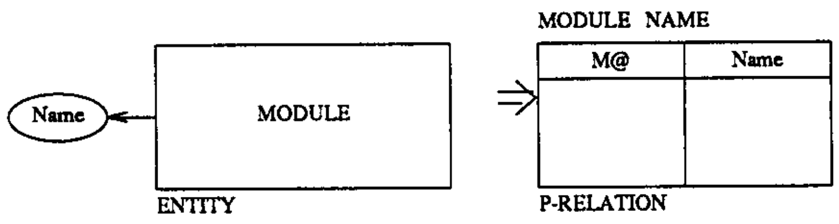


Figure 6.6 Property Identification

Rule 3

Each entity which has a property of type reference is mapped into a Designative entity.

Entities can be Designative as well as being classified as either Kernel, Characteristic or Associative. This mapping rule also results from the natural correspondence between [C-8] and the augmented ERD definition of type reference. Application of rule 3 does not create any new relations. However, the system relations are modified so as to indicate that an entity is of type Designative and to also indicate the Designee.

Rule 4

> Each multivalued property of an entity is mapped into a unary E-relation, which is classified as a characteristic entity.

A single P-relation is required to be created for this new E-relation. The P-relation's name is the name of its E-relation concatenated with 'P'. This P-relation identifies each instance of the property and indicates to which entity it belongs.
This translation rule results from application of Codd's definition of characteristic entity [C-5] to a multivalued ERD property. The CADIS naming convention for this E-relation is to concatenate the first n unique characters of the characterized entity's E-relation with the attribute name. Figure 6.7 shows the multivalued property SYNONYM , of the entity Design mapping into the E-Relation SYNONYM and its respective P-Relation SYNONYMD. The binary P-relations relates an instance of a SYNONYM entity to the instance of its DESIGN entity.

Rule 5

> Each entity set which is existence-dependent on an entity set and participates in a binary 1:n relationship with that entity set is mapped into an E-relation of type Characteristic.

This rule is also derived from the natural correspondence between [C-5] and Chen's

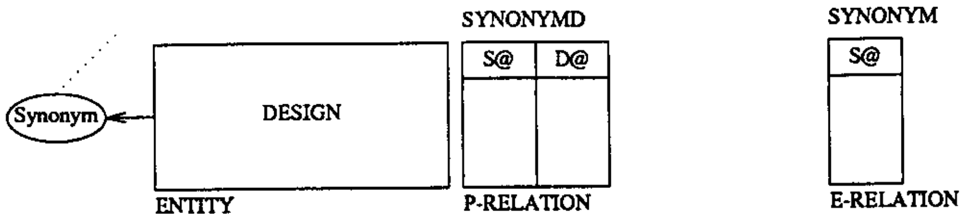


Figure 6.7 Multivalued Identification

definition of a dependent entity. As an example, SOCKETS and INTERCONNECTS are defined to be existence-dependent upon entity set MODULE and therefore map into characteristic entities of MODULE. The relations defined are similar to those described in Figure 6.7 for the characteristic entity SYNONYM. The schema would then consist of two relations SOCKETS and INTERCONNECTS, along with their respective P-Relations, SOCKETSM and INTERCONNECTSM. Using this rule, would eliminate any instances of the relationships MODIC and MODSOC. Therefore, the algorithm developed in 6.4 retains the defined relationships by placing them in the Characteristic Graph Relation as values of the attribute RelType - see Figure 6.3.

Rule 6

> Each m:n unrestricted relationship type is mapped into a unary E-relation which is classified as associative. A P-relation is required to be created which indicates each participant in the association.

This is another direct mapping from the basic ERM based on Codd's definition of an associative entity [C-8]. Figure 6.8 shows the relationship type USE as an E-relation along with its P-relation USEP. P-relation USEP has as entries those entities which are paired in the relationship USE. There is also an entry made in the AG-relation indicating the name of each entity along with the name of the associative entity name.

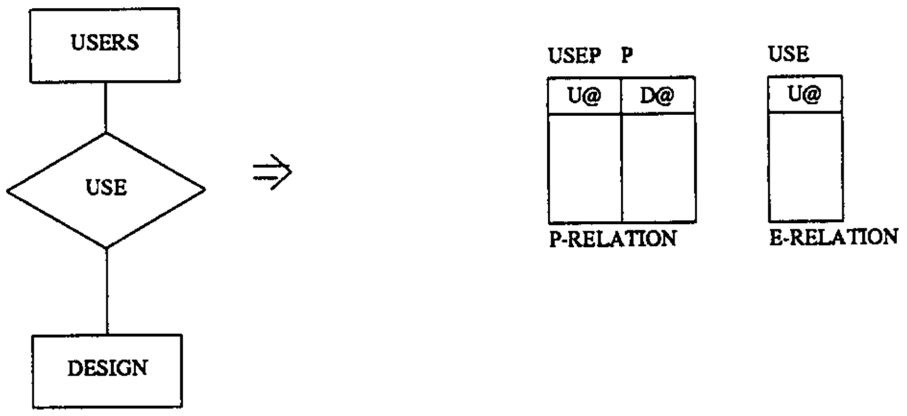


Figure 6.8 M:N Relationships

Rule 7

Each aggregation is mapped into an E-relation. All entities which comprise this entity are then marked as characteristic of this entity.

[C-12] provides the correspondence between the augmented ERM's definition of aggregation and RM/T support for aggregation. The dotted aggregate notation encompasses all of the entities which are part of the aggregate object. Relationships within these boundaries are also included. Use of the explicit relationship ISCOMPOF will result in an equivalent translation. Recall from Chapter Five that UNIVERSE and MODULE are related via the relationship type ISCOMPOF (the aggregating relationship). Application of this rule would cause the MODULE entity to be changed from a kernel entity to a characteristic class. A new P-relation is also created for MODULE which contains the UNIVERSE entity identifiers which define MODULE (see Rule 4). *Rule 8*

Each role which has no existence-dependencies is mapped into a subtype for the entity for which it is defined.

The use of roles in the augmented ERM serves to identify subsets of entity types and allow properties to be associated with that subset. Codd defines the categorization of entity types into subsets as a subtype [C-9]. Roles, in the augmented ERM may have properties defined on them as well as constraints. The only restriction is that they may not participate in other relatioships using the same rolenames. Thus, in the SM ERD, parent role is mapped into the E-relation PARENT, which is declared to be a subtype of MODULE.

Rule 9

Each existence-dependent role is mapped to an E-relation which is characteristic of

1. the entity it participates in a non-recursive relationship with, or
2. is characteristic of the role the other entity assumes in recursive relationship.

Since roles assume the status of an entity all of the mapping rules associated with constrained entities apply to roles which have similar constraints placed upon them. Therefore, as a result of applying Rule 4 to the role entity, a characteristic entity for CHILDREN is created. This entity is characteristic of the PARENT entity which was described in the preceding rule.

We have formulated nine rules which provide the mechanism by which an augmented ERD can be mapped into the RM/T model. The rules have dealt with the augments of: generalization, aggregation, expanded role sets, expanded property types and a portion of the expanded relationship representation. We have not provided rules for the mapping of direction, ordering and the concept of *complete* relationships. These constraints have no direct structural mapping and must be included with the operations. Chapter Seven describes the operational mappings.

6.4 Data Base Compiler Algorithm

Application of the preceding rules enables us to develop a DB_COMPILER. The algorithm used by this compiler takes as input the augmented ERD and produces a database schema for that particular ERD. This algorithm uses the rules defined in Section 6.3 to create relations, both E-relations and P-relations, and to modify the existing system relations which were defined in 6.2. Modification of the system tables involves both the insertion of tuples and the modification of attribute values. Level 1 of this algorithm is given in this section using pseudo code although the actual implementation was done in T, a dialect of LISP developed at Yale. This first portion is the main driver which traverses the ERD looking for entities and relationships.The entities are classified as to being *characteristic* or *kernel*. All unmarked relationships are considered *associative*. Following Part I we look at portions of the algorithm in Parts II and III, again using pseudo code.

6.4.1 Algorithm

PART I. Algorithm Transformation: Level 1

```
While traversing ERD do
                for each internal entity do
                 if Characteristic-entity then
                    call apply Rule 5
                 else
                  if not Aggregate-entity then
                   call apply Rule 1
                  end if
                  mark visited
                 end do
```

```
            for each relationship do
                  if not marked done then
                     call apply Rule 6
                     mark done
                  end if
            end do


               for each internal entity not visited do
                  call apply Rule 7
         end do

   End while
```

The main driver uses only Rules 1, 5, 6 and 7. These are the rules which create the major entities found in the RM/T. The boolean functions Characteristic-Entity and Aggregate-Entity use the definitions of these types of entities in order to determine the return value. The marking of a relationship is done if it was subsumed through the creation of a *characteristic* entity and does not produce an *associative* entity. We see in the following portions of the algorithm that the major rules apply the remaining rules. It is within these sections of the algorithm where modifications are made to the existing system table which identifies subtypes of previously defined entity types.

Part II. Algorithm Transformation: Level 2

```
Rule 1
          call Create Kernel-Entity
          If Subtype entity then
            call Modify Sub Graph
          end if
          Mark entity created
End Rule 1

Rule 5

          call Create Characteristic-Entity
          Mark entity created
End Rule 5


Rule 6
          call Create Associative-Entity
          Mark relationship created
End Rule 6

Rule 7
          call Create Aggregate-Entity
          if Subtype entity then
           call Modify Sub Graph
          end if
          Mark entity created
End Rule 7
```

The creation of the four entity types triggers the creations of relations for the representation of their properties. It is at this point that entities of type Designative are determined, either as the property of type reference or as a complete relationship. Roles are considered when creating *associative* entities.

Part III Algorithm Transformation: Level 3

```
  Create Kernel Entity

          Create E-relation
         for each property do
             if single-valued property then
               call apply Rule 2
             else
               if multi-valued property then
                 call apply Rule 3
               end if
     end do

End Create Kernel Entity

Create Characteristic Entity

      Create E-relation
        for each property do
          if single-valued property then
```

```
                call apply Rule 2
            else
               if multi-valued property then
                call apply Rule 3
             end if
           end do

   Create Characteristic P-relation
   Modify CG relation

  End Create Characteristic Entity

  Create Associative Entity

         Create E-relation
         for each property do
              if single-valued property then
                call apply Rule 2
           else
                if multi-valued property then
                  call apply Rule 3
           end if
         end do

         Modify AG relation
         if Complete relationships then
                call apply Rule 5
                Modify Relation relation
                Modify DG relation
         end if

   End Create Associative Entity


  Create Aggregate Entity

        Create E-relation
        for each property do
             if single-valued property then
               call apply Rule 2
          else
               if multi-valued property then
                 call apply Rule 3
          end if
        end do

        for each aggregate component do
          Create characteristic P-relation
          Modify Relation relation
          Modify CG relation
        end do
```

End Create Aggregate Entity

6.4.2 T Classifier

The prototype implementation of the SARA/IDEAS system is in the T language, a dialect of LISP. The DB_COMPILER, considered an auxiliary tool of this system, was also implemented using T. The following represents the introductory T code implementation of Part I of the compiler. Appendix C contains the T code which implements the remaining algorithms described in the previous section.

```
;***********************
;This is the classifier which traverses the erd and
; translates the objects, relationships and properties into
; entities in the RM/T data model
;***************************
(define (traverse-erd)
  ;We first examine each object and determine if it is
  ;either kernel or characteristic. Creation of the appropriate
  ;entity type takes place and the object is marked as created
  ;Order is not predetermined, so it is necessary sometimes to
  ;search for the root kernel entity (inner kernel)

(walk (lambda (x)
  (if (not (checked x))
         (cond ((check-constraints 'E x)
                (if (checked
                     (find-other-object x))
                    (block (set (checked x) t)
                           (characteristic-create x
                                                  (name (find-other-object x))))
                    (block (parent-create
                            (find-other-object x));find inner kernel
                           (set (checked x) t)
                           (characteristic-create x (name (find-other-object x))))))
               (else (block (set (checked x) t)
                            (kernel-create x))))))
 (objects *erd*))
```

6.4.3 Troll code

Figure 3.8 indicates that the input to the DB_COMPILER is an augmented ERD produced from the OReO Compiler. The OReO Compiler is responsible for automatically producing the graphical representation of the ERD along with any additional text needed to completely augment the graph. At this time the OReO Compiler is not implemented and so the input to the DB_COMPILER is a textual description of functional calls which generate the ERD. This text is manually created and merely creates the graphical form found in Figure 6.2. The creation of an ERD is realized invoking functions which make entities, make relations, and make links connecting entities and relations. The parameters to these functions include all of the properties and all the constraints placed on entities, relations and roles. The role constraints are included in the parameter list of the *link-arc* function

```
;********************
; This file is a test file to generate an augmented ERD
;*************************
(create-erd)

;The following section creates all of the entities of the ERD

(create-object "module"
  '((properties ("name" "string"  M)) (constraints A Comp INT)))
(create-object "sockets"
  '((properties ("name" "string"  M)) (constraints E INT)))
(create-object "interconn"
  '((properties ("name" "string"  M)) (constraints E  INT)))

;The following section creates all of the relationship types of the ERD

(create-relation "modic"
  '((constraints E)))
(create-relation "modsoc"
  '((constraints E)))
(create-relation "has"
  '())
(create-relation "link"
  '((constraints E)))
```

```
;The following section links an entity type to a relationship type.

(link-arc
 (find-object objects *erd* "module")
 (find-object relations *erd* "modic")
 '1 '())
(link-arc
 (find-object objects *erd* "interconn")
 (find-object relations *erd* "modic")
 'N '())
(link-arc
 (find-object objects *erd* "module")
 (find-object relations *erd* "modsoc")
 '1 '())
(link-arc
 (find-object objects *erd* "sockets")
 (find-object relations *erd* "modsoc")
 'N '())
(link-arc
 (find-object objects *erd* "sockets")
 (find-object relations *erd* "link")
 '1 '())
(link-arc
 (find-object objects *erd* "interconn")
 (find-object relations *erd* "link")
 '1 '((role "inside") (constraints E)))
(link-arc
 (find-object objects *erd* "interconn")
 (find-object relations *erd* "link")
 '1 '((role "outside") (constraints E)))
(link-arc
 (find-object objects *erd* "module")
 (find-object relations *erd* "has")
 '1 '((role "parent")))
(link-arc
 (find-object objects *erd* "module")
 (find-object relations *erd* "has")
 'N '((role "children") (constraints E ID)))
```

Figure 6.8 Code to Create Augmented ERD

The output of the DB_COMPILER is a Troll database schema for the tool which is being developed. We have chosen to use the Troll database management system for the reasons described in Chapter Four. The resulting relational schema differs from a more traditional relational schema, such as was described in Section 2.2.1, in that there are in general more relations each of a smaller degree. This is a result of having individual P-relations for each object and maintaining relations which indicate the molecular structure of the objects - the system tables described in 6.3. Figure 6.9 shows the resultant Troll schema for the SM Editor. We have identified each object and relationship by preceding the respective Troll relation with a label (in italics) indicating which object, relationship, or role is represented by the relation(s).

```
;Aggregrate Object Universe
relation Universe   [
        uid      :surrogate  (1)]

;Object Module
relation Module    [                 relation ModuleU   [
        mid      :surrogate  (1)]            mid      :surrogate  (1)
                                             uid      :surrogate  ]
;Object Sockets
relation Sockets   [                 relation Socketsname   [
        sid      :surrogate  (1)]            sid      :surrogate  (1)

relation SocketsM   [
        sid      :surrogate  (1)
        mid      :surrogate  ]
;Object Interconn
relation Interconn  [                relation Interconnname   [
        iid      :surrogate  (1)]            iid      :surrogate  (1)
                                             name     :string  ]
relation InterconnM [
     iid      :surrogate  (1)
        mid      :surrogate  ]
;Relationship Link
relation Link   [
        lid      :surrogate  (1)]
```

```
;Role Child
relation Child  [                 relation ChildP   [
        mid     :surrogate  (1)]          mid      :surrogate  (1)
                                          mid      :surrogate  ]
;Role Parent
relation Parent  [
        mid     :surrogate  (1)]
;Role Inside
relation Inside  [                  relation InsideS   [
        iid     :surrogate  (1)]            iid      :surrogate  (1)
                                            sid      :surrogate  ]
;Role Outside
relation Outside  [                 relation OutsideS   [
        iid     :surrogate  (1)]            iid      :surrogate  (1)
                                            sid      :surrogate  ]
```

Figure 6.9 Troll Relations for SM Editor

The relationships MODSOC, MODINT, and HAS are found in the system tables and are not represented as a relation in the schema. The translation rules absorb these relations when a participating object is of type *characteristic*. The CharGraph system relation indicates the name of the relationship as the attribute *Reltype*. Figure 6.10 contains four of the system tables, previously defined in section 6.2.1.

CharGraph Relation for SM Editor

ERelName	ChRelName	Reltype
Module	Interconn	Modinc
Module	Sockets	Mosoc
Parent	Child	Has
Sockets	Inside	Link
Sockets	Outside	Link
Universe	Module	Contains

SubGraph Relation for SM Editor

SubRelName	SupRelName	Status
Child	Module	ISA
Inside	Interconn	ISA
Outside	Interconn	ISA
Parent	Module	ISA

AssocGraph Relation for SM Editor

AssocName	Keyid	ERelName
Link	Iid	Inside
Link	Iid	Outside
Link	Sid	Sockets

PropGraph Relation for SM Editor

PRelName	ERelName
Childp	Child
InsideS	Inside
Interconnm	Interconn
Interconnname	Interconn
Modulename	Module
Moduleu	Universe
Socketsm	Sockets
Socketsname	Sockets

Figure 6.10 SM Editor Graph Relations

Had we used the more traditional relational approach for representing the SM objects and relations the amount of relations would be reduced by about 50%. However, the increase in the amount of code required in the application programs would be a comparable increase (Chapter Seven). These base relations, together with the system meta data, enable the tool developer to provide query capabilities and access operations on the information content of the tool as well as the structural content of the tool. Informational queries would include the standards such as: find module where name= 'A'; find sockets where name = 'B', etc. Structural queries available would include, but not be limited to: list the children of module 'x'; list the parent of child 'y'; list the components of design 'z'; give the subclasses of module(if any), etc. Chapters Seven and Eight continue the development of access and modification to the database.

6.5 Summary

This chapter first briefly defined the RM/T extended relational model. This was the model chosen to serve as the implementation model for the development of the CADIS database. Although this chapter focused on the mapping of the augmented ERM into this relational model, the mapping could have been into some other model. The philosophy behind the Tool Building Methodology [Worl86] supports the use of a user conceptual model which may be used to create schemas for any existing database system. The fact that the basic ERM has been mapped into various traditional models lends support for this philosophy.

We presented the translation rules used in the mapping process. The majority of these rules were seen to be a direct result of the similarity of basic constructs found in the augmented ERM and the RM/T. However, the augmented ERM contained some constructs which were not directly supported by the RM/T. The rules involving the mapping of roles did not have a natural correspondence. Based on the definition of roles as subsets, we were able to make use of the subclass entity in order to map roles.

The concluding sections presented fundamental portions of the algorithm designed to automatically generate the underlying database schema for the tool's ERD. A partial database schema was illustrated in order to give the reader an idea of the underlying structure. The number of relations generated by the DB_COMPILER seem to be numerous when compared to the number of relations found in standard relational data base systems. However, the realization that we are able to maintain not only the standard information, but also the structural information outweighs any disadvantage arising from having a large number of relations. A major advantage seen in the use of the DB_COMPILER is the reduction of time needed in order to generate the Troll relations and also to modify the system relations which maintain the semantics of the the tool's objects and relationships.

CHAPTER 7

The System Level

The system level of the CADIS architecture combined with the kernel level comprises the conceptual model of the CADIS data base system. We regard CADIS as a type of aggregate object containing objects and relationships needed to provide the data management support for both the SARA/IDEAS tool building system and the SARA/IDEAS run-time system. One such object type within CADIS is the SYSTEM. The SYSTEM object contains objects and relationships defined by a system developer, as well as objects and relationships defined by the tool developer. It is responsible for maintaining relationships expressed between tool objects, maintaining relationships between tool objects and system objects, and maintaining relationships between the various system objects.

Many of these relationships are defined as a new tool is brought into the system. Integrating a new tool into the SARA/IDEAS system requires integrating the tool's data model into an existing global system data model. The augmented ERD and the tool's database schema serve as input to a DB_INTEGRATOR.

This chapter defines the SYSTEM object. A description of some of the system objects, generic and non-generic, provide insight into the functioning of the CADIS database system. Operations performed by these system objects are detailed. Following this description, system relationships are defined and examples of their use are given. The concluding section presents a description of the CADIS database and presents an approach by which a tool is integrated into the CADIS database.

7.1 The System

Figure 7.1 illustrates the relationships which exist between the TOOL object, KERNEL object, and SYSTEM object. The aggregate object containing these objects and relationships is the CADIS data base management system. CADIS is the object through which all interactions with the underlying database takes place.

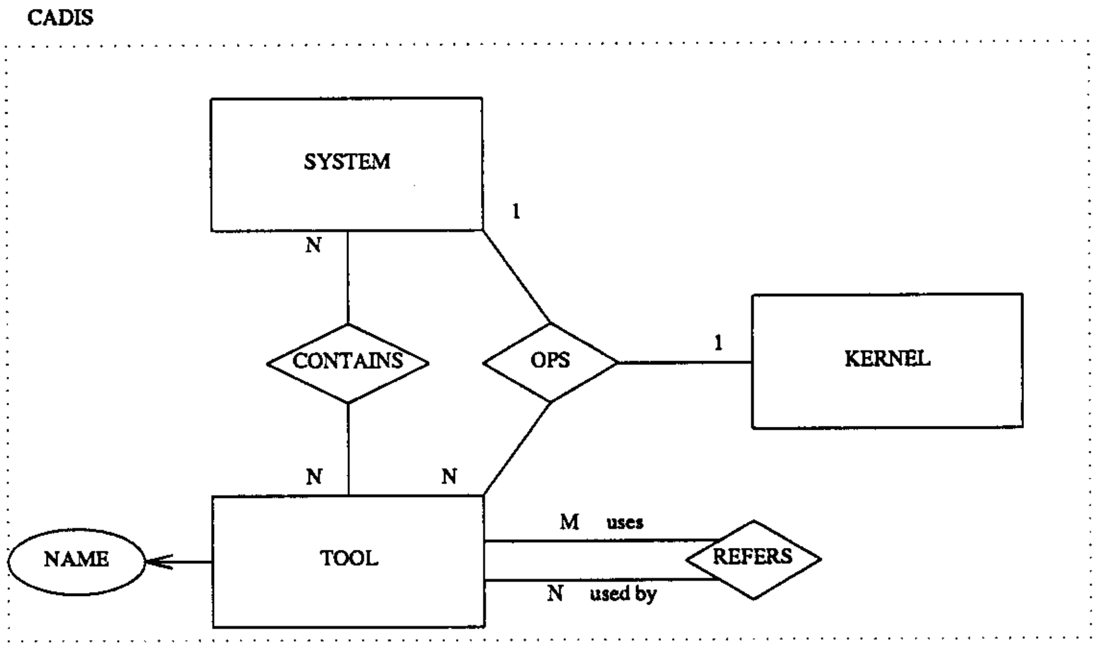


Figure 7.1 ERD of CADIS

The object TOOL consists primarily of user defined objects. These tools have been described as those built specifically for the purposes of designing models and/or evaluating and analyzing models, such as the GMB editor, the Simulator, or the Control Flow Analyzer (Chapter Three). There might also be auxiliary type tools such as tools built to maintain versions of design (RCS), configure designs (MAKE), or provide browsing capabilities to the users of the system (SARA/IDEAS Browser).

We have also previously described (Chapter Five) how the tool builder is able to use objects which have been created by other tools. An example of such a capability is found in the GMB ERD, in which the GMB ARC is associated with SM SOCKETS. The GMB tool builder creates a relationship (REFERS) with an *external* object, the SM SOCKET. Other relationships can also be established during the process of specifying the tool's objects and relationships. The SYSTEM is the object through which instances of these relationships (REFERS) are established.

The object KERNEL consists of the mechanisms which allow the underlying database objects (Troll relations) to be accessed, created, and modified. The KERNEL contains those operations which send messages to perform operations on the underlying Troll database. Both the TOOL object and SYSTEM object communicate with the KERNEL via the relationship OPS.

The SYSTEM object contains objects and relationships which are defined by the system developers. These system developers are the database experts, whereas the tool developers are considered the non-database experts. It is via these objects that system policies are defined and implemented. Such policies might include: defining the level of granularity at which to lock objects ; defining the different types of protection to provide for tool objects (read, write, or modification of selected values); the extent of update propagation as changes in objects occur (whether to cascade changes to relationships of objects); types of notification to provide for changes in a design object. SYSTEM relationships provide the interface between these various objects as well as between tool objects and system objects. The TOOL object becomes a component of the SYSTEM object after integrating the tool into the SARA/IDEAS system.

The top level view of the CADIS data base management system conceptualizes CADIS as an object which has operations defined for its manipulation. Initial interaction with CADIS occurs when a request is made to access a database object. Operations to access any tool's database are invoked by sending a message to the object CADIS. The tool builder, as well as the end user, does not need to know how to formulate complex database requests. The database is simply accessed in terms of the objects or relationship defined for the tool (section 5.6.2). CADIS in turn then accesses these various objects through the SYSTEM object.

The following two sections describe objects and relationships found in the SYSTEM.

7.2 System Objects

The previous chapters, defining the conceptual model in terms of objects and relationships, concentrated on the representation of information concerning these objects through its properties, relationships, and constraints. The augmented ERD model served the primary purpose of providing a non-database expert with the ability to aid in the specification of the tool's underlying database through generation of the tool's specifications. However, we did not discuss at that time the other dimension of objects. This dimension is the object's capability to perform various operations. Borrowing from the SMALLTALK [Gold83] definition, objects are considered to be a uniform representation of information that is an abstraction of the database's capabilities. Database capabilities include the storage of information and the accessing of information. Operations can be performed on objects through the use of messages sent to the object. An object operates on itself when another object sends a message to do so. Methods defined for each object represent these various operations. Groups of objects which respond to messages in a similar way are considered classes. These classes correspond to the concept of a generic object, as the inheritance of methods from a class to a subclass (super object to sub object) is supported.

The kinds of objects and classes needed to represent a data base management system's capabilities and the operations supported by these objects are considered out of the realm of an expert tool builder. Therefore, CADIS provides these supporting objects in order to achieve such functionality. This relieves a tool builder from the additional task of having to define specific database operations

7.2.1 Generic Objects

The SYSTEM object contains generic types created by the system developers. These objects are provided so that the tool objects can be classified as belonging to a specific subtype and thus inherit database specific methods defined for these objects. Database activity involves access operations and/or modification operations. Hence, two objects are required to support both activities. The two generic objects, defined for CADIS, are an access object and an instantiation object. All SARA/IDEAS tool objects are placed into a sub category of the access object and a sub category of the instantiation object. These subsets are analagous to the SMALLTALK class concept, hence the terms generic object and class are used interchangeably.

In the SARA/IDEAS system, one such message sent to a MODULE object would be:

Retrieve M1@

where M1@ is the database's unique identifier of a MODULE object.

The method which is used to respond to this message depends upon which subclass of access object the tool builder had originally designated MODULE to be. The two classes available for the tool builder to use are COMPLEX and SIMPLE. Recall that the SM Editor creates a somewhat complex object, the MODULE. Figure 7.2 illustrates the complexity of this object. The fact that SOCKETS and INTERCONN are contained in MODULE is noted by the dependency constraint. This constraint affects the retrieval of the complex MODULE, in that instances of SOCKET and INTERCONN must also be retrieved if the entire structure of MODULE is to be retrieved. Another indication of the complexity is that an instance of a MODULE which is a *parent* needs to have all of its *children* retrieved. Since this is a recursive relationship, then all the *children* MODULE(S) are retrieved recursively.

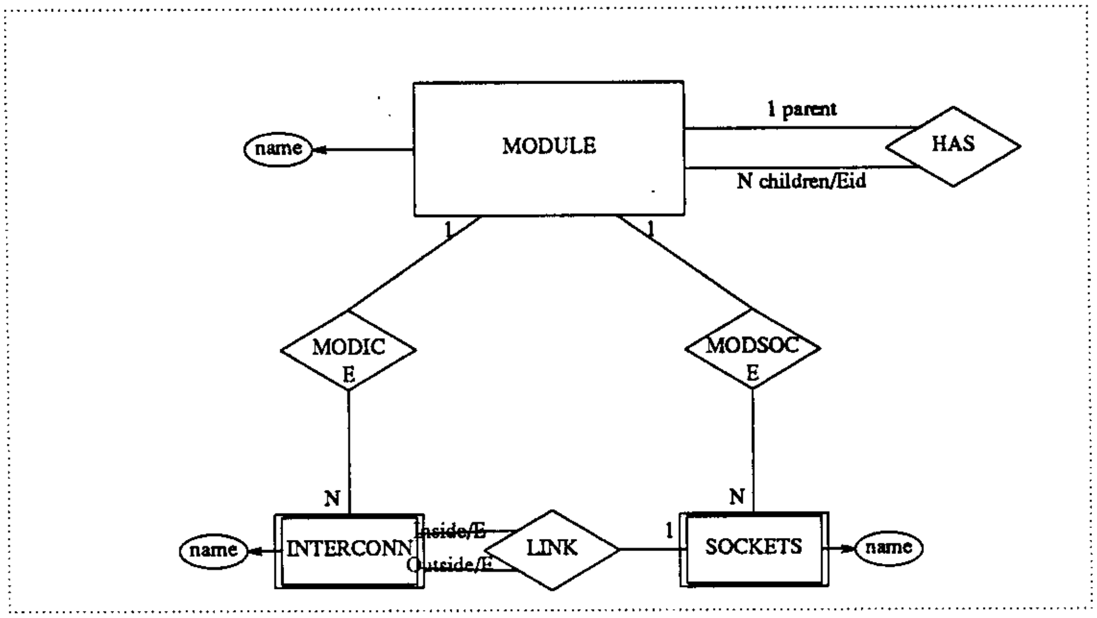


Figure 7.2 SM ERD

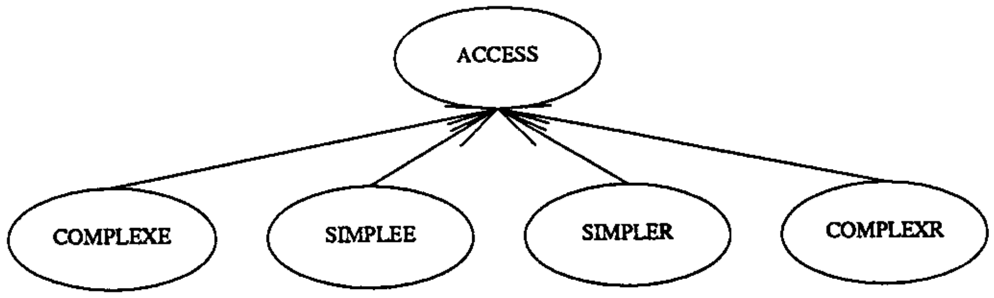


Figure 7.3 Access Class Hierarchy

ACCESS CLASS:

Figure 7.3 shows that there are four subclasses of the generic *access* object. This object provides the tool developer with the capability of identifying the unit of access. In a traditional DBMS, the unit of access is a record. However, since complex objects are found in a design environment, and do not necessarily comprise a record, there is a need to change the unit of access to something more meaningful to the end user as well as the designer. A system defined method is provided for each generic object. If there does not exist an access method for an object or relationship, then that object *inherits* the access method defined for the object's superclass. In the CADIS system, we can regard the implementation of such an inheritance mechanism as the following:

```
    Case Id-type
    Object:     case Access class
                    Simple : SIME
                  Complex : COME
                    End case
  Relationship: case access class of
                    Simple : SIMR
```

```
            Complex : COMR
            end case
      End Case
```

Figure 7.4 is an instance of such an SM Editor design used to illustrate the definitions of the various classes of access.

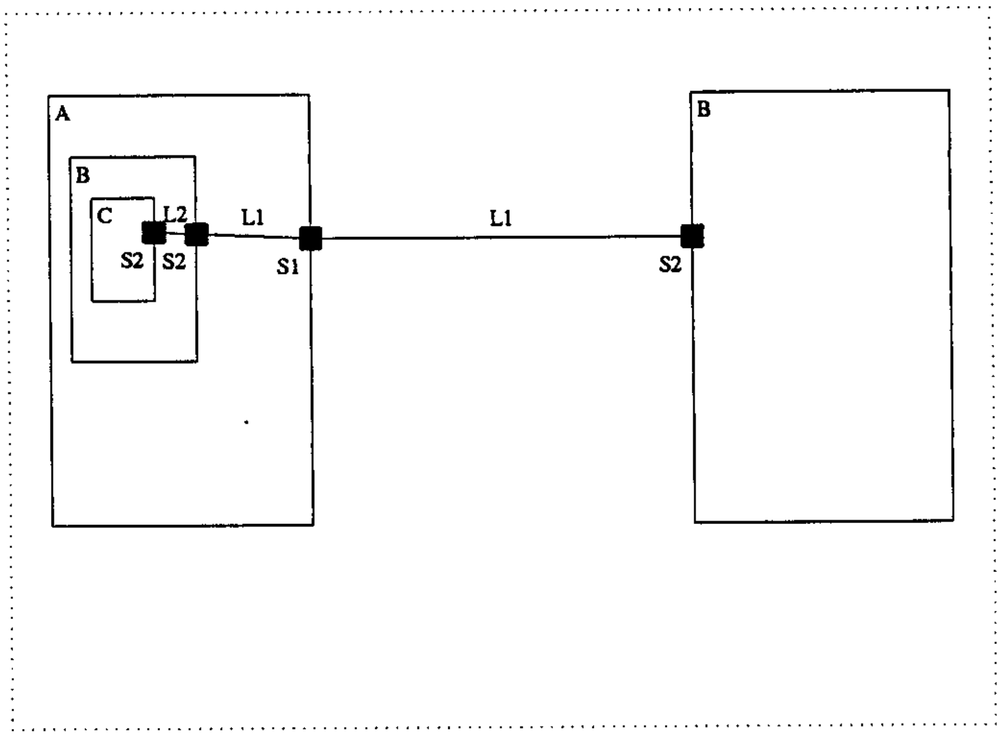


Figure 7.4 SM Instance

Simple Entity Access:

A simple entity access object is *null expanded.* A null expanded object is one which is accessed at the top level. That is, in response to a request to retrieve a simple entity, the immediate properties are accessed along with a list of the names of the entities are contained within this entity. Also included is a list of relationships in which this object participates. This type of access is information bearing but does not provide the structure of the object to the user. A typical use of this type of access would be by a browser, which is interested in properties of the object, but does not need to recreate the multi-dimensional structure of the object. This classification is designated by placing "(SIME)" in the rectangle defining the object type. This is the default classification of all entities.

SIME Access Algorithm

```
For object-id do
    Foreach prop in property-list do
      Retrieve (prop,value)
    End do

    Foreach type in relationship-list do
      If type = DEP then
        Foreach key in characteristic-list do
            Retrieve (key,value)
        End do
      End if
  End do
End do
```

Suppose (Figure 7.4) the unique id for Module A was used as the parameter for the Retrieve operation, then the following information would be the result of this operation.

```
OBJECT MODULE

Name = A;
MODINC INTERCONN name = L1;
MODSOC SOCKETS  name = S1;
HAS Children name = B;
HAS Parent   ;
```

The relationships that the Module participates in are listed and since they are all of type characteristic, the key values of the other participants are also listed. However, the fact that L1 links S1 and S2 is not known from this retrieval, indicates that the relationship LINK is not traversed. Although the SM Editor ERD shows that LINK is contained within the Module object, Module does not participate directly in this relationship.

Complex Entity Access:

A complex entity access object is one which is *fully* expanded when accessed. A fully expanded object contains all of the information about the object as well as its structural composition. Structural composition includes all of the objects contained within the object along with a list of any relationships the object is associated with. In Figure 7.2, MODULE, when classified as a complex entity is accessed in its entirety. That is, a retrieval of MODULE will materialize a complete module along with all objects contained in it, such as sockets, interconnections and children modules. The placement of "(COME)" in the ERD rectangle defining MODULE classifies it as a complex entity.

COME Access Algorithm:

```
For object-id do
    Foreach prop in property-list do
      Retrieve (prop,value)
    End do

    Foreach rel-type in relationship-list do
      Retrieve (rel,rel-type)
      If rel-type = DEP then
        Foreach key in characteristic-list do
          Retrieve (key,value)
        End do
      End if
    End do
End do
```

Using the same retrieval request as before, we illustrate the information access if MODULE had been classified as a complex access object.

```
OBJECT MODULE

Name =  A;
MODINC Interconn name = L1;
MODSOC Sockets name = S1;
HAS children name = B;
  OBJECT MODULE

    MODINC Interconn name = L2;
    MODSOC Sockets name = S2;
    HAS children name = C;
        OBJECT MODULE
          MODINC  ;
          MODSOC Sockets name = S3;
          HAS children ;
          HAS parent name = B;
          LINK outside S3-L2;

    HAS parent name = A;
    LINK inside S2-L2;

HAS parent;
LINK inside S1-L1;
```

A complex retrieval returns all of the information with which the structure of the object can be recreated.

Simple Relationship Access:

A simple relationship access object is *null expanded.* A null expanded relationship is one which is accessed at the top level of the relationship. That is, the immediate properties of the relationship and the keys of the entities which participate in the relationship are retrieved. "SIMR" is the designation for a relationship as the unit of access. The algorithm is similar to SIME in that the properties of the relationship are retrieved along with any roles defined on that relationship.

Complex Relationship Access:

A complex relationship access object is one which is *partially expanded* when accessed. A partial expansion returns a portion of the structural content. As an example, the following text shows the results of a retrieval request for the relationship LINK. The immediate properties of the relationship along with the entities, not fully expanded, participating in the relationship are returned. "COMR" is the designative term for this type of relationship.

```
RELATION LINK

Name = L1;
Outside SOCKETS name = S1;
Inside SOCKETS name = S2;
```

There is no default access type for relationships. If a relationship has no designation then access through relationships is not available.

INSTANTIATION CLASS:

The generic object instantiation contains three subclasses: *internal, external and standard.* Each of these generic objects contain system defined methods for modifying any instance of an object belonging in its class. These methods are defined by the system's developers in order to implement the particular policies of the system with regard to modifications: deletion, insertion and updates.

Internal Object:

Internal objects are those objects which are defined by the tool builder. The SARA/IDEAS policy applied to instances of this internal object is that they can be created, destroyed, changed, and manipulated by the user of this tool. If additionally a particular instance can only be modified by the user who created it, then protection constraints can be associated with that particular object as to specifically who or when such modification are allowed. In Figure 7.2 SOCKETS has been defined to be internal. Pictorially this is indicated by the (INT) found in the SOCKET rectangle.

External Object:

Again reference is made to the GMB Editor to illustrate the use of an external object. MODULE and SOCKET entities are external to the GMB tool as indicated by the (EXT) found in reference to the two entities by the GMB builder. This GMB Editor needs to link to objects which have already been created by the SM Editor tool. Use of these objects might be limited by the SM Editor tool designers. However, if the SM Editor tool designer has not designated specific protection constraints, then the SARA/IDEAS policy is that the GMB may access these entities, but may not modify their values. If the linked external object is deleted from the database, a ghost will survive for use by the GMB until a decision is arrived at whether to unlink or relink to a currently existing external entity.

Standard Object:

Standard objects are those objects which have been placed in the SYSTEM for direct use by the tool builder. There are two categories of standard objects: generic and non-generic. Within the non-generic class there are two categories of objects, non-generic templates, and non-generic instances. The tool builder indirectly uses standard generic objects by classifying objects according to their access class and instantiation class. However, the SYSTEM can provide direct use of object types for use in relationships with tool objects. These objects may serve as templates, whose instance variables are given values by the tool user, or as objects whose instance variables have been previously given values by the system.

7.2.2 Non-Generic Objects

Since we have classified standard objects as a subclass of the instantiation object, there is a method associated with these objects which determines how such an object can be modified. The modification policy for each of the types of system objects might be:

1. modifications are not allowed on the methods of standard generic objects

2. modifications are allowed to the instance variables of standard non-generic template objects

3. modifications are not allowed on the instance variables standard non-generic instance objects.

Although the modification policy for generic and non-generic instance objects appears to be similar, the additional constraint is that generic objects may participate only in the predefined relationship ISA, whereas the non-generic object may participate in any user defined or system defined relationship.

An example of a template might be an Authorization object which contains the list of authorized users associated with an SM design. The list of users would be designated by the author of the design at the time the design is stored in the Database. This same standard object could be used to associate a design with an instance of a predefined authorization list.

7.3 Relationships

System relationships are those relationships which are defined by the system. We have previously defined a relationship as being an entity which associates two or more object entities. This relationship may have properties and additional constraints associated with it, such as completeness class, direction and ordering. This is only one dimension to the multi-dimensional relationship. Another dimension is that the relationship defines an interface between related objects. Since we have described objects as communicating with other objects via messages, we similarly can describe relationships as the message bearers. An object may send the same type of message to the same object on two different occasions. Each occasion may result in a completely different response. The different responses are due to the different relationships in which the two objects participate each time. For example, Object A may send a message to object B to access its contents. If its contents consists of a program, then the message may mean to display in textual format the program code or it may mean to execute the program. The relationship which serves as the interface modifies the message so that the receiving object responds correctly for that type of relationship. An analagous situation occurs within some data base management systems which incorporate some form of query modification in order to implement integrity constraints.

CADIS contains predefined relationships ISA, IS_PART_OF, CONTAINS. These relationships modify access operations so that the resultant object makes sense within the context of the relationship. For example, in response to a request to retrieve instances of CONTAINS, an access operation would be applied to each of the types of objects participating in that relationship. These relationships are those which allow us to implement the abstract notions of generalization and aggregation.

The SYSTEM could also contain instances of other types of relationships. These relationships would need to be created by the system's programmers to serve as interfaces to other objects. One such relationship which is of importance in a design system would be the *Version* relationship which provides an interface to some versioning process such as RCS. This relationship maps the store and retrieve request for an object into the appropriate RCS commands to store/retrieve a given version. In other words, the relationship serves as a dispatcher and modifier of a sent message. Other relationships can be defined by the system developers as new tools are brought into the system. This interface capability expands upon the types of tools which can be integrated into SARA/IDEAS tool building and run-time systems.

7.4 DB_INTEGRATOR

The DB_INTEGRATOR is part of the SYSGEN component in the SARA/IDEAS system (Chapter Three). It is the sub processor within the Integration Facility (Figure 3.8) which is responsible for integrating a tool database into the CADIS database. Input to this processor (Figure 4.2) consists of the tool data base declarations generated by the DB_COMPILER (Chapter Six) as well as a listing of any constraints not processed by the DB_COMPILER, the CADIS data base declarations, and any tool-specific data base manipulations. *At this time the processor has not been automated so that this processing is manually performed.* The output from the DB_INTEGRATOR is a modified CADIS data base system, consisting of the CADIS data base declarations and CADIS data base manipulations.

A description of the tool's schema has previously been given in Chapter Six. The CADIS data base declarations contains the schema for representing all of the standard objects defined previously as well as the objects already placed into the system by other tools. Figure 7.5 serves as merely a skeletal ERD representation of the

CADIS database system. A detailed representation would only cloud the salient features of the CADIS database schema.

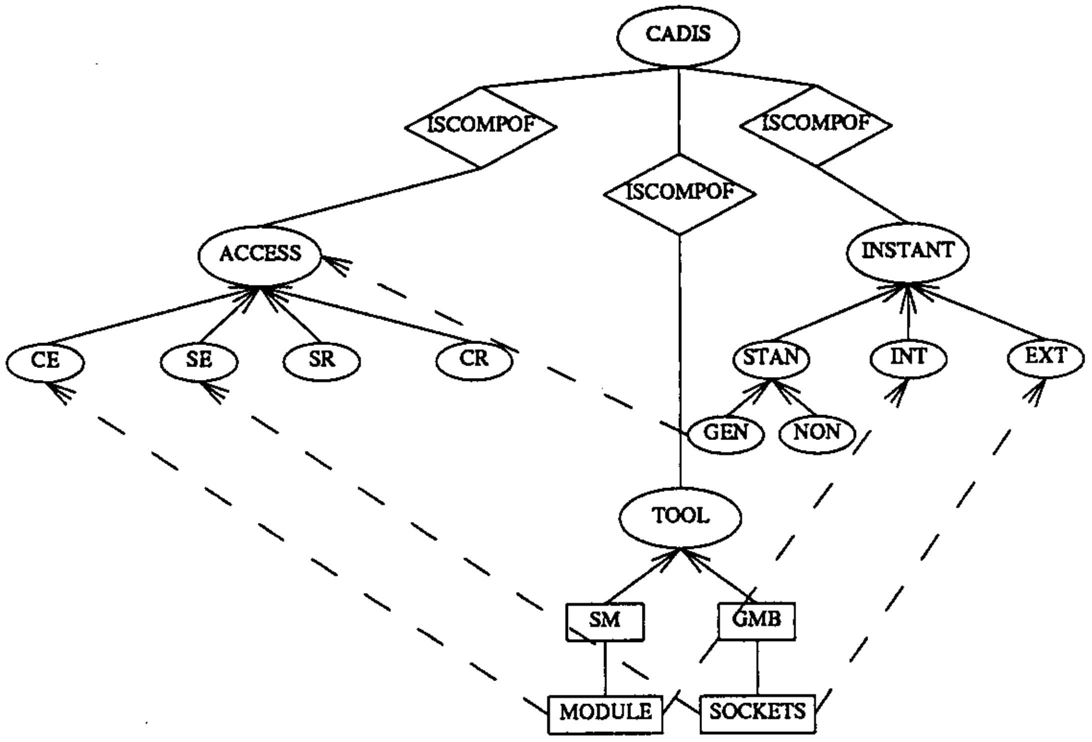


Figure 7.5 CADIS Database System

The CADIS schema is shown to be composed of three objects; the Access object (ACCESS), the tool object (TOOL), and the Instantiation object (INSTANT). We have chosen to represent these three system objects, as well as the object CADIS, using ovals instead of rectangles. This is so that the reader remains aware that these are special system objects that have been previously placed in the CADIS database. The system maintains system catalogs and specially defined procedures for manipulation of these objects.

The various subclasses of these three CADIS objects are also represented as ovals connected to their superclass via an upward pointing arrow. This representation is the standard hierarchical tree representation. Subclasses of the ACCESS object are shown to be the four access classes defined in section 7.2: Complex Entity (CE); Simple Entity(SE); Simple Relationship(SR); and Complex Relationship (CR). The dotted upward arrows of Figure 7.5 show that the object MODULE, represented using the standard ERD notation of rectangle, is classified as a Complex Entity and that SOCKETS is classified as a Simple Entity. We use the dotted upward arrow notation to indicate that this classification is user-defined. This classification identifies the type of access needed for retrieval of such objects. Subclasses of the TOOL object are the SM Editor(SM) and GMB Editor(GMB). MODULE and SOCKETS are the only individual tool objects shown in Figure 7.5 so as not to complicate the figure with too much detail. The Instantiation object (INSTANT) has three subclasses: Standard (STAN); Internal (INT); and External (EXT). The Standard object has two subclasses, Generic (GEN) and Non-Generic (NON). We see also that the ACCESS object is classified as a generic object (GEN), that MODULE is classified as an internal object (INT) and SOCKETS is classified as and external object (EXT) of the GMB tool.

The DB_INTEGRATOR places the tool's database declaration in the database and then proceeds to modify system catalogs indicating that these objects and relationships are those belonging to the specified tool. Then the listing of the constraints which are input to the DB_INTEGRATOR are processed. Such constraints are those represented in the augmented ERD such as the specification of objects as to class of instantiation and access. The DB_INTEGRATOR creates the necessary catalog information to maintain these classifications. Additionally, the relationship classes, directionality and ordering, are entered into the catalogs used by the access and modification database procedures. Finally, information concerning any tool specific database procedures is placed into the CADIS library of procedures. These procedures are invoked by the system as a particular tool calls on them. The process of integration is straightforward, consisting of primarily the modification of existing system relations and files and additionally, the introduction of new relations and files to represent the new tool's objects and relationships.

7.5 Summary

This chapter has described in detail the methods defined for four of the more important system defined objects. These methods provide access to both objects and relationships so that a tool developer need only classify a tool's object or relationship within one of these categories so that she/he might be provided with the complex data manipulation operations necessary to access these entities.

We continued with the description of the SYSTEM object by defining other objects such as the Internal, External, and Standard. Methods associated with these objects provide modification procedures for the CADIS database.

The concluding section briefly discussed steps taken to integrate a tool's database schema into the system's schema. The schema defined for CADIS is easily modified during such integration.

Chapter Eight is a description of a tool, the SI Browser, which enables a tool developer to browse through this schema.

CHAPTER 8

The Tool Level

The concept definition phase presented in [Worl86] (Chapter Three) is applied to the description of a new SARA/IDEAS tool, the *SI Browser*. In general, a browser presents information about the underlying data relevant to a system. In particular, the *SI Browser* presents information about the data of the SARA/IDEAS System which is relevant to the development of a new tool. We have described that data in previous chapters. The basic data, that is of importance to the tool designer, is that data associated with the CADIS System object and the CADIS Tool object (Chapter Seven).

After describing the fundamental characteristics of the *SI Browser* , each step within the concept definition phases of the Worley Tool Building System is applied. The complete concept specification corresponding to the ERD of the SI Browser is presented. The output of this conceptual definition phase is then described.

8.1 The SI Browser

The concept of a browser is taken from the use of browsers in the SMALLTALK system. [Gold84] The SMALLTALK programmer typically creates special purpose browsers, either as additional software development tools or as applications.

One such software development tool is the system class browser. In SMALLTALK, the main way to find out about classes in the system is to use the system class browser. This browser presents a hierarchical index to such information. The index is independent of programming logic; it is designed solely for user access to class descriptions via subject categories. The browser presents categories that organize the classes within the system, and categories that organize messages within each class. SMALLTALK browsers have views and dependent subviews. A view consists of information choices and a subview consists of the information itself. The SMALLTALK user is provided with editing capabilities in the subview. Such editing capabilities include the addition of new categories, the renaming of categories, the removal of categories, and the updating of views.

An example of a SMALLTALK application browser is a budget browser, which browses a database of financial plans, income and expenditures. The designer of this browser determines the structure of the information displayed to the end user.

The SI Browser, similar to the SMALLTALK system class browser, is designed for use by a SARA/IDEAS tool builder in order to find out about the information structures of the system objects, including the objects defined by other tool designers. It is a SARA/IDEAS software development tool. Unlike its counterpart, this browser provides only a limited amount of editing capability because the end user does not necessarily have the expertise which enables him/her to edit the structure of the CADIS database architecture. The information about the System objects that you can retrieve using a browser includes:

- a comment about the role of the object in the system;
- a description of the level in the CADIS hierarchy in which the particular object is found;
- a description of the subclasses of the object;
- a description of the relationships in which the object participates;
- access to operations defined for that object.

The remaining paragraphs describe the overall functioning of the SI Browser tool. This description is in the format of a scenario between the end user and the SI Browser. Following this description, the objects, relationships, and operations will be defined.

The SI Browser consists of a view. Displaying the browser view brings up a list of the objects contained in the system, as well as a paragraph description of the SARA/IDEAS System. Associated with this view are a set of view commands. Each view has appropriate commands defined for it, such as: *select; addcategory; edit; and quit.*

Figure 8.1 is an illustration of what an initial view display might look like. We will refer to this display as the menu. The tool developer chooses a general format for all menus: the upper left corner contains the information choices; the area to

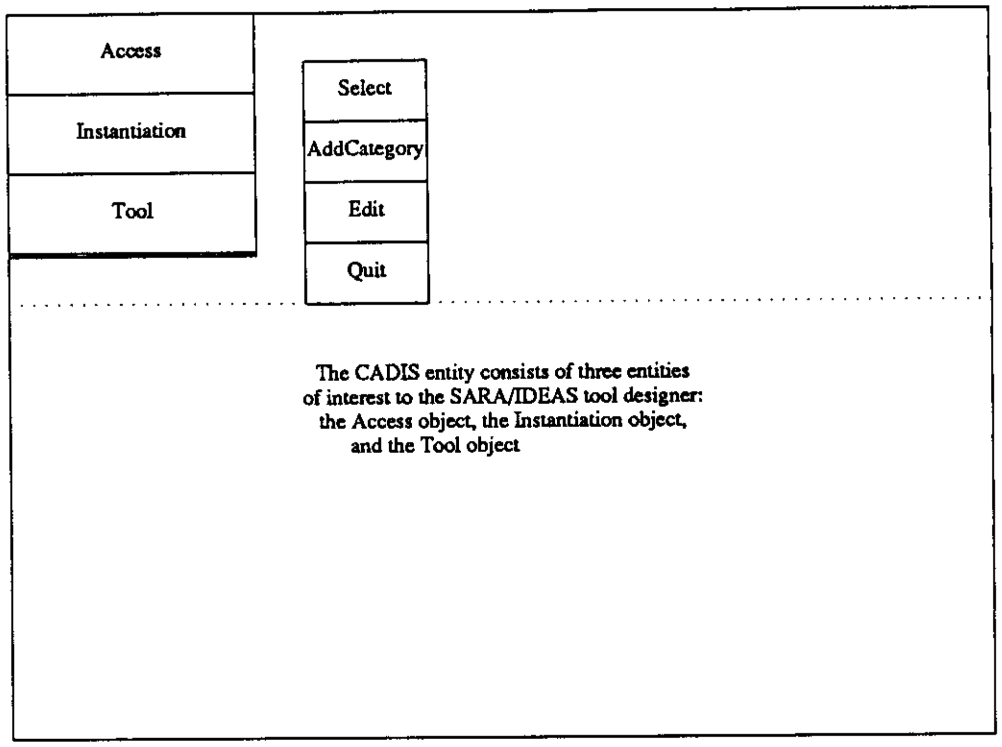
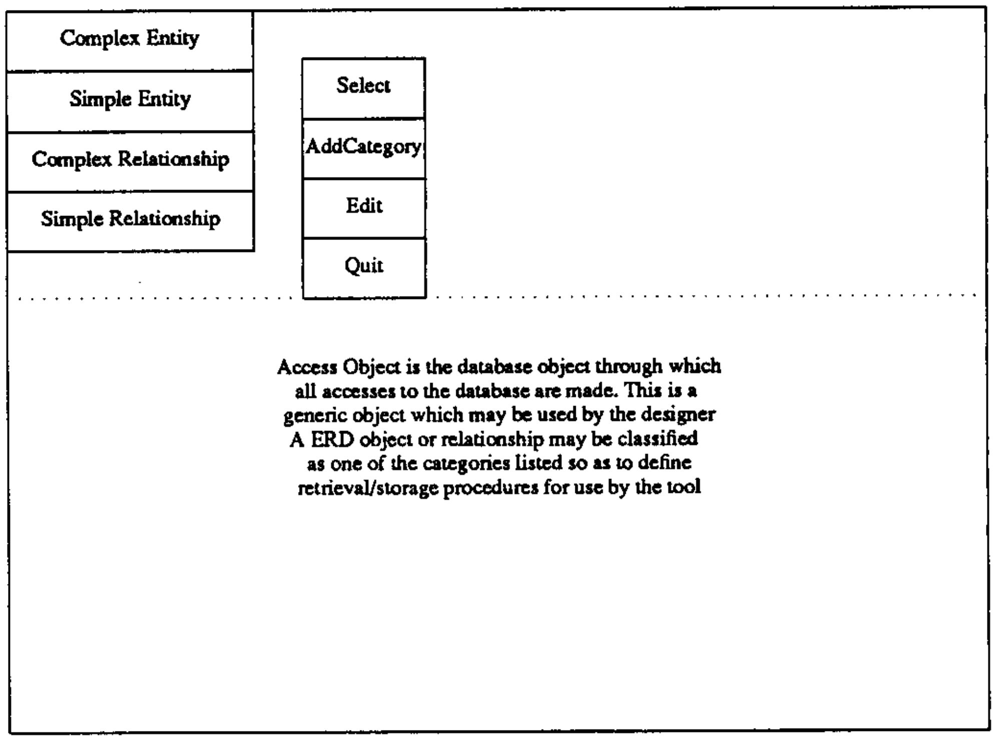

Access
Instantiation
Tool
Select
AddCategory
Edit
Quit
The CADIS entity consists of three entities of interest to the SARA/IDEAS tool designer: the Access object, the Instantiation object, and the Tool object

Figure 8.1 SI Browser Initial Menu

the right of the information choices contains a list of permissable menu commands; the bottom third of the menu contains descriptive text. The menu of Figure 8.5, Structure Model View, differs from this menu in that an additional list, between the information choices and the menu command list, indicates those components associated with a selected item from the information choices. Selection of choices is through either a command line or mouse, etc. This is defined in the logical - physical definition phases of the tool's development. In Figure 8.1, the view is displayed in the upper left corner, the descriptive text in the bottom part of the screen, and the menu commands to the right of the view display. The initial menu of Figure 8.1 shows the data of interest in the SARA/IDEAS System. This is the top level in the CADIS hierarchy consisting of: the Access object, the Instantiation object, and the Tool object. The commands which the user may choose, within the initial menu, to invoke are:

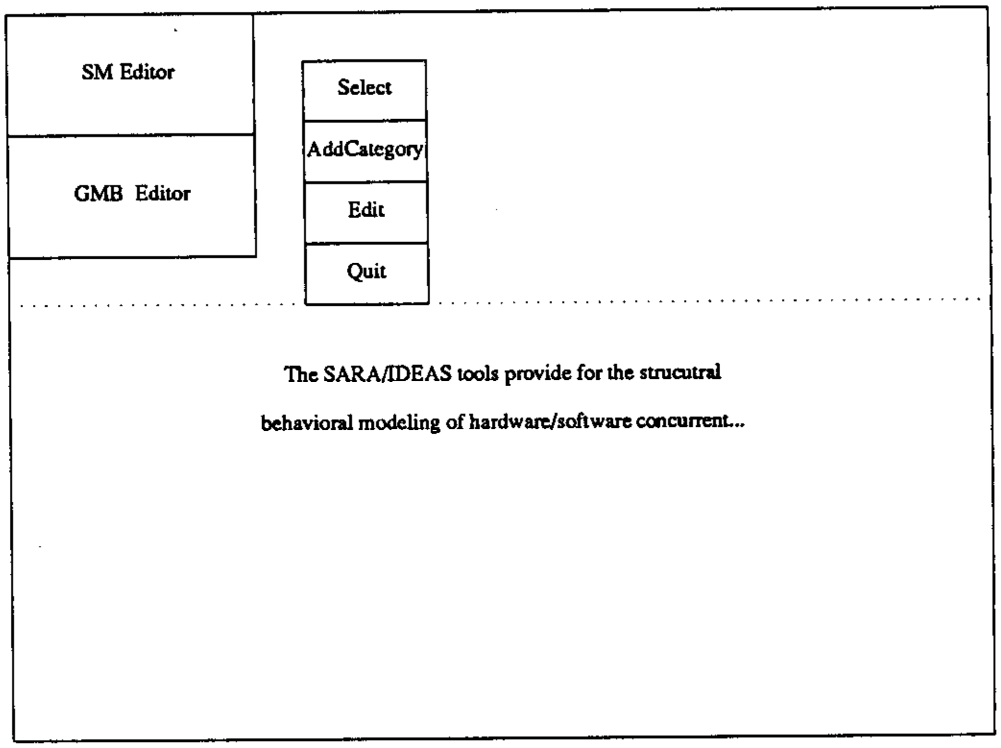
- *Select:* selection of a view choice causes another view or instance to be displayed;
- *AddCategory:* places the user in edit mode so as to enter in name of new category;
- *Edit:* places the user in the edit mode in the bottom part of display so that the text may be edited;
- *Quit:* places the user at the previous level of the hierarchy.



Access Object is the database object through which all accesses to the database are made. This is a generic object which may be used by the designer A ERD object or relationship may be classified as one of the categories listed so as to define retrieval/storage procedures for use by the tool

Figure 8.2 Access View

Figure 8.2 is the display resulting from selection of Access in the initial menu. The view of the Access category is one which contains information choices of Complex Entity, Simple Entity, Complex Relationship, Simple Relationship. The descriptive text defines an Access object. If the user chooses the menu command AddCategory, then a new type of Access object can be defined. However, this capability is restricted to users with system status so as to maintain the integrity of the database structures.



The SARA/IDEAS tools provide for the strucutral behavioral modeling of hardware/software concurrent...

Figure 8.3 Tool View

Selection of the view choice Tool (Figure 8.1) brings up a view which displays all of the current SARA/IDEAS tools in the system. Again, the menu commands provide the user with the capability of browsing further by selecting one of the tools, adding a new tool name to the list, editing the descriptive text, or returning to the previous menu. Figure 8.3 is a display of the Tool view.

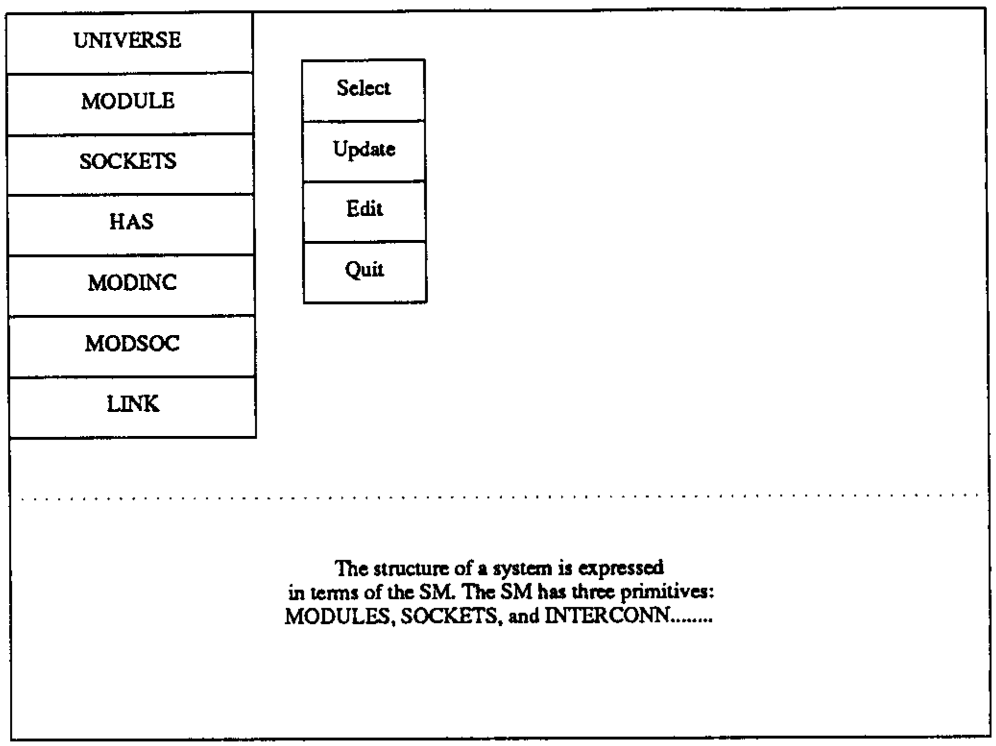

UNIVERSE
MODULE
SOCKETS
HAS
MODINC
MODSOC
LINK
Select
Update
Edit
Quit
The structure of a system is expressed in terms of the SM. The SM has three primitives: MODULES, SOCKETS, and INTERCONN........

Figure 8.4 SM Editor View

Selection of the SM Editor entry in Figure 8.3 brings up a new view; the SM Editor view (figure 8.4). This view has entries which consist of the database entities defined for the SM Editor. These entities are the objects and relationships found in the augmented ERD (Figure 5.5). Since the initial object of the SM Editor is the aggregate object UNIVERSE, it is included in the listing. The menu command list for this view differs from the previous menu command lists. The menu choices at this level in the CADIS hierarchy are: *Select, Update, Edit and Quit.* Select, edit and quit operate in the same manner. However, AddCategory is replaced with Update. At this level in the hierarchy the user is not allowed to add ERM entities. Choosing the Update entry results in the database being searched for any modifications to the SM Editor's ERM. If it has been modified since the SM Editor View had been created then a new SM Editor View will replace the one on the screen. The user is not allowed to modify the view in an arbitrary manner.

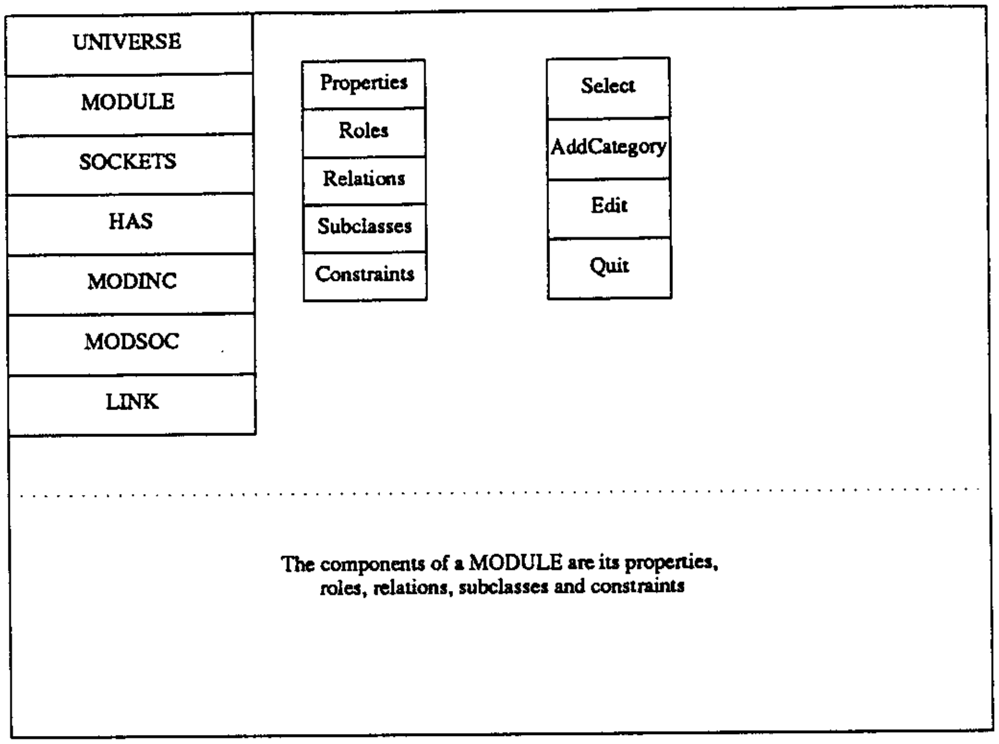


Figure 8.5 Component Subview of Structure Model

Figure 8.5 shows the display of a subview of the SM Editor view. This subview is dependent upon the selection chosen by the user. Selection of MODULE brings up the subview which lists properties, roles and relationships as the components of MODULE, whereas, selection of HAS brings up the subview which lists properties, objects, relationtypes as components of HAS.

Figure 8.6 is an illustration of the display which results from the selection of *roles* in Figure 8.5.

The roles of MODULE are themselves objects and so the display will have a format similar to that of figure 8.4 Since roles are limited to objects the components are listed at the same time. Note that neither the component *roles* nor the components *subclasses* is included in the menu. The reason for this omission is that roles do not

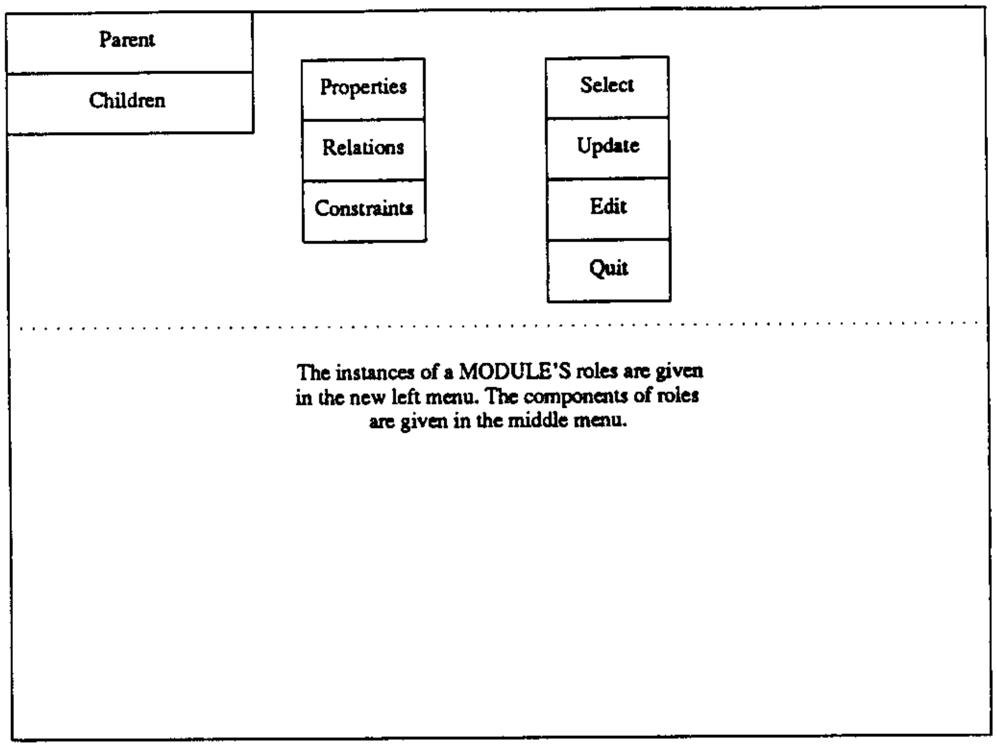

Parent
Children
Properties
Relations
Constraints
Select
Update
Edit
Quit
The instances of a MODULE'S roles are given in the new left menu. The components of roles are given in the middle menu.

Figure 8.6 Role Instance

have other roles associated with them, nor do they have any subclasses associated with them. Selection of properties will list all of the attributes associated with that particular role.

8.2 SI Conceptual Specification

According to the Worley methodology, the first step in the design of the SI Browser is to identify the objects that the user must be aware of. These objects are viewed and manipulated through tool use. The end-user of the Browser can view this tool as an object with which to interact. The end-user will also recognize some other objects within the Browser tool. First on our list is to describe those objects that the end-user sees and eventually manipulates via the user interface. Worley has provided a Concept Definition Language(CDL) to use in order to describe these objects. We use both the CDL and the augmented ERD in this section to describe the SI objects. Figure 8.7 represents the designer's conceptualization, via the ERD, of the SI objects, their properties and relationships. The conclusion of this section gives the CDL description of each object. The sections following the CDL objects' description then describe each object's attributes, relationships, and defined operations.

8.2.1 Object Identification

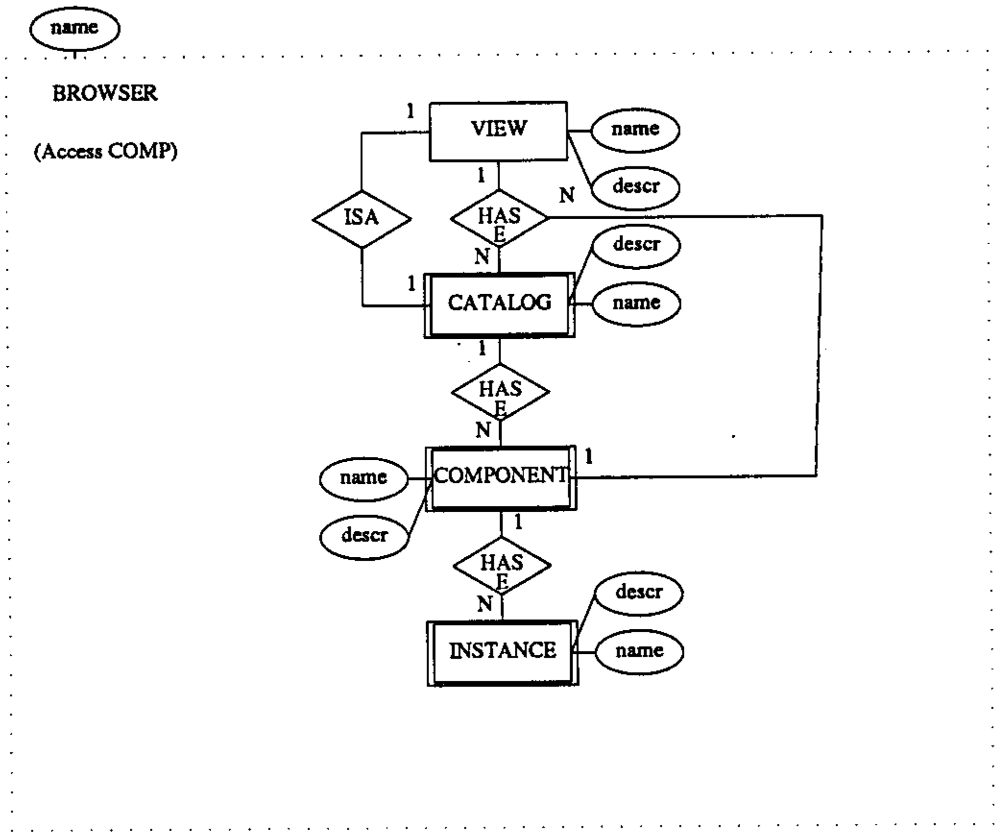


Figure 8.7 SI Browser ERD

BROWSER: The initial step in the design of a tool is the determination of the tool's objects. We have previously stated that the Browser itself is an object (BROWSER) and we choose to associate with this object a name. We have associated a name with the BROWSER so that later, additional browsers could be designed which would browse portions of the system other than the data base.

VIEW: Since the Browser was defined in Section 8.1 to *contain* a view, another ERD object shown in Figure 8.7 is a view (VIEW). We indicate this containment in Figure 8.7 by use of the dotted aggregate notation which was discussed in Chapter Five. When retrieving the Browser object whose name is SI, we desire that all of the objects contained in this aggregate object also be retrieved. Therefore, we classify the Browser object as a complex entity. This is indicated by the textual portion "Access COMP" in Figure 8.7. This view object is to represent information about each individual system object, represented in Figure 7.5 as ovals. A VIEW has a name, its system name, such as CADIS, ACCESS, etc. Another property of this object is a textual description of this system object. Thus, we will later need to indicate the augmentation described in Chapter Five which provides for the support of properties of type *text*. This mapping of types to attributes is done using the CDL (section 8.2.3).

CATALOG: Since the objects in Figure 7.5 have subclasses, which are themselves objects, we need to define another Browser object to represent this classification. We consider these subclass objects *catalog* type objects. Thus, we define the catalog object (CATALOG). Also, as seen in Figure 7.5, the system creates a type of tool object for each tool introduced into the CADIS database. These tool objects *contain* user-defined objects. Therefore, in addition to subclasses being defined as a CATALOG object, we consider the user-defined objects to also be *catalog* type objects. The relationship between a VIEW and a CATALOG is defined to be "has." A VIEW *has* many CATALOG objects associated with it. This is graphically represented in the augmented ERD as the 1:N relationship HAS. The object CATALOG also is shown to have an existence dependency upon the object VIEW, indicated by the double box enclosing CATALOG and the "E" within the diamond. This dependency represents the semantics that such objects cannot exist independent of their parent object. Catalog objects also have name and description attributes. In CADIS (Figure 7.5), such objects have the name "CE", "CR", "MODULE", "GMB", etc. Figure 8.7 becomes more complex with the inclusion of the 1:1 relationship "ISA" linking CATALOG and VIEW. This is necessary to enable us to represent the fact that if a catalog object is a system subclass then it ISA view. In other words, a subset of catalog objects are also each considered a view.

COMPONENT: Since some of the CATALOG objects in the Browser represent user-defined objects and relationships which were generated using the augmented ERM, we need to associate with those particular CATALOG objects another object which represents the characteristics of the augmented ERM (Chapter Five). This object is defined to be a COMPONENT object. Component objects represent augments defined in Chapter Five such as roles, properties, generics, aggregates, relationships. These augments constitute the meta data of the CADIS database. Associated with each COMPONENT are a name, indicating which of the above augments are associated with the particular component, and a textual description. Note that some CATALOG objects will have only one COMPONENT, whereas, some will have a COMPONENT object for each type of augment defined in Chapter Five. For example, if the CATALOG object associated with the COMPONENT is a user-defined *object*, then the individual COMPONENT objects would include a component whose name is "property", a component whose name is "role", a component whose name is "constraint", a component whose name is "relationships", and a component whose name is "subclass".

INSTANCE : Since the user has defined attributes for objects and relationships, and also defined specific relationships using generic objects and aggregate objects, we need to represent these definitions. We define an object INSTANCE which represents the actual instances of properties and constraints associated with a particular COMPONENT object. To illustrate, the component object whose name is Property and which is related to the MODULE catalog object will have an instance object associated with it which has the name of *Name* and a description which indicates that this property *name* is of type *string*. We are also able to indicate to the end-user of the Browser, the user-defined objects which are roles, subclasses of a particular object, the relationships they participate in, any objects contained in the relationships as well as their attributes and the data types associated with these attributes. We are also able to allow the end-user to traverse any generalization tree which may have been created through the use of the COMPONENT subclass. The parts of an object may be found by viewing all of those instances of COMPONENT aggregate.

Although we have described the Browser objects and relationships using the augmented ERD, the original method of designing such a tool would commence with the CDL specification. Work is presently begin undertaken which would allow for the initial step in the tool development process to be the graphical, as well as textual specification of the tool's objects, relationships, and operations. We now present the CDL specification of the Browser.

The object *Browser*.

**The Browser is an object. Initially, there are none; later, there is 1
Initially, the end-user may or may not be provided with a
Browser. If a Browser object does not exist in the database, then it
is the user's responsibility to create a Browser object.**

The object *View.*

A View is an object. Initially there are none; later, there may be many. Views represent a level in the hierarchy of information found in the CADIS database.

Catalog.

A Catalog_item is an object. Initially there are none; later, there may be many. Catalogs represent a sublevel of a View.

The object *Component.*

A Component is an object. Initially there are none; later, there may be many. Components represent the meta data of a Catalog.

The object *Instance.*

An Instance is an object. Initially there are none; later, there may be many. Instance represent the database entities for each component.

8.2.2 Relationship Identification

The next step in the design of the Browser is to identify the relationships into which the objects enter. The three types of relationships identified by Worley are: a primitive type; an aggregating type; and a user-defined type. Augmentation of the ERM has added a fourth type of relationship, the *class* type relationship. A primitive relationship is that relation which objects enter into with identifying properties which are considered attributes. An aggregating relationship is one in which an object possesses other objects. The third type of relationship is one in which two or more objects are related to each other via a verb. An example of this type is: Object (Designers) Verb (Workon) Object (Designs). A class relationship is one in which the verb is ISA. We make use of this fourth type in the description of the relationships between objects. The ERD of Figure 8.7 indicates these various relationships using the dotted notation, the diamond and the oval.

The relationships in the SI Browser are primarily the aggregating type of relationship. The relationships identified in the SI Browser are:

The Browser has a name and has a view. The Browser ISA Complex Entity

A view has a name, a description and many (m) catalog items. A view represents a System object.

A catalog has a name, a description, may be a view and may have many (m) components. A catalog represents subclasses and aggregate objects

A component has a name, a component description, may have many (m) catalogs, and may have many (m) instance.

An instance has a name, an instance description.

8.2.3 Attribute Identification

The attribute identification section merely includes a listing of the domains on which this simple type relationship is defined. The SI Browser ERD indicates that each object has a name and a description. The descriptions of each object may differ and are qualified to indicate the object to which it refers.

A name is a string

A description is text;

An instance description is one of [text, string, integer, boolean, code, real]

A component description is one of [properties, roles, relationships, objects, constraints]

8.2.4 Operations

Worley has defined four categories of operations which need to be specified by the tool developer. The categories of operations are transformation, query, pre-transformation validation, and time-invariant validation.

8.2.4.1 Transformation Operations

Transformation operations **to** are those operations which cause a state change. Such operations are those which would create views, add a category to a view, etc. Since the creation of such objects needs input from the CADIS data base, the tool developer will need to include CADIS manipulation operations within the code. The development of such code takes place after specifications are produced by the OReO Compiler. We will illustrate such data base operations in Section 8.2.5. The following represent the transformation operation for the SI Browser:

```
to CreateBrowser() returns Browser
  -- A new, empty Browser will be created.
  -- Only one Browser may exist at one time.
  -- Returns Nil if it fails for any reason.
  --  or a new Browser if it succeeds
to CreateView() returns View
  -- A new, empty View will be created.
  -- Returns Nil if it fails for any reason.
  --  or a new View if it succeeds
to CreateCatalog() returns Catalog
  -- A new, empty Catalog will be created.
  -- Returns Nil if it fails for any reason.
  --  or a new Catalog if it succeeds
to CreateComponent() returns Component
  -- A new, empty Component will be created.
  -- Returns Nil if it fails for any reason.
  --  or a new Component if it succeeds
to CreateInstance() returns Instance
  -- A new, empty Instance will be created.
  -- Returns Nil if it fails for any reason.
  --  or a new Instance if it succeeds
to BrowserNameSet(B: in out Browser; N: in name)
  -- Returns Nil if fails for any reason.
  -- If it succeeds, the name of B will be set to N and
  --  the updated B will be returned.
to ViewNameSet(V: in out View; N: in name)
  -- Returns Nil if fails for any reason.
  -- If it succeeds, the name of V will be set to N and
  --  the updated V will be returned.
to CatalogNameSet(C: in out Catalog; N: in name)
  -- Returns Nil if fails for any reason.
  -- If it succeeds, the name of C will be set to N and
  --  the updated C will be returned.
to ComponentNameSet(CP: in out Component; N: in name)
  -- Returns Nil if fails for any reason.
  -- If it succeeds, the name of CP will be set to N and
  --  the updated CP will be returned.
to InstanceNameSet(I: in out Instance; N: in name)
  -- Returns Nil if fails for any reason.
  -- If it succeeds, the name of I will be set to N and
  --  the updated I will be returned.
to ViewDescriptionSet(V: in out View; D: in description)
  -- Returns Nil if fails for any reason.
  -- If it succeeds, the description of V will be set to D and
  --  the updated V will be returned.
to CatalogDescriptionSet(C: in out Catalog; D: in description)
  -- Returns Nil if fails for any reason.
  -- If it succeeds, the description of C will be set to D and
  --  the updated C will be returned.
to ComponentDescritpionSet(CP: in out Component; N: in name)
  -- Returns Nil if fails for any reason.
  --  the updated CP will be returned.
to InstanceDescriptionSet(I: in out Instance; N: in name)
  -- Returns Nil if fails for any reason.
  -- If it succeeds, the description of I will be set to N and
  --  the updated I will be returned.
to AddViewToBrowser(B: in out Browser; V: in view)
  -- Returns Nil if fails for any reason.
  -- If it succeeds, V will be added to the collection of
  --  views contained in B, the updated B will be returned.
to AddCatalogToView(V: in out View; C: in catalog)
  -- Returns Nil if fails for any reason.
  -- If it succeeds, C will be added to the collection of
  --  catalogs contained in V, the updated V will be returned.
to AddViewToCatalog(C: in out Catalog; V: in view)
  -- Returns Nil if fails for any reason.
  -- If it succeeds, V will be added to the collection of
  --  views contained in C, the updated C will be returned.
to AddComponentToCatalog(C: in out Catalog; CP: in component)
  -- Returns Nil if fails for any reason.
  -- If it succeeds, CP will be added to the collection of
  --  components contained in C, the updated C will be returned.
to AddCatalogToComponent(CP: in out Component; C: in catalog)
  -- Returns Nil if fails for any reason.
  -- If it succeeds, C will be added to the collection of
  --  catalogs contained in CP, the updated CP will be returned.
to AddInstanceToComponent(CP: in out Component; I: in instance
  -- Returns Nil if fails for any reason.
  -- If it succeeds, I will be added to the collection of
  -- instances contained in CP, the updated CP will be returned.
to RemoveViewFromBrowser(B: in out browser; V: in view)
  -- Returns Nil if fails for any reason.
  -- If it succeeds, V will be removed from B
  --  the updated B will be returned.
to RemoveCatalogFromView(V: in out view; C: in catalog)
  -- Returns Nil if fails for any reason.
  -- If it succeeds, C will be removed from the collection of
  --  catalogs contained in V, the updated V will be returned.
to RemoveViewFromCatalog(C: in out catalog; V: in view)
  -- Returns Nil if fails for any reason.
  -- If it succeeds, V will be removed from C and
  --  the updated C will be returned.
to RemoveComponentFromCatalog(C: in out catalog; CP: in com-
ponent)
  -- Returns Nil if fails for any reason.
  -- If it succeeds, CP will be removed from the collection of
  --  components contained in C, the updated C will be returned.
to RemoveCatalogFromComponent(CP: in out component; C: in cata-
log)
  -- Returns Nil if fails for any reason.
  -- If it succeeds, C will be removed from CP
  --  and the updated CP will be returned.
to RemoveInstanceFromComponent(CP: in out component; I: in
instance)
  -- Returns Nil if fails for any reason.
  -- If it succeeds, I will be removed from the collection of
  -- instances contained in CP,  and the updated CP will be returned.
```

8.2.4.2 Query Operations

Query operations **qo** do not cause state change. They are often invoked to extract information. The information the end user of the SI Browser needs to extract is the name and description of each object.

```
qo BrowserName(B: in browser) returns name
  -- Returns Nil if fails for any reason,
  -- or the name of B if it succeeds.
qo ViewName(V: in View) returns name
  -- Returns Nil if fails for any reason,
-- or the name of V if it succeeds.
qo CatalogName(C: in Catalog) returns name
  -- Returns Nil if fails for any reason,
  -- or the name of C if it succeeds.
qo ComponentName (CP: in Component) returns name
  -- Returns Nil if fails for any reason,
  -- or the name of CP if it succeeds.
qo InstanceName(I: in instance) returns name
  -- Returns Nil if fails for any reason,
  -- or the name of I if it succeeds.
qo ViewDescription(V: in view) returns description
  -- Returns Nil if fails for any reason,
  -- or the description of V if it succeeds.
qo CatalogDescription(C: in catalog) returns description
  -- Returns Nil if fails for any reason,
  -- or the description of C if it succeeds.
qo ComponentDescription(CP: in component) returns description
  -- Returns Nil if fails for any reason,
  -- or the description of CP if it succeeds.
qo InstanceDescription(I: in instance) returns description
  -- Returns Nil if fails for any reason,
  -- or the description of I if it succeeds.
```

8.2.4.3 Pre-transformation Validation Operations

Pre-transformation validation operations **ptvo** are those operations which are invoked so as to ascertain whether or not the end user is able to enter a legal or desirable state. Since the transformation operations involve modification of the objects Browser, View, Catalog, Component, and Instance, the following routines are typical of pre-transformation operations required by the Worley methodology.

```
ptvo ExistsBrowser()
  -- Returns the Browser if it exists, Nil otherwise.
ptvo ExistsView()
  -- Returns the View if it exists, Nil otherwise.
ptvo ExistsCatalog()
  -- Returns the Catalog if it exists, Nil otherwise.
ptvo ExistsComponent()
  -- Returns the Component if it exists, Nil otherwise.
ptvo ExistsInstance()
  -- Returns the Instance if it exists, Nil otherwise.
```

8.2.4.4 Time-Invariant Transformation Operations

Time-invariant validation operations **tivo** are those operations which check the validity of the design model, particularly when it is about to be stored. The semantics of the SI Browser are such that there must be one and only one Browser View, that there cannot be any catalog, component, or instance item not participating in the aggregate relationship HAS, there must be a name associated with every object and there must be a description for each component and instance. The following routines are those which are necessary to validate a browser.

```
tivo ValidBrowser(B: in Browser)
  -- Returns B if it is a valid one, Nil otherwise.
  -- A valid browser is one that passes the following predicates.
tivo ExistsUniqueView(B: in Browser)
  -- Returns Nil if there is not one and only one view
  -- belonging to the Browser, otherwise it returns the
  -- browser.
tivo NoDanglingCatalog(B: in Browser)
  -- Returns Nil is there is no catalog not belonging to
  -- a View, otherwise it returns the browser.
tivo NoDanglingComponent(B: in Browser)
  -- Returns Nil is there is no component not belonging to
  -- a Catalog, otherwise it returns the browser.
tivo NoDanglingInstance(B: in Browser)
  -- Returns Nil is there is no instance not belonging to
  -- a Component, otherwise it returns the browser.
tivo UnnamedView(B: in Browser)
  -- Each View must have a name.
  -- Returns Nil if there is no View without a name, otherwise
  -- it returns a list of unnamed Views.
tivo UnnamedCatalog(B: in Browser)
  -- Each Catalog must have a name.
  -- Returns Nil if there is no Catalog without a name, otherwise
  -- it returns a list of unnamed Catalogs.
tivo UnnamedComponent(B: in Browser)
  -- Each View must have a name.
  -- Returns Nil if there is no Component without a name, otherwise
  -- it returns a list of unnamed Component.
tivo UnnamedInstance(B: in Browser)
  -- Each Instance must have a name.
  -- Returns Nil if there is no Instance without a name, otherwise
  -- it returns a list of unnamed Instance.
tivo UnDescribedComponent(B: in Browser)
  -- Each Component must have a description.
  -- Returns Nil if there is no Component without a description,
  otherwise
  -- it returns a list of Components without descriptions.
tivo UnDescribedInstance(B: in Browser)
  -- Each Instance must have a description.
  -- Returns Nil if there is no Instance without a description, otherwise
  -- it returns a list of Instance without descriptions.
```

8.2.5 Output

Having generated the SI tool specification, the OReO Compiler(Figure 3.8) produces output which consists of class definitions, operation specifications, and an augmented ERD which is then input to the DB_COMPILER.

CLASS DEFINITIONS

The class definitions incorporate the information extracted from the object and relation specification. They include the necessary data structures to represent an object, its attributes and its relations with other objects. The prototype implementation of the SARA/IDEAS system is in the T language. Worley hypothesized that many of the T objects would be able to be generated automatically from the specification. However, some portions of the code could not be generated automatically. The tool developer must then supply such code. Designers of the SARA/IDEAS system supplied all of the code. Figures 8.8 and 8.9 are example T code class definitions of the Browser and View objects.

```
(define (MakeBrowser)
 (let ((bname "")
         (bview '()))
      (object nil
              ((name self) bname)
              ((view self) bview)
              (((setter view) self val) (set bview val))
              (((setter name) self val) (set bname val))
              ((browser? self) t)
              ((show self)
                  (print bname (standard-output))
                  (walk (lambda(x) (show x)) bview)))))
```

Figure 8.8 Browser Class Definition

```
(define (MakeView)
 (let ((vname "")
       (description "")
       (vcatalog '()))
     (object nil
            ((name self) vname)
            ((describe self) description)
            ((catalog self) vcatalog)
            (((setter name) self val) (set bname val))
            (((setter describe) self val) (set description val))
            (((setter catalog) self val) (set vcatalog val))
            ((view? self) t)
            ((show self)
               (print vname (standard-output))
               (print description (standard-output))
               (walk (lambda(x)
                          (show x (standard-output))) vcatalog)))))
```

Figure 8.9 View Class Definition

OPERATION SPECIFICATIONS

The OReO compiler's output generated from the analysis of the operation section is provided to the implementor as a template. The implementor is responsible for filling in the routine in accordance with its specification. Simple, descriptive examples are provided for each of the types of operations. These routines are then input to the Integration Facility (Figure 3.8) in order to integrate the new tool into the CADOCS System. We give an example of each type of operation specification, transformation, query, pre-transformation validation, and time-invariant validation. Following each example specification will be code supplied by the tool implementor. The sample code includes the use of the manipulation operations defined in Chapter Five.

Transformation Operation

```
(define CatalogNameSet C N);; a transformation operation
;;;;C is a Catalog which is both input and output
;;;;N is a name which is input
;;;;Returns Nil if it fails for any reason
;;;;  or C if it succeeds
```

The operation which sets the name of a catalog object requires that the name be input not from the user, as is the case in the creation of objects within editors, but from information found in the CADIS database. Therefore, the tool implementor must write operations which access CADIS in order to retrieve this information. In the case of the *names* of catalog objects, the names of the subclasses of a system object, as well as all objects and relationships defined for each tool are needed to be retrieved from the CADIS database. The following T code illustrates the creation of the CATALOG objects related to the VIEW named CADIS. Recall that CADIS is considered a system object, therefore the tool developer starts by retrieving all system objects within CADIS.

```
;********************************
; The following function is invoked by the call "create-View "cadis"
;**********************************

;********************************
;This creates a View
;**********************************
(define (create-View namestring)
  (let ((temp
    (tmp1 '()))

    (set (name temp) namestring)
    (set (describe temp) (input-desc temp))
    (set  (catalog temp)
          (append
           (catalog temp)
           (map (lambda(x) (create-catalog
                         (string-downcase
                          (return-string x)) "system"))
           (set tmp1 (mksyslst namestring)))))
    (close-sysdb)
```

```
(set (catalog temp)
   (append
       (catalog temp)
       (map (lambda(x)
            (cond
             ((if-exists? x)
              (table-entry
               *catalogtable*
               (string->symbol (string-downcase (return-string x)))))
             (else
              (create-catalog
               (string-downcase (return-string x)) "else"))))
     (mksyscatlst namestring))))

(cond (( eq? (catalog temp) nil)
     (set (catalog temp)
             (append
              (catalog temp)
              (map
               (lambda(x)
                      (cond ((eq?
                               (sys-obj?
                               (string-downcase (return-string x))) T)
                              (create-catalog
                               (string-downcase
                                (return-string x)) "system"))
                     (else
                                (cond ((if-exists? x)
                                        (table-entry
                                         *catalogtable*
                                         (string->symbol
                                             (string-downcase
                                              (return-string x)))))
                                     (else
                                      (create-catalog
                                       (string-downcase
                                        (return-string x)) "user"))))))
              (mkcatloglst (delch namestring) tmp1))))))
  temp))

;************************************
; This creates a Catalog
;**********************************

(define (create-Catalog  catalogtype origin)
 (let ((.....

    (set (name temp ) catalogtype)
    (set (describe temp) (input-desc temp))
```

```
           (cond ((string-equal? "system" origin)
                 (block (trollmsg "close;")
                        (set *open* nil)
                        (set (view temp)
                              (create-view  catalogtype))
                        (close-sysdb)))

                ((string-equal? "else" origin)
                 (trollmsg
                  (string-append "open "
                                 (car (return-Dbstring catalogtype)) ";"))
                 (set (component temp)
                       (map (lambda (x)
                                   (create-component
                                    catalogtype x)) (mkcomplst
                                   (return-string catalogtype))))
                 (set *open* nil)
                 (trollmsg "close;"))

                (else (set (component temp)
                             (map (lambda(x) (create-component catalogtype x))
                                (mkcomplst (return-string catalogtype))))))
                 temp))
```

Query Operation

```
(define ComponentName CP N);; a query operation
;;;;CP is a component which is input
;;;;N is a name which is output
;;;;Returns Nil if it fails for any reason,
;;;;   or N if it succeeds.
```

The query operations defined by Worley, merely query the active data structure and are not meant to query the database. We have seen that the tool implementor of the SI Browser queries the database when performing the transformation operations. Therefore, the following portion of code merely returns the name of a given component, in other words, the method associated with the T object MakeComponent and referred to as *name* is invoked.

```
(define (ComponentName Component namestring))
        (set namestring (name Component)))
```

Pre-transformation Validation Operation

```
                define ExistsInstance CP);;a pre-transformation validation operation
                ;;;;CP is a component which is input
                ;;;; Returns True if an instance for CP exists
                ;;;;  or Nil otherwise


        (define (existsinstance component)
                (cond (findins (name component))
                    else nil))
```

Time-invariant Validation Operation

```
                (define NoDanglingCatalog B);; a time-invariant validation operation
                ;;;; B is a Browser which is input
                ;;;; Returns True if there is no catalog without a view
                ;;;; otherwise it returns a list of those catalogs failing the test.


        (define (nodanglingcatalog browser)
                 (cond (searchforcatalog browser)
                     else nil))
```

AUGMENTED ERD

The final output from the OReO Compiler would be similar to the ERD of Figure 8.7. This augmented ERD would then be used to generate a database schema for the SI Browser. We have shown in Chapter Six the generation of a schema for the SM Editor. The textual input generated by this author follows the format of Figure 6.8. The SI Browser schema differs only in that its database consists of more traditional database entries, those that are informational and not structural. In fact, the database entries for the Browser can be regarded as information similar to that found in a catalog file of a reference library. The actual entries represent information about the structure and contents of the SARA/IDEAS database system, CADIS. The SI Browser can be regenerated by accessing the complex entity Browser, according to the algorithms given in Chapter Seven. The following code shows the tool implementor needed to create so that if a Browser already exists then the tool invokes a load operation - "loadbrowser namestring". This load operation need only find the Browser by querying the database *(findobj)* and call on the database operation *retrieve*.

```
;*******************************
; This function creates a Browser
;*********************************
(define (create-Browser namestring )
 (lset temp (MakeBrowser))
 (set (name temp) namestring)
 (dbcontrol "open" "/u/landis/cadis/tools")
 (cond ((sysobjx namestring)
        (loadbrowser namestring))
      (else
          (set (view temp)  (create-View "cadis")))))
;*******************************
; This function loads all of the objects of the browser
; it calls a findobj and retrieve function supplied by the DB_KERNEL
;*************************************
(define (loadbrowser namestring)
        (set tmp (findobj "Browser" "name" "=" namestring))
        (retrieve tmp))
```

The outputs of the OReO Compiler which have been described, the routines and the database schema are a portion of the input to the Integration Facility (Figure 3.8). The database schema is then used by the subprocessor, DB_INTEGRATOR, within this facility in order to integrate this tool's schema into the CADIS database. Chapter Seven has described what that process would involve.

8.2.6 Generation of Screen Graphics

Although the focus of this chapter is on the conceptual specification phase of the SI Browser tool, we presented the scenario of the tool's use using pictures of menus which we envisioned would display the information. These menus are generated during the creation of the Browser objects which were described in detail in section 8.2. The tool implementor is able to associate with each Browser object, which is to be displayed, a menu containing the appropriate items. The SARA/IDEAS tool implementor is able to create such menus through the use of graphic primitives which are available within the SARA/IDEAS KERNEL. For our purposes, we needed to generate the following code.

```
(herald menu)

(define (MakeMenu)
 (let* ((MenuTable (make-table))
        (MenuCount 0)
        (RectangleObject nil)
        (TextItems nil))
   (object
    nil
    ((print self port)
     (format port
             "#{Menu ~d}"
             (object-hash self)))
    ((rectangle self) RectangleObject)

    (((setter Rectangle) self v)
     (set RectangleObject v))

    ((selectitem self point)
     (catch found
        (walk (lambda (x)
                (if (picked? x point)
                   (found (table-entry MenuTable x))))
            TextItems)
        nil))

    ((AddSelection self view Text)
     (let ((item (MakeText (init-TextShp (init-point 5
                                                   (fx+ (fx* MenuCount
                                                              15)
                                                        5))
```

```
                                        (init-point 0 0)
                                        Text)
                            self)))

                   (set (screen item) (screen self))
                   (set TextItems (cons item TextItems))
                   (set MenuCount (fx+ MenuCount 1))
                   (set (table-entry MenuTable item) view)
                   (->rtshape self)
                   (set-bbox self)
                   (set (rtshape RectangleObject)
                      (init-RectShp (coord self)
                            150
                            (fx+ (fx* MenuCount 15) 10)))))

               ((SelectByName self name)
                (catch found
                   (walk (lambda (x)
                         (if (string-equal? name (TextShp-txt (rtshape self)))
                            (found (table-entry MenuTable x))))
                      TextItems) nil)))))
            (define-operation (selectitem self))
            (define-operation (SelectByName self))
            (define-operation (AddSelection self))
            (define-settable-operation (rectangle self))
```

Figure 8.10 Menu Object Code

The above code defined a *Menu* object. Although the browser tool was originally implemented on a system which had little graphical capabilities, we later transported the tool to the current system used for the SARA/IDEAS system, a SUN workstation. In order to take advantage of the window capabilities for use with the Browser tool, we went back through the original code for each object found within the Browser and added code which would associate with each VIEW, CATALOG, COMPONENT and INSTANCE, an individual *menu-object*. We did this by augmenting a "Menu" object with the Browser object. This "Menu" is the object created by the call to MakeMenu found in the above code. Then for each item found in an individual object's list, we invoked AddSelection with the respective name. The ability to select a menu item from a window with the mouse is provided by the graphic primitives.

Recall that the selection of a particular menu item by the end-user causes another menu associated with the selected item to appear. The mechanisms to display and return selections are provided by these same graphic primitives.

8.3 Summary

This chapter has defined a new SARA/IDEAS tool and has described it using both the augmented ERD and the Concept Definition Language. This definition covered the conceptual phase of the Worley methodology. The final input to the Integration Facility (Figure 3.8) also includes output from the other three phases, the semantic, lexical, and physical. Details of the mechanics involved in the completion of these phases can be found in [Worl86] , Chapters Five, Six and Seven.

The purpose of this exercise was twofold. The primary purpose was to develop a tool which the SARA/IDEAS tool implementor could use to browse through the existing database. This browser precludes any knowledge of the structure of the CADIS database. The presentation of the design of this type of tool included a description of the conceptualization of the objects which would be manipulated and viewed by the end-user. The actual tool presents the CADIS database by representing the traversal of the CADIS tree presented in Figure 7.5. Use of this tool by another tool implementor would provide information as to existing system objects and tool objects which the tool implementor might find useful when designing a new tool.

The second purpose of this exercise was to demonstrate the manner in which a different type of tool could be added to the existing system. Portions of code needed to realize such a tool were presented. Although the focus of Section 8.2 was on the steps taken to specify the tool, in terms of its objects, relationships, and operations, there were many occasions when the need arose to use the augments developed by this author for use within the conceptual design phase. Such occurences were the classification of the Browser as a *complex entity* so as to provide easy database access to the Browser's information, use of the *dotted aggregate notation* so as to easily represent the inclusion of views, catalogs, and components within the Browser, etc. Code produced by the tool implementor also used those portions of this author's work which defined data manipulation operations for the augmented ERM (Chapter Five). Another contribution of this work, related to the conceptual specification phase is the automated generation of a database schema associated with the definition. We did not, within this chapter, show the structure of the Browser's database, since it would be similar to that database schema shown in Chapter Six.

CHAPTER 9

Conclusions

As computer-based systems become more complex and assume greater responsibilities, the need for intelligent data management becomes increasingly important. An area in which the growth of computer use is rapidly growing is that of computer-aided design. Many tools have been proposed and built that support the design activity. When several of these tools are gathered together, the aggregation is called a Computer-Aided Design (CAD) System.

We have seen that often, CAD systems are constructed from separately developed computer programs. These programs are frequently written in different programming languages to run on different computer systems; they offer different user interfaces; and they store the results of their analysis or simulation in a way that precludes information sharing between programs. Tool developers typically concentrate their efforts on the design capability of a tool and are unable to justify the expense (time and money) of providing a state-of-the-art support system. The resulting CAD systems are only nominally integrated and are difficult to use at best. This dissertation and a companion dissertation [Worl86] have provided both a methodology to develop extensible, integrated CAD systems, and an underlying nucleus which provides support for the tool building methodology. This particular dissertation focused on the backend of this system, the database design and support.

As a result of the investigation into the problem domain associated with the design and development of data management support of the computer-aided design process, a major contribution has been the provision of a mechanism by which the tool builder can still focus on the design capability, but at the same time specify components of the underlying database. This mechanism captures the semantics of the tool's output so that information sharing can be promoted.

The fundamental approach is to decouple the database functions from the tool functions. This is realized by providing, within the tool building methodology, the means to represent the tool's data in a semantically meaningful way so that this data may be used by other tool builders.

The mechanism is the Data Base Kernel. This DB_KERNEL supports a user's conceptual model generated during the conceptual definition phase. Besides defining operations the tool performs on objects, the tool builder defines in somewhat greater detail, the structure and information associated with each tool object. This description is used to generate the database specifications allowing for automation of the database facility.

The CADIS System interacts with this DB_KERNEL in order to provide the data-base-specific functions applicable to the CAD system under development. This CADIS System regards the tools as objects which themselves contain objects and therefore the System must maintain inter-tool and inter-system relationships between such objects. The provision of generic system objects, which contain the complicated methods to retrieve and modify complex objects removes the tool designer from the data base arena. A Data Base Integrator (DB_INTEGRATOR) is necessary to bring new tools into the system.

The SARA/IDEAS system has proven to be a fertile testbed for this data base concept. Heretofore, the SARA/IDEAS system stored complete designs in a file, and retrieval was at this gross level of granularity. The semantics of the internal components of the design was known only to the specific tool which created the design. Obtaining knowledge about, or being able to use components deep within the design was an impossibility. Since, the design tools are varied and at different levels of complexity, identification of objects and relationships within these tools promotes the access and retrieval of finer units of granularity. The interaction we see among the SARA/IDEAS tools (SM-GMB) promotes the need for some type of inter-tool relationships.

9.1 Contributions

Design Data Base System

We have developed a design data base system based on a four level architecture, in order to meet the proposed goal of implementing a design data base system within the SARA/IDEAS environment. This architecture provides for view representations of the tools (Tool level) as well as the data base conceptual representation (System + Kernel level). The separation of the traditional conceptual level into two levels enables the non database-expert (we hypothesize this to be the tool builder) to concentrate on those aspects of tool-building with which she/he is an expert. Database experts supply the System objects, which can be later integrated with the tool objects, to provide a completely functional data management system (CADIS). The physical level of the architecture is a commercial relational system which has been modified so as to provide for the mamagement of the complex objects and relatioships. The benefits derived from using a relational system are that such systems are proven as well as strongly supported.

Data Base Management Kernel

The major focus of this work was on research into desired properties of a DB_KERNEL and the development of such a DB_KERNEL. Included in this DB_KERNEL were the procedures to create and manipulate the view defined by a tool designer. It was important to us that the tool designer does not have to be a database expert in order to produce an underlying schema. Of equal importance was the support for design knowledge primitives within the DB_KERNEL so that the data base may be more powerfully extended than otherwise. Such a DB_KERNEL not only provides for the management of design data, but also provides the basis for use of intelligent design tools in capturing design expertise and design histories. To meet these goals, the DB_KERNEL was designed to support an extended data model. We defined (Chapter Five) such extensions necessary for the support of a design environnment, the augmented ERM.

Augmented ERM

The augmented ERM provided us with an extensible and flexible data structure so that we could provide required data management support for a design environment, as well as provide the necessary mechanisms for the acquisition and management of design knowledge. Extensibility, the ability to easily modify the model, is achieved by being able to easily add new entities and relationships. If the system is represented as an object, then tools are linked to this object via previously defined relationships. Flexibility, the capability to represent information in more than one way, is facilitated through the use of generic objects and the introduction of role sets.

Data Base Compiler

We have met the goal of providing extensible data definition facilities for use by both design tools and the CAD system designer. A DB_COMPILER was implemented in the T dialect of LISP for use in the SARA/IDEAS system. This DB_COMPILER produced standard Troll relations for objects and relationships defined for the SM Editor tool. The DB_COMPILER also modified the system's meta data; relational tables which are used to process classes and aggregates. The completion of such a compiler allowed us to meet the stated goal of integrating existing tools, e.g. Troll, into the system.

Translation into RM/T relations

We have provided a set of mapping rules, which allowed us to develop an algorithm to translate the augmented ERD structure into an underlying extended relational model, the RM/T. The RM/T allowed us to maintain integrity constraints on molecular entities as well as atomic entities. This provision, we found, is is an essential component in a design environment which is composed of complex molecular objects.

SARA/IDEAS Browser

A data intensive tool, the SARA/IDEAS (SI) Browser was designed using the defined tool building methodology. Through the use of menus the structure of the entire CADIS database can be viewed. The development of this tool illustrated the power and flexibility of the data base design.

During development of this tool, the process by which a designer arrives at a choice of objects in and their respective relationships was examined. This examination resulted in a description (Section 8.2 - Object Identification) of the process by which one might determine appropriate objects which a tool would manipulate.

Additionally, the design of this tool combined the use of CADIS database primitives, as well as primitives provided by the User Interface Management System (UIMS). These primitives included those that allowed graphical objects, Menus, to be associated with the Browser objects.

9.2 Future Research

This research has produced a model and data base support system on which a robust integrated CAD system can be built. We envision ths SARA/IDEAS system to be an evolving system, supported by a state-of-the-art development environment. There are still many components and areas of research associated with the database aspect of the environment which need to be developed.

Multiple Underlying Models

The philosophy behind the tool building methodology was that the resulting integrated system could be built on any type of data base system. This research chose to use an extended relational system as the underlying support system. However, recent advances in database technology [Maie86] have produced systems which much more effectively support an entity-relationship model . The overhead incurred in the mapping from the augmented ERM into a relational scheme might be considerably reduced if CADIS could be supported by this type of database.

Graphical User Interface

This research produced a rich semantic model which can easily support a graphical, interactive user facility. Work has already begun on developing an ERD editor which can be used by a tool developer to express the semantics of the tool's objects, relationships, properties and constraints in lieu of using the Conceptual Definition Language provided by Worley.

Related to the provision of a graphical ERD editor, would be the creation of a more 'intelligent' tool which would be able to used by a designer to help in the generation of alternate ERDs. This tool might be provided with keyword descriptions of a tools's specification and produce high level objects corresponding to such specification. The generation of appropriate objects is perhaps the most difficult task presented to a tool designer.

Tool Browsers

The research produced a design of a system browser. However, tool browsers, for use while creating a design would enhance the development environment by allowing designers to browse through portions of their designs while in the process of creating the design. The creation of individual tool browsers is actually provided through the subdialogue mechanism provided by Worley.

Building Block Library

As of yet, there exists no library of previously designed models for use by the designer. The CADIS database architecture should be able to easily support the addition of such a library. It is hypothesized that the Worley methodology could be applied to the design of such a tool. The tool would be a *Librarian* which would enter the building block into a system library. The Librarian's subdialogue would provide appropriate browsing capabilities for the user.

Update Propagation

Incorporate mechanisms to provide for update propagation among the different representations of design. A change in the control graph would automatically change the data graph. This could be provided using the *instantiation* object as a vehicle through which such constraints could be maintained.

Multi-user Environment

The work done for this research concentrated on the provision of a data model and the resultant data base, for use in the development of design tools for the SARA/IDEAS system. However, at this time the SARA/IDEAS system supports only a single user environment. It is envisioned that this system will eventually support a multi-user environment. To support such a system, the data base facilities must support concurrency in the use of objects and relationships. Future research into the definition of appropriate system objects which would be able to provide such concurrent usage of the data model is necessary.

APPENDIX A

SM_Editor Code

The following T code represents the object definitions for the SM_Editor

```
;****************************************
;   OBJECT MODULE
;****************************************
(define (MakeModule)
        (let ((mname "")
              (mparent nil)
              (mchildren nil)
            (msockets nil)
              (interconn nil))
              (object nil
                      (( name self) mname)
                      (( parent self) mparent)
                      (( children self) mchildren)
                      (( sockets self) msockets)
                      (( interconnections self) interconn)
                      (((setter  name) self val) (set mname val))
                      (((setter  parent) self val) (set mparent val))
                      (((setter  children ) self val) (set mchildren val))
                      (((setter  sockets ) self val) (set msockets val))
                      (((setter  interconnections) self val)(set interconn val))
                      ((module? self)t)
                      )))
(define-settable-operation ( name x))
(define-settable-operation ( parent x))
(define-settable-operation( children x))
(define-settable-operation ( sockets x))
(define-settable-operation ( interconnections x))
(define-predicate module?)
;****************************************
;  OBJECT SOCKET
;****************************************
(define (MakeSocket)
        (let ((sname "")
              (sparent nil)
              (insideC nil)
              (outsideC nil))
             (object nil
                     ((name self) sname)
                     ((parent self) sparent)
```

```
                    ((socketInside self) insideC)
                    ((socketOutside self) outsideC)
                    (((setter parent) self val) (set sparent val))
                    (((setter socketInside) self val) (set insideC val))
                    (((setter socketOutside) self val) (set outsideC val))
                    (((setter name) self val) (set sname val ))
                    ((socket? self) t)
                    )
          )
 )
(define-settable-operation (socketInside s))
(define-settable-operation (socketOutside s))
(define-predicate socket?)
;****************************************
;   OBJECT INTERCONNECTION
;****************************************
(define (MakeInterconnection)
   (let ((iname "")
     (iparent nil)
     (isockets nil))
    (object nil
            ((name self) iname)
            ((parent self) iparent)
            ((sockets self) isockets)
            (((setter name) self val) (set iname val))
            (((setter parent) self val) (set iparent val))
            (((setter sockets) self val) (set isockets val))
            ((interconnection? self) t))))
(define-predicate interconnection?)
```

The folloowing T code represents the semantics of the SM_Editor

```
;*********************************************
;    CREATES THE UNIVERSE
;*********************************************
(define (CreateUniverse nameString)
          (set *UNIVERSE* (MakeModule))
             (set (CurrentModule) *UNIVERSE*)
                   (if (string-empty? nameString)
                  (set ( name (CurrentModule)) "universe")
                  (set ( name (CurrentModule)) nameString)))
;**************************************************
;   CREATES A MODULE
;**************************************************
(define ( CreateModules module nameStringlist)
 (walk (lambda(nameString)
          (lset temp (MakeModule))
          (set ( name temp) nameString)
          (set ( parent temp) module  )
          (set ( children module)
           (append ( children module) (list  temp))))
 nameStringlist))

;*********************************************
;    CREATES A SOCKET
;*********************************************
(define (CreateSockets module nameStringlist)
 (walk (lambda(nameString)
     (lset temp (MakeSocket))
        (set (name temp) nameString)
        (set (parent temp) module)
        (set ( sockets module)
           (append ( sockets module) (list  temp)
                                        )))nameStringlist))
;*******************************************
;     CREATES AN INTERCONNECTION
;*******************************************
(define (CreateInterconnects  nameString  sockets1  sockets2)
        (lset temp (MakeInterconnection))
        (set (name temp) nameString)
        (set (sockets temp) (cons sockets1 sockets2))
        (set tmp1 (parent sockets1))
        (set tmp2 (parent sockets2))
        (cond ((eq? (parent tmp1) (parent tmp2))
                (set (socketOutside sockets1) temp)
                (set (socketOutside sockets2) temp)
                   (set (parent temp) (parent tmp1))
                   (set (interconnections (parent tmp1))
                         (append (interconnections (parent tmp1)) (list temp))))
            ((eq? tmp1 (parent tmp2))
             (set (socketInside sockets1) temp)
             (set (socketOutside sockets2) temp)
             (set (parent temp) tmp1)
             (set (interconnections tmp1)
                   (append (interconnections tmp1) (list temp))))
            ((eq? (parent tmp1) tmp2)
             (set (socketOutside sockets1) temp)
             (set (socketInside sockets2) temp)
             (set (parent temp) tmp2)
             (set (interconnections tmp2)
                   (append (interconnections tmp2) (list temp))))))


;******************************************
;    DELETES A MODULE
;******************************************
(define (DeleteModules module)
        (if (not (null? (sockets module)))
           (map (lambda(x) (DeleteSockets x)) (sockets module)))
           (set ( children ( parent module))
               (delq module ( children (parent module)))))
;****************************************
```

```
;     DELETES A SOCKET
;*****************************************
(define (DeleteSockets socket)
        (if (not (null? (socketInside socket) ))
           (DeleteInterconnects  (socketInside socket)))
        (if (not (null? (socketOutside socket) ))
           (DeleteInterconnects  (socketOutside socket)))
        (set ( sockets (parent socket) ) (delq socket ( sockets (parent socket)))))
;*****************************************
;     DELETES AN INTERCONNECTION
;*****************************************
(define (DeleteInterconnects interconnect )
         (walk (lambda(x)
                    (walk (lambda(y) (cond ((eq? interconnect
                                                  (socketInside y))
                                (set (socketInside y) nil))
                               ((eq? interconnect (socketOutside y))
                                (set (socketOutside y) nil))))
                          ( sockets x))) ( children (parent interconnect)))
        (set ( interconnections (parent interconnect)
                             ) (delq interconnect ( interconnections (parent
                                                              interconnect)
                                                              ))))
```

APPENDIX B
GMB_Editor Code

The following T code defines the GMB objects

```
;;;*****************
;;;* The GMB object *
;;;*****************
(define (MakeGMB)
 (let
   ((the-cg nil)
    (the-dg nil)
    (smmodule nil)                 ; an SM module
    (cg-dg nil))                   ; the mapping
  (object nil
         ((print self port)
          (format port "{SemanticGMB ~d}~%"
                 (object-hash self)))
         ((name self) (string->symbol "GMB"))
         ((gmb? self) '#T)
         ((cg self) the-cg)
         ((dg self) the-dg)
         ((cg-dgMapping self) cg-dg)
         ((module self) smmodule)
         (((setter cg) self val)
          (set the-cg val))
         (((setter dg) self val)
          (set the-dg val))
         (((setter cg-dgMapping) self val);; may want to change the rep.
          (set cg-dg  val));; of mapping to objects later
         (((setter module) self val)
          (set smmodule val))
         ((add-object self x)
          ;; should set the parent here
          (cond ((controlgraph? x)
                 (set the-cg x)
                 (set (parent x) self))
                ((datagraph? x)
                 (set the-dg x)
                 (set (parent x) self))
                (else   (error-mess "object cannot be added to GMB"))))
         ((del-object self x)
          (cond ((controlgraph? x) (set the-cg nil))
                ((datagraph? x) (set the-dg nil))
                (else   (error-mess "object is not a part of GMB")))
          (del-child self x))
         ((delete self)
            (del-object (parent self) self)
            (erase self) )
           ((store self port)
            (format port
                    "~%(let ((ThisGMB
                             (AddGMBfromDB ThisModule)))~%")
            (store (cg self) port)
            (store (dg self) port)
            (format port "(AddMappingFromDB ThisGMB '~s)~%"
                    (map (lambda (x) (list (name (car x))
                                           (name (last x))))
                         (cg-dgmapping self)))
            (format port "ThisGMB)~%")))))

;;****************************************
;;*  The Control Graph Semantic Object  *
;;****************************************

(define (MakeControlGraph)
  (let
     ((cnodes nil)                        ; control nodes
      (carcs nil)                 ; control arcs
      (cgmb nil)
      )
    (object
     nil
     ((print self port)
      (format port "{SemanticControlGraph ~d}~%"
              (object-hash self)))
     ((name self) (string->symbol "CG"))
     ((ControlGraph? self) '#T)
     ((controlnodes self) cnodes)
     ((controlarcs self) carcs)
     ((gmb self) cgmb)
     ((module self) (module (gmb self)))
     (((setter controlnodes) self val)
      (set cnodes val))
     (((setter controlarcs) self val)
      (set carcs val))
     (((setter gmb) self val)
      (set cgmb val))
     ;; translogic takes an logic expression which uses the
     ;; normal logic operators and tranlate it into one using
     ;; operators of the form 'apply-OP', where OP is the
     ;; corresponding logic operator
     ((translogic self logic)
      (COND ((null? logic) nil)
            ((list? (car logic))
             (cons (translogic self (car logic))
                   (translogic self (cdr logic))))
            (else
             (case (car logic)
                    ((and)
                     (cons apply-and
                           (translogic self (cdr logic))))
                    ((or)
                     (cons apply-or
                           (translogic self (cdr logic))))
                    ((>)
                     (cons apply-prio
                           (translogic self (cdr logic))))
                    (else
                     ;; should take care of the case when
                     ;; logic is neither input or
                     ;; output arc of node,  this is signalled
                     ;; by get-arc returning nil
                     (cons (name->container self
                                            (string->symbol
                                             (string-downcase
                                              (symbol->string
                                               (car logic)))))
                           ;; need to check what obj-assq does
                           (translogic self (cdr logic)))))))
((add-object self x)
 (cond ((ControlNode? x)
        (set cnodes (cons x cnodes))
        (set (parent x) self))
       ((ControlArc? x)
        (set carcs (cons x carcs))
        (set (parent x) self))
       (else
        (error-mess "object cannot be added to Control Graph"))))
((del-object self x)
 (cond ((controlnode? x)
        (set cnodes (delq x cnodes)))
       ((controlarc? x)
        (set carcs  (delq x carcs)))
       (else
      (error-mess "object not a part of Control Graph")))
 (del-child self x))
((delete self)
 (del-object (parent self) self)
 (erase self))
((store self port)
 (let ((logiclist nil))
     (format port
           "(let ((ThisCG (AddCGFromDB ThisGMB)))~%")
     (set logiclist (map (lambda (x) (list
                                      (name  x)
                                      (reverse-translogic (inputLogic x))
                                      (reverse-translogic (outputLogic x))))
                     (controlnodes self)))
     (walk (lambda (x) (store x port)) (controlnodes self))
     (walk (lambda (x) (store x port)) (controlarcs self))
```

```
        (format port
                "(AddLogicFromDB ThisCG '~s)"
                logiclist)
        (format port
                "ThisCG)~%"))))))

;;****************************************
;;*  The Data Graph Object           *
;;****************************************

(define (MakeDataGraph)
 (let
   ((dprocs nil)                  ; data processors
    (darcs nil)              ; data arcs
    (dsets nil)              ; datasets
  (object nil
          ((print self port)
           (format port "{SemanticDataGraph ~d}~%"
                   (object-hash self)))
          ((name self) (string->symbol "DG"))
          ((datagraph? self) '#T)
          ((dataprocs self) dprocs)
          ((dataarcs self) darcs)
          ((datasets self) dsets)
          ((gmb self) cgmb)
          ((module self) (module (gmb self)))
          (((setter dataprocs) self val)
           (set dprocs val))
          (((setter dataarcs) self val)
           (set darcs val))
          (((setter datasets) self val)
           (set dsets val))
          (((setter gmb) self val)
           (set cgmb val))
          ((add-object self x)
           (cond ((dataset? x)
                  (set dsets (cons x dsets))
                  (set (parent x) self))
                 ((dataarc? x)
                  (set darcs (cons x darcs))
                  (set (parent x) self))
                 ((dataprocessor? x) (set dprocs (cons x dprocs)))
                 (else (error-mess "object cannot be added to Data Graph"))))
          ((del-object self x)
           (cond ((dataset? x)
                  (set dsets (delq x dsets)))
                 ((dataarc? x)
                  (set darcs (delq x darcs)))
                 ((dataprocessor? x)
                  (set dprocs (delq x dprocs)))
                 (else (error-mess "object not a part of Data Graph")))
           (del-child self x))
          ((delete self)
        (del-object (parent self) self)
        (erase self))
          ((store self port)
           (format port
                   "(let ((ThisDg (AddDGFromDB ThisGMB)))~%")
           (walk (lambda (x) (store x port)) (dataprocs self))
           (walk (lambda (x) (store x port)) (datasets self))
           (walk (lambda (x) (store x port)) (dataarcs self))
           (format port
                   "ThisDG)~%")))))

;;;***************************
;;;* The Control Node object *
;;;***************************
(define (MakeControlNode)
 ;; static information
 (let ((cname nil)                    ; name
        (iarcs nil)                   ; input arcs
        (oarcs nil)                   ; output arcs
        (ilogic nil)                  ; input Logic
        (ologic nil)                  ; output Logic
        (cap 1)                       ; capacity
        ;; dynamic information,
        ;; subject to change during simulation
        (custmr 0)                    ; current # of tokens
        (visits 0)                    ; number of times node has been executed
        (act? nil)                    ; currently active or not
        (queue nil)                   ; queueing
        (bpoint (MakeNBrk))           ; breakpoint stuff
        )
   (object nil
          ((print self port)
           (format port "{SemanticControlNode ~d ~a}~%"
                   (object-hash self)
                   (symbol->string (name self))))
          ((controlNode? self) '#T)
          ((name self) cname)
          ((gmb self) (parent (parent self)))
          ((inputArcs self) iarcs)
          ((outputArcs self) oarcs)
          ((inputLogic self) ilogic)
          ((outputLogic self) ologic)
          ((capacity self) cap)
          ((customers self) custmr)
          ((numvisits self) visits)
          ((active? self) act?)
          ((queueing self) queue)
          ((breakpoint self) bpoint)
          (((setter name) self val)
           (set cname val))
          (((setter inputArcs) self val)
 (set iarcs val))
(((setter outputArcs) self val)
 (set oarcs val))
(((setter inputLogic) self val)
 (set ilogic val))
(((setter outputLogic) self val)
 (set ologic val))
(((setter capacity) self val)
 (set cap val))
(((setter customers) self val)
 (set custmr val))
(((setter numvisits) self val)
 (set visits val))
(((setter active?) self val)
 (set act? val))
(((setter queueing) self val)
 (set queue val))
(((setter breakpoint) self  val)
 (set bpoint val))
((delete self)
 (del-object (parent self) self)
 (walk (lambda (x ) (del-head x self)) iarcs)
 (walk (lambda (x ) (del-tail x self)) oarcs)
 (erase self))
((add-object self arc)
 (cond ((controlarc? arc)
         (cond ((memq self (headset arc))
                 (set iarcs (cons arc iarcs)))
                (else
                 (set oarcs (cons arc oarcs))) ))
        (else (error-mess "not a control arc"))))
((del-object self arc)
 (cond ((controlarc? arc)
         (set iarcs (delq arc iarcs))
         (set oarcs (delq arc oarcs)))
        (else (error-mess "not a Control Arc"))))
((addlogic self logic-exp in/out)
 (let ((Cgraph (parent self)))
   (case in/out
          ((in)
           (set (InputLogic self) logic-exp))
          ((out)
           (set (OutputLogic self) logic-exp)))))
((checkbrk self tag)
 (break? bpoint tag visits))
((store self port)
 (format port
          "(AddCnodeFromDB ThisCG ~s ~d  ~d ~d ~d ~d)~%"
          (symbol->string (name self))
          (capacity self)
          ;; this could be done with (store ... point port)
          (point-x (cplxshp-loc (abshape self)))
```

```
                  (point-y (cplxshp-loc (abshape self)))
                  (circshp-rad (abshape (circle self)))
                  (Pscale self)))))
;;;*************************
;;;* The Control Arc Object *
;;;*************************
(define (MakeControlArc)
 ;; static information
 (let ((aname nil)                  ; name
       (hset nil)                   ; headset
       (tset nil)                   ; tailset
       (itoks 0)                    ; initial # tokens
       ;; dynamic information, subject to change during simulation
       (als nil)                    ; alias
       (toks 0)                     ; current total # tokens
       (ltoks 0)                    ; current local # tokens
       (bpoint (MakeABrk))          ; breakpoint
       (queue nil)                  ; queueing info
       )
  (object
   nil
   ((print self port)
    (format port "{SemanticControlArc ~d ~a}~%"
            (object-hash self)
            (symbol->string (name self))))
   ((name self) aname)
   ((gmb self) (parent (parent self)))
   ((headset self) hset)
   ((alias self) als)
   ((tailset self) tset)
   ((initialTokens self) itoks)
   ((localtokens self) ltoks)
   ((tokens self) toks)
   ((breakpoint self) bpoint)
   ((queueing self) queue)
   ((add-tail self x)
    (cond ((or (socket? x) (controlnode? x))
           (set tset (cons x tset)))
          (else (error-mess "picked TAIL not a Control Node"))))
   ((add-head self x)
    (cond ((or (socket? x) (controlnode? x))
           (set hset (cons x hset)))
          (else (error-mess "picked HEAD not a Control Node"))))
   ((del-tail self x)
    (cond ((or (socket? x) (controlnode? x))
           (if (not (null? (assq x (tailpairs self))))
               (erase (cadr (assq x (tailpairs self)))))
           (set tset (delq x tset))
           (if (null? tset) (delete self)))
          (else (error-mess "picked TAIL not a Control Node"))))
   ((del-head self x)
    (cond ((or (socket? x) (controlnode? x))
          (if (not (null? (assq x (headpairs self))))
                (erase (cadr (assq x (headpairs self)))))
          (set hset (delq x hset))
          (if (null? hset) (delete self)))
         (else (error-mess "picked HEAD not a Control Node"))))
(((setter name) self val)
 (set aname val))
(((setter headset) self val)
 (set hset val))
(((setter tailset) self val)
 (set tset val))
(((setter initialtokens) self val)
 (set itoks val))
(((setter localtokens) self val)
 (set ltoks val))
(((setter tokens) self val)
 (set toks val))
(((setter breakpoint) self val)
 (set bpoint val))
(((setter queueing) self val)
 (set queue val))
(((setter alias) self val)
 (set als val))
((delete self)
 (del-object (parent self) self)
 (walk (lambda (X) (disconnect self x)) tset)
 (walk (lambda (X) (disconnect self x)) hset)
 (erase self))
                                          ; end traverse
;;
;;
((addtoken self)
 (increment ltoks)
 (increment toks)
 (walk (lambda (arc)
           (set (tokens arc) (fx+ 1 (tokens arc))))
         als)
 )
((remtoken self)
 (decrement ltoks)
 (decrement toks)
 (walk (lambda (arc)
           (set (tokens arc) (fx- (tokens arc) 1)))
         als)
)
((disconnect self node)
 (del-object node self)
 (set hset (delq node hset))
 (set test (delq node hset))
 (disconnect (inner-object self) node))
((checkbrk self tag)
 (break? bpoint tag toks))
```

```
  ((controlArc? self) '#T)
  ((store self port)
   (format port
           "(AddCarcFromDB ThisCG ~s '~s '~s ~d ~d ~d '~s ~d)~%"
           (textshp-txt (rtshape (name (inner-object self))))
           (map (lambda (x)
                  (list (name (car x))
                        (point-x (cplxshp-loc (abshape (cadr x))))
                        (point-y (cplxshp-loc (abshape (cadr x))))
                        (list (ptlist->xylist
                               (polyshp-pts (abshape (cadr
                                                     (children (cadr x))))))
                              (ptlist->xylist
                               (splishp-pts
                                (abshape (car (children (cadr x)))))))))
                (headpairs self))
           (map (lambda (x)
                  (list (name (car x))
                        (ptlist->xylist
                         (splishp-pts (abshape  (cadr x))))))
                (tailpairs self))
           (initialtokens self)
           (point-x (cplxshp-loc (abshape self)))
           (point-y (cplxshp-loc (abshape self)))
           (ptlist->xylist (splishp-pts (abshape (body self))))
           (Pscale self)))))

;;;**********************
;;;* The Dataset Object *
;;;**********************
(define (MakeDataSet)
 ;; static information
 (let ((dname nil)                     ; name
       (dival nil)                     ; initial value
       (darcs nil)                     ; data arcs

       ;; dynamic information
       (dval nil)                      ; current value
       )
  (object nil
          ((print self port)
           (format port "{SemanticDataSet ~d ~a}~%"
                   (object-hash self)
                   (symbol->string (name self))))
          ((name self) dname)
          ((gmb self) (parent (parent self)))
          ((initialValue self) dival)
          ((Value self) dval)
          ((dataArcs self) darcs)
          (((setter name) self val)
           (set dname val))
          (((setter initialValue) self val)
              (set dival val))
             (((setter Value) self val)
              (set dval val))
             (((setter dataArcs) self val)
              (set darcs val))
             ((add-object self x)
              (cond ((dataarc? x)
                     (set darcs (cons x darcs)))
                    (else (error-mess "not a data arc"))))
             ((del-object self x)
              (cond ((dataarc? x)
                     (set darcs (delq x darcs)))
                    (else (error-mess "not a data arc"))))
             ((delete self)
              (walk delete darcs)
              (del-object (parent self) self)
              (erase self))
             ((store self port)
              (format port
                      "(AddDsetFromDB ThisDg ~s '~s '~s ~d ~d ~d ~d ~d
                                  ~d ~d ~d ~d ~d ~d ~d ~d)~%"
                      (textshp-txt (rtshape (name (inner-object self))))
                      (initialValue self)
                      (Value self)
                      (point-x (cplxshp-loc (abshape self)))
                      (point-y (cplxshp-loc (abshape  self)))
                      (rectshp-wi (abshape (rect self)))
                      (rectshp-he (abshape (rect self)))
                      (northleft self)
                      (northright self)
                      (southleft self)
                      (southright self)
                      (easttop self)
                      (eastbottom self)
                      (westtop self)
                      (westbottom self)
                      (Pscale self)))
             ((dataset? self) '#T))))

;;;***********************
;;;* The Data Arc Object *
;;;***********************
(define (MakeDataArc)
  ;; static information
  (let  ((dname nil)                    ; name
         (dsets nil)                    ; can be also a socket
         (dprocs nil)                   ; can contain sockets
         ;; dynamic information
         (ddsets nil)                   ; dynamic dataset
         (varctype 'rw)
         )
    (object
nil
((print self port)
 (format port "{SemanticDataArc ~d ~a}~%"
          (object-hash self)
          (symbol->string (name self))))
((name self) dname)
((gmb self) (parent (parent self)))
((dataset self) dsets)
((dyn-dataset self) ddsets)
((dataprocs self) dprocs)
(((setter name) self val)
 (set dname val))
(((setter dataset) self val)
 (set dsets val))
(((setter dyn-dataset) self val)
 (set ddsets val))
(((setter dataprocs) self val)
 (set dprocs val))
((arctype self) varctype)
(((setter arctype) self v)
 (set varctype v))
;; need to handle sockets
((add-object self x)
 (cond
  ((dataProcessor? x) (set dprocs x ))
  ((dataset? x) (set dsets  x )
      (set ddsets x ))
  (else (cond ((null? dprocs)
                (set dprocs x))
               (else
                (set dsets x)
                (set ddsets x))))))
((del-object self x)
 (cond ((eq? x dprocs)
         (set dprocs nil))
        (else
         (set dsets nil)
         (set ddsets nil))))
((delete self)
 (del-object (parent self) self)
 (del-object dprocs self)
 (del-object dsets self)
 (erase self))
((store self port)
 (format port
          "(AddDarcFromDB ThisDG ~s '~s '~s ~s '~s ~d ~d '~s ~d)~%"
          (textshp-txt (rtshape (name (inner-object self))))
          (name (obj1 self))
          (name (obj2 self))
          (cond ((null? (dyn-dataset self)) 'nil)
                (else
                 (textshp-txt
                 (rtshape (name (inner-object
                                 (dyn-dataset self)))))))
          (ptlist->xylist
           (polyshp-pts (abshape (polyline self))))
          (point-x (cplxshp-loc (abshape self)))
          (point-y (cplxshp-loc (abshape self)))
          (arctype self)
          (Pscale self)))
   ((dataArc? self) '#T))))


;;;**********************************
;;;* The Data Processor Object *
;;;**********************************
(define (MakeDataProc)
 ;; all attributes are static
 (let ((pname nil)                   ; name
       (darcs nil)                   ; data arcs
       (id nil))                     ; id
  (object nil
          ((print self port)
           (format port "{semanticDataProc ~d ~a}~%"
                 (object-hash self)
                 (symbol->string (name self))))
          ((name self) pname)
          ((gmb self) (parent (parent self)))
          ((dataArcs self) darcs)
          ((idcode self) id)
          (((setter name) self val)
           (set pname val))
          (((setter dataArcs) self val)
           (set darcs val))
          (((setter idcode) self val)
           (set id val))
          ((delMapping self cnlist)
           (set cnlist (map xy->container cnlist))
           (cond ((null? cnlist)
                  ;; if no cnodes given, all the mapping to
                  ;; data proc is removed
                  (set (cg-dgMapping (parent (parent self)))
                        (del (lambda (x y)
                               (eq? (cadr y) x))
                             self
                             (cg-dgMapping (parent (parent self))))))
                 (else
                  ;; otherwise take care only of the specified nodes
                  (walk
                   (lambda (cnode)
                    (set (cg-dgMapping (parent (parent self)))
                         (del (lambda (x y)
                                (and (eq? (car y) x)
                                     (eq? (cadr y) self)))
                                cnode
                                (cg-dgMapping (parent (parent self))))))
              cnlist))))
         ((addmapping self clist)
          (set clist (map xy->container clist))
          (cond ((and (every? (lambda (x)
                                  (eq? (parent (parent self))
                                       (parent (parent x))))
                                Clist)
                      (every? Controlnode? CList))
                 (set (cg-dgMapping (parent (parent self)))
                      (append  (map (lambda (x)
                                        (list x self))
                                     Clist)
                                (cg-dgMapping (parent (parent self))))))
                (else
                 ;; this is an error
                 (error-mess "invalid mapping"))))
         ((add-object self x)
          (cond ((dataarc? x) (set darcs (cons x darcs)))
                (else (error-mess "not a Data Arc"))))
         ((del-object self x)
          (cond ((dataarc? x) (set darcs (delq x darcs)))
                (else (error-mess "not a Data Arc"))))
         ((delete self)
          ;;may need change if rep. of dataarc is changed
          (walk delete darcs)
          (del-object (parent self) self)
          (erase self))
         ((store self port)
          (format port
                  "(AddDprocFromDB ThisDG ~s '~s ~d ~d ~d ~d '~s
                                  ~d ~d ~d ~d ~d ~d ~d ~d ~d)~%"
                  (textshp-txt (rtshape (name (inner-object self))))
                  (idcode self)
                  (point-x (cplxshp-loc (abshape self)))
                  (point-y (cplxshp-loc (abshape self)))
                  (point-x (pgonshp-base (abshape (polygon self))))
                  (point-y (pgonshp-base (abshape (polygon self))))
                  (ptlist->xylist (pgonshp-pts
                                   (abshape (polygon self))))
                  (northleft self)
                  (northright self)
                  (southleft self)
                  (southright self)
                  (easttop self)
                  (eastbottom self)
                  (westtop self)
                  (westbottom self)
                  (Pscale self)))
         ((dataprocessor? self) '#T))))
;;****************************************
```

APPENDIX C

DB_Classfier Code

```
;After all objects have been transformed into entities, then
; the relationships are examined, one at a time, and their
;entity type, if applicable, is determined.
;After creation of the appropriate entity type or modification of
;existing entities the relationship is marked.
;This portion will also examine roles associated with the arcs connected
;to each relationship and create the proper entity.
;Relationships with no roles and participating in existence dependencies
; will not  be represented. There is no loss to the semantic understanding
;of the model.

(walk (lambda(x)
        (block (print x (standard-output))
               (if (not (checked x))

;Relationships with E constraint and no roles on arcs.

        (cond
           ((and (check-constraints 'E x) (not (role-attached (arcs x))))
            (if (properties x)
                  (block
                   (set (checked x) t)
                   (characteristic-create x (name (find-1-entity x)))
                   (designative-create (find-dep-entity x) x))))
           ;Relationships with E constraint and roles attached to some arcs.
           ((and (check-constraints 'E x) (role-attached (arcs x)))
            (print "special case" (standard-output)))

           ;Relationships with no E constraints and roles attached to arcs.
           ((role-attached (arcs x))
            (if (recursive-relation x)
                  (if (and (or (check-constraints 'E (car (arcs x)))
                               (check-constraints 'E (car (cdr (arcs x)))))
                           (or (string-equal? "1toN"
                                              (mapping-string
                                               (car (arcs x)) (car (cdr (arcs x)))))
                               (string-equal? "Nto1"
                                              (mapping-string
                                               (car (arcs x))(car (cdr (arcs x))))))
                      (block (subtype-create
                               (role (find-arc '1 x)) (find-1-entity x))
```

```
                    (characteristic-create
                     (role (find-arc 'N x))
                     (role (find-arc '1 x)))
                    (set (checked x) t))
              (block
               (subtype-create
                (role (car (arcs x)))(find--1-entity x))
               (subtype-create
                (role (car (cdr (arcs x)))) (find-1-entity x))
               (set (checked x) t))) nil))
        ;Relationships with no E constraints and no roles.
        (else (block
              (if (eq? (length (arcs x)) 2)
                (cond ((or
                         (string-equal? "1toN"
                                      (mapping-string
                                       (car (arcs x))
                                       (car (cdr (arcs x)))))
                         (string-equal? "Nto1" (mapping-string
                                              (car (arcs x))
                                              (car (cdr (arcs x))))))
                        (designative-create
                         find-n-entity find-1-entity))
                       ((or (string-equal? "MtoN" (mapping-string
                                                  (car (arcs x))
                                                  (car (cdr (arcs x)))))
                            (string-equal? "NtoM" (mapping-string
                                                  (car (arcs x))
                                                  (car (cdr (arcs x))))))

                        (associative-create x))) nil)))))))

 (relations *erd*)))
;Functions which create the various entities are next.
;***********************
; This function creates a designative entity
;**********************
(define (designative-create object1 object2)
   (create-entity (name object1) "D" (list (return-erelation (name object2))) '()))


;********************
;This function creates an associative entity
;********************
(define (associative-create relation)
 (let ((extra '())
        (entity '()))
    (if (set extra (check-multiplicity relation))
       (block
          (set entity (create-entity
                       (name relation) "A" (object-list relation)
                       (properties relation)))
          (walk (lambda(x)
                        (create-entity (car x) "C" (list entity)
                                        x))
                extra))
          (create-entity
                        (name relation) "A" (object-list relation)
                        (properties relation)))))



;********************
;This function creates a type entity
;**************************
(define (subtype-create rolename supertype)
 (create-entity rolename "S" (append (list
                                  (return-erelation (name supertype)))
                                 (list "has"))'()))




;************************
;this function defines a characteristic entity
;************************
(define (characteristic-create object1 object2)
 (let ((extra '())
       (entity '()))
    (if (if (objects? object1)
             (set extra (check-multiplicity object1)) nil)
       (block
             (if (string? object1)
                (set entity
                      (create-entity object1 "C"
                                     (list
                                      (return-erelation object2)) '()))
                (set entity (create-entity (name object1) "C"
                 (list (return-erelation object2))
                 (properties object1)))))
          (walk (lambda(x)
                        (create-entity (car x) "C" (list entity)
                                        x))
                extra))
          (if (string? object1)
             (create-entity object1 "C"
                           (list (return-erelation object2)) '())
             (create-entity (name object1) "C"
                           (list (return-erelation object2))
                           (properties object1)))))

;*********************
;This function defines kernel entities
;************************
```

```
(define (kernel-create object1)
 (let ((extra '())
        (entity '()))
     (if (set extra (check-multiplicity object1))
          (block
         (set entity (create-entity (name object1) "K" '() (properties object1)))
         (walk (lambda (x)
                              (create-entity (car x) "C" (list entity) (list x)))
            extra))
        (create-entity (name object1) "K" '() (properties object1)))))

;*********************
;This function recursively generates characteristic entities
;************************
(define (parent-create x)
 (cond ((check-constraints 'E x)
         (if (checked (find-other-object x))
            (block
             (set (checked x) t)
             (characteristic-create x (find-other-object x)))
            (block
             (parent-create x)
             (set (checked x) t)
             (characteristic-create x (find-other-object x)))))
        (else
         (set (checked x) t)
         (kernel-create x)))
        (print "in parent create" (standard-output)))
```

REFERENCES

[Afsa85] Afsarmanesh, H., ''The 3 Dimensional Information Space(3DIS), An Extensible Browsing-Oriented Framework for Database Systems,'' University of Southern California, Los Angeles, CA, (1985).

[Afsa85] Afsarmanesh, H., D. McLeod, D. Knapp, and A. Parker, ''An Extensible Object-Oriented Approach to Databases for VLSI/CAD,'' Tech. Rep. CRI-85-09, University of Southern California, Los Angeles, CA, (October 7, 1985).

[Beet82] Beetem, A., J. Milton, and G. Wiedenhold, ''Performance of Database Management Systems in VLSI Design,'' *Database Engineering Newsletter* **5**(2), pp. 15-20 (June 1982).

[Benn82] Bennett, J., ''A Database Management System for Design Engineers,'' *19th Design Automation Conference*, pp. 268-273 (June 14-16, 1982).

[Berr84] Berry, D.N., ''On the Use of ADA as a Module Interface Description,'' in *Proceedings Hawaii International Conference on System Sciences* (January 1984).

[Blai85] Blain, T., M. Dohler, R. Michaels, and E. Qureshi, ''Managing the Printed Circuit Board Design Process,'' pp. 477 - 485 in *Proceedings ACM - SIGMOD International Conference on the Management of Data*, Austin TX (December 1985).

[Boeh84] Boehm, B., M. Penedo, E. Stuckel, R. William, and A. Pyster, ''A Software Development Environment to Improve Productivity,'' *IEEE Computer*, pp. 30 -40 (1984).

[Bork78] Borkin, S.A., ''Data Model Equivalence,'' pp. 526-534 in *Proceedings 4th International Conference VLDB* (1978).

[Brow83] Brown, H., C. Tong, and G. Foyster, ''Palladio: An Exploratory Environment for Circuit Design,'' *IEEE Computer* **15**(12), pp. 41-58 (December 1983).

[Cai85] Cai, S., D. Landis, D. Patel, and D. Worley, ''An Experiment to Determine Appropriate Data Structuring Methods for the SARA/IDEAS Structure Model (SM),'' Tech. Rep. Internal Memorandum #212, Computer Science Department, University of California, Los Angeles, Los Angeles CA, (1985).

[Camp78] Campos, I. M. and G. Estrin, ''SARA Aided Design of Software for Concurrent Systems,'' *Proceedings of the National Computer Conference*, Anaheim, California (June 1978).

[Chen77] Chen, P.P.S., "The Entity Relationship Model- A Basis for The Enterprise View of Data," *AFIPS Conference Proceedings of National Computer Conference* **46**, pp. 77-84 (1977).

[Chou82] Chou, H-T, Q. DeWitt, R. Katz, and A. Klug, *Design and Implementation of the Wisconsin Storage System,* Computer Science Department, U of Wisconsin,Madison (1982).

[Ciam76] Ciampi, P. L., A. D. Donovan, and J. D. Nash, "Control and Integration of a CAD Data Base," pp. 394 -401 in *Proceedings 13th Design Automation Conference* (June 1976).

[Codd79] Codd, E. F., "Extending the Database Relational Model to Capture More Meaning," *ACM Transactions on Database Systems* **4**(4), pp. 397-434 (December 1979).

[Davi83] Davis, R. and H. Strobe, "Representing Structure and Behavior of Digital Hardware," *IEEE Computer* **16**(10), pp. 75-82 (October 1983).

[Defe80] Defense, US Department of, *STONEMAN: DoD Requirements for the Ada Programming Support Environment*, February 1980.

[Drob80] Drobman, J. H., "A Model-Based Design System and Methodology for Composition of Microprocessor-Based Digital Systems," Tech. Rep. PhD Dissertation, Computer Science Department, University of California, Los Angeles, (1980).

[Dump81] Dumpala, S. R. and S. K. Arora, "Schema Translation Using the Entity-Relationship Approach," *Entity-Relationship Approach to Information Modeling and Analysis*, pp. 339-360 (October 1981).

[East80] Eastman, C. M., "System Facilities for CAD Databases," *Computer Science Research Review*, pp. 36-52 (1980).

[East81] Eastman, C. M. and G. M. E. LaFue, "Semantic Integrity Transactions in Design Databases," pp. 45-54 in *File Structures and Data Bases for CAD*, North-Holland (1981).

[East81] Eastman, C. M., "Recent Developments in Representation in the Science of Design," *18th Design Automation Conference*, pp. 13-20 (1981).

[Enca83] Encarnacao, J. and E. G. Schlechtendahl, *Computer Aided Design,* Springer-Verlag, Berlin Heidelberg New York Tokyo (1983).

[Estr78] Estrin, G., "A Methodology for Design of Digital Systems - Supported by SARA at the Age of One," *Proceedings of the National Computer Conference*, Anaheim, California (June 1978).

[Fenc78] Fenchel, R. S., "SARA SLR(1) Grammar Tools," Tech. Rep. 186, UCLA Computer Science Department, Los Angeles, (November 1978).

[Fenc80] Fenchel, R. S., ''Integral Help for Interactive systems,'' Tech. Rep. UCLA-ENG-8051, UCLA Computer Science Department, Los Angeles, California, (September 1980).

[Fenc81] Fenchel, R. S. and G. Estrin, ''Self-Describing Systems Using Integral Help,'' *IEEE Transactions on Systems, Man and Cybernetics special issue on User Assistance and Human Fators in Interactive Computer Systems* ???(???), p. ??? (Late Summer 1981).

[Fisc81] Fischer, W. E., ''PHIDAS - A Database Management System for CAD/CAM Application Software,'' *Computer Aided Design* **13**(4), p. tbd (1981).

[Fois81] Foisseau, J. and F. R. Valette, ''A Computer Aided Design Data Model: FLOREAL,'' pp. 315-330 in *File Structures and Data Bases for CAD*, North-Holland (1981).

[Frie82] Friedenson, R. A., J. R. Breiland, and T. J. Thompson, ''Designer's Workbench: Delivery of CAD Tools,'' *19th Design Automation Conference*, pp. 15-21 (June 14-16, 1982).

[Glas82] Glass, G. J., ''A User Interface for Architectural Design, A Case Study,'' *19th Design Automation Conference*, pp. 508-513 (June 14-16, 1982).

[Gold83] Goldberg, A. and D. Robson, *Smalltalk-80, The Language and its Implementation*, 1983.

[Gold84] Goldberg, A., *Smalltalk-80, The Interactive Programming Environment*, 1984.

[Gutt82] Guttman, A. and M. Stonebraker, ''Using a Relational Database Management System for Computer Aided Design,'' *Database Engineering Newsletter* **5**(2), pp. 21 -28 (June 1982).

[Hask82] Haskin, R. and R. Lorie, ''Relational Database for Circuit Design,'' *Database Engineering Newsletter* **5**(2), pp. 10-14 (June 1982).

[Hayn81] Haynie, M. H., ''The Relational/Network Data Model for Design Automation Databases,'' *Proceedings of the Eighteenth Design Automation Conference*, pp. 646-652 (June 23-25, 1981).

[Hosk81] Hoskins, E. M., ''Descriptive Databases in Some Design Manufacturing Environments,'' *Computer Aided Design* **13**(4), p. tbd (1981).

[Hutc85] Hutchings, A., R. Bonneau, and W. Fisher, ''Integrated VLSI CAD Systems at Digital Equipment Corporation,'' pp. 524-548 in *Proceedings 22nd ACM/IEEE Design Automation Conference* (June 1985).

[Inmo81] Inmon, W. H., in *Effective Data Base Design*, Prentice Hall, Inc., New Jersey (1981).

[Jajo83] Jajodia, S. and P.A. Ng, "On the Representation of Relational Structures by Entity-Relationship Diagrams," pp. 249-263 in *Proceedings Entity-Relatioinship Approach to Software Engineering*, Anaheim, CA (October 1983).

[John82] Johnson, H. R. and D. L. Bernhardt, "Engineering Data Management Activities Within the IPAD Project," *Database Engineering Newsletter* **5**(2), pp. 2-9 (June 1982).

[Katz82] Katz, R., "DAVID: Design Aids for VLSI Using Integrated Databases," *Database Engineering Newsletter* **5**(2), pp. 29-32 (June 1982).

[Katz83] Katz, R., "Managing the Chip Design Database," *IEEE Computer* **16**(12), pp. 26-36 (December 1983).

[Katz84] Katz, R. and S. Weiss, *Design Transaction Management,* unknown (1984).

[Katz85] Katz, R., *Information Management for Engineering Design,* Springer Verlag, Berlin New York (1985).

[Kers81] Kersten, M. L. and A. I. Wasserman, "The Architecture of the PLAIN Data Base Handler," *Software- Practice and Experience* **II.**, pp. 175-186 (1981).

[Kore75] Korenjak, A. J. and A. H. Teger, "An Integrated CAD Data Base System," pp. 399-406 in *Proceedings 12th Design Automation Conference* (June 1975).

[Kras83] Krasner, G. editor, *Smalltalk-80, Bits of History, Words of Advice,* 1983.

[Krel86] Krell, E. A., "ADA as an Interpretation Language for SARA," Tech. Rep. Phd Dissertation, UCLA Computer Science Department, Los Angeles, California, (1986).

[Land83] Landis, D. M., "Design Considerations for the Satisfaction of CAD Library Requirements," Tech. Rep. UCLA CSD-830614, UCLA Computer Science Department, Los Angeles, California, (June 1983).

[Lien80] Lien, Y.E., "On the Equivalence of Database Models," Tech. Rep. 3, Bell Labs, Holmdel, N.J., (1980).

[Lori81] Lorie, R., "Issues in Databases for Design Applications," pp. 213-222 in *File Structures and Data Bases for CAD*, North-Holland (1981).

[Lusk80] Lusk, E.L., R.A. Overbeek, and B. Parrello, "A Practical Design Methodology for the Implementation of IMS Databases, Using the Entity-Relationship Model," pp. 99-21 in *Proceedings ACM-SIGMOD 1980 International Conference on Management of Data*, Santa Monica, CA (May 1980).

[Madd81] Madden, J. and V. T. Taylor, "Design Data Requirements in the Process Industries," *Computer Aided Design* **13**(4), p. tbd (1981).

[Maie86] Maier, D., J. Stern, A. Otis, and A. Purdy, "Development of an Object Oriented DBMS," pp. 472 - 482 in *Proceedings OOPSLA Conference*, Portland OR (November 1986).

[Mars83] Marshall, J., "A Design Automation Database for Computer Aided Design of Computer Systems," Tech. Rep. PhD Dissertation, UCLA Computer Science Department, Los Angeles, California, (1983).

[Myer82] Myers, W., "CAD/CAM: The Need for a Broader Focus," *IEEE Computer*, pp. 105-116 (January 1982).

[Narf84] Narfelt, K. and D. Schefstrom, "Towards a KAPSE Database," *IEEE Ada Applications and Environments*, pp. 42 -51 (October 1984).

[Pene79] Penedo, M. H. and D. M. Berry, "The Use of a Module Interconnection Language in the SARA System Design Methodology," in *Proceedings 4th Internationl Conference on Software Engineering* (September 1979).

[Pene81] Penedo, M. H., "The Use of a Module Interface Description in the Synthesis of Reliable Software Systems," Tech. Rep. Phd Dissertation, UCLA Computer Science Department, Los Angeles, California, (1981).

[Pete81] Peterson, J. L., *Petri Net Theory and the Modeling of Systems*, 1981.

[Pete83] Peterson, J. L. and A. Silberschatz, "Design Principles," pp. 422 in *Operating System Concepts*, Addison-Wewley Publishing Company, Inc. (1983).

[Prei82] Preiss, K., "Shape Representation and Solid Modelling in CAD Systems," pp. 155-167 in *CAD Systems Framework*, ed. F. M. Lillehagen, North-Holland, Roros, Norway (June 1982).

[Razo77] Razouk, R. R. and G. Estrin, "The Graph Model of Behavior," *Proceedings of the Symposium on Design Automation and Microprocessors*, pp. 67-76, IEEE Piscataway, New Jersey (December 1980).

[Razo79] Razouk, R. R., M. Vernon, and G. Estrin, "Evaluation Methods in SARA - The Graph Model Simulator," pp. 189-206 in *Proceedings of Conference on Simulation, Measures and Modeling of Computer Systems* (1979).

[Rood85] Rood, H.J., *Logic and Structured Design for Computer Programmers*, Prindle, Weber, and Schmidt, Boston, MA (1985).

[Samp79] Sampaio, A.B.C., "A Scheme of Attributes to Detect Inconsistencies in Design of Computer Systems," Tech. Rep. PhD Dissertation Prospectus, Computer Science Department, University of California, Los Angeles, (1979).

[Sche79] Scherermann, P., G. Schiffner, and H. Weber, "Abstraction Capabilities and Invariant Properties Modelling Within theEntity-Relationship Approach," pp. 121-140 in *Proceedings International Conference on Entity-Relationship Approach to Systems Analysis and Design*, Los Angeles, CA (1979).

[Shoo83] Shoonan, M.J., *Software Engineering,* McGraw Hill, New York (1983).

[Sidl80] Sidle, T. W., "Weaknesses of Commercial Data Base Management Systems in Engineering Applications," *Proceedings of the Seventeenth Design Automation Conference*, pp. 57-61 (June 1980).

[Slad87] Slade, S., *The T Programming Language,* Prentice Hall, Princeton N. J. (1987).

[Snug79] Snuggs, M. E., G. J. Popek, and R. J. Peterson, "Data Base System Objectives and Design Constraints," UCLA, Los Angeles, CA, (1979).

[Tayl86] Taylor, H., L. Clarke, L. Osterweil, J. Wiledin, and M. Young, "Arcadia : A Software Development Research Project," *IEEE Software*, pp. 137 - 149 (April 1986).

[Tich82] Tichy, W. F., "A Data Model for Programming Support Environments and its Application," *Automated Tools for Information Systems Design-IFIP*, pp. 31-47, North-Holland Publishing Company (1982).

[Tich82] Tichy, W. F., "Design Implementation, and Evaluation of a Revision Control system," *IEEE*, pp. 58-67 (1982).

[Tsub81] Tsubaki, M., "Family Generation in Integrated Engineering Systems," *Computer Aided Design* **13**(4), p. tbd (1981).

[Ulfs81] Ulfsby, S., S. Meen, and J. Oian, "TORNADO: A DBMS for CAD/CAM Systems," *Computer Aided Design* **13**(4), pp. 193-197 (1981).

[Vern78] Vernon, M., R. S. Fenchel, and R. Razouk, "Standard Procedure for Designing a New Tool for the SARA System," Tech. Rep. 185, UCLA Computer Science Department, Los Angeles, California, (November 1978).

[Wass81] Wasserman, A. I., "Automated Development Environments," *IEEE Computer* **14**(4), pp. 7 - 11 (April 1981).

[Winc81] Winchester, J. W., "Requirements Definition and its Interface to the SARA Design Methodology Based Systems," Tech. Rep. PhD Dissertation, UCLA Computer Science Department, Los Angeles, California, (1981).

[Wood70] Woods, W.A., "Transition Network Grammar for Natural Language Analysis," *Communications of the ACM* **13**(10), pp. 591-606 (October 1970).

[Worl86] Worley, D. R., ''A Methodology, Specification Language and Automated Support Environment for Computer Aided Design Systems,'' Tech. Rep. UCLA Computer Science Department, (1986).