MODEL, LANGUAGE AND IMPLEMENTATION ASPECTS OF A LOGIC-BASED OBJECT-ORIENTED DATABASE SYSTEM

Jianhua Zhu
B.Sc., South-China Institute of Technology, P.R. China, 1982
M.Sc., Oregon State University, Corvallis, Oregon, 1984

A dissertation submitted to the faculty
of the Oregon Graduate Institute
of Science and Technology
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

July, 1989

The dissertation "Model, Language and Implementation Aspects of A Logic-Based Object-Oriented Database System" by Jianhua Zhu has been examined and approved by the following Examination committee:

Dr. David Maier, Advisor
Professor, Oregon Graduate Institute

Dr. Daniel Hammerstrom
Associate Professor, Oregon Graduate Institute

[illegible]
Dr. Goetz Graefe
Assistant Professor, University of Colorado

Dr. Earl F. Ecklund, Jr
Principle Scientist, Mentor Graphics Corporation

ACKNOWLEDGEMENT

I am indebted to numerous people, without whom this dissertation would not have been possible. I wish to express my sincere thanks to Dr. David Maier, my thesis advisor. He is not only an excellent teacher and advisor with high standards, but also a wonderful person and a great role model in every aspect of life. I am grateful for his intellectual inspiration, professional guidance, constant encouragement, as well as the compliment that "I have a facility with the English language that eludes most native speakers." I wish to express my sincere thanks to Dr. Earl Ecklund, who taught me the first lesson on the subject of database management systems. He has also been a constant source of knowledge and encouragement. I wish to express my sincere thanks to the other members of my thesis committee. Dr. Daniel Hammerstrom gave me much needed help on applying object-oriented data models to the domain of microarchitecture simulations. Dr. Goetz Graefe provided many suggestions that have improved the clarity and presentation of the thesis. I also wish to thank the other faculty members and fellow graduate students of the Computer Science and Engineering department, from whom I have been learning the science of computing and the art of life.

I thank my parents, Siping Zhu and Yuzhen Zhang, for their enduring love; I thank my wife, Xing Liu, for her caring and emotional support.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Model, Language and Implementation Aspects
of A Logic-based Object-Oriented Database System

Jianhua Zhu, Ph.D.
Oregon Graduate Institute of Science and Technology

Supervising Professor: David Maier

With the goal of a "total objectification", this study investigates issues and extensions to conceptual data models founded on a logic-based and object-oriented framework. In particular, a distinction was proposed to separate logic variables into pure placeholders and object tags. The concept of *abstract objects* was articulated and was used to interpret the class of object tags.

Abstract objects and their pattern-matching semantics make it possible to interpret database languages uniformly as database objects. Therefore, the semantics of a language is directly determined by database objects. The implication is that the resulting data model can be decoupled from specific database languages. Concrete syntax is highly adaptable to individual application domains; and variations on syntax affect neither the underlying representation nor processing.

A memory-based prototype was implemented. The implementation consisted of a Smalltalk-based user interaction facility and a Prolog-based model manager, with a variety of database language interfaces that support object construction, type definition, command definition and interactive query processing.

CHAPTER 1

INTRODUCTION

This thesis studies several important aspects of a conceptual data model with features from the object-oriented programming paradigm combined with ideas from the logic programming area. The Tektronix Engineering Data Model (TEDM), originally proposed by Maier [Maier85], intends to explore alternative ways of supporting non-traditional database applications using traditional database technologies. The study encompasses several issues in database systems including modeling, language, as well as implementation. Other equally important issues such as concurrency control and query optimization are beyond the scope of this thesis. It should be noted that we employ TEDM as a vehicle for our study; many of the concepts presented throughout the thesis are generally applicable, and need not be constrained to any specific data model.

As of today, the predominant technologies for most of the database applications have been based on the following three data models: the *hierarchical* model, the *network* model and the *relational* model. (We call them traditional data models in this thesis to distinguish them from the new generation of database technologies, such as semantic models and object-oriented models.) These traditional data models, all dating back to 60's or early 70's, are very effective and perform well in many application areas. For example, the relational model fits very well with business data processing, such as bank transaction processing, warehouse inventory tracking and personnel information systems for organizations. These examples all share a common characteristic: The data structure required for representing information is simple enough that most of the time a single level table is sufficient. However, the effectiveness of the traditional data models has been strongly challenged by application domains where information structure does not share this simplicity. An example where support for more complex data structure is needed is the use of databases to support computer-aided design and manufacturing (CAD/CAM). A minimum set of requirements on a database system for effective support of CAD/CAM applications is:

1) A set of formal concepts that allow the design objects to be rigorously described, in a way that is consistent with or close to the mental model.

2) A flexible storage manager that allows the design objects to be stored economically, retrieved efficiently and manipulated consistently.

3) An expressive command (query and manipulation) language that allows retrieval and updates against the database to be formulated easily.

4) A transaction facility that supports access and update to large amount of data, over a long period of time.

Apparently, the traditional database technology comes up short-handed, being scrutinized under this set of requirements. In particular, the hierarchical model and the network model provide insufficient support for high level abstraction. The users of such systems have to work at the level of the storage structure: physical adjacency in the hierarchical model and pointers in the network model. The relational data model supplies a high level query and manipulation language (the relational algebra). The user of such a system is shielded from the physical properties of the the database, a major improvement over the hierarchical model and the network model. But it provides simple (first normal form) tables as its only data structuring facility, which is too restrictive and too inefficient for applications where primary data elements are records with mutual references. When forced to use simple tables for their representation, complex objects become highly fragmented, resulting in a loss of access efficiency. Another problem with decomposing complex objects into first normal form is that doing so changes the appearance of the conceptual objects, resulting in obscured semantics.

Despite the negative results of many studies on the effectiveness of the traditional database technology on application domains other than business applications (see, for example, [Maier84, Rosenberg80 and Sidle80]), most of the applications are still being built using these platforms. Our study investigates new concepts and strategies to improve the effectiveness and efficiency of the database technology, and to make them applicable to complex applications.

The demand on good database technology is an ever increasing trend, as more and more applications find database support a necessity, for improved quality and productivity. The CAD/CAM example a couple of paragraphs back is a point in case. We briefly mention a few others. The database is also an ideal place to store the outcome of a design, which is the master plan for building machinery, equipment or circuitry. (In electronic design, such a master plan would include what components are used and how the components are interconnected.) But there are other advantages more significant than making available a storage space. Firstly, the database is not limited to storing the outcome of a design. It can store the design process itself as well. That is, all of the important design issues and decisions can be recorded along with the result of the design. This information is a valuable source of quality measurement, process evaluation and product improvement. Secondly, when the design is stored as database objects, the structure or other important details about the design are available to the people who have access to the database. Those who understand the semantics of the database representation can interpret the design correctly. In this scenario, the database representation is a common language for communication among the design engineers, or more accurately, among design tools that the engineers use. Thirdly, the access and update protocols of the database can be used as a coordination mechanism for team work, such that each member has the freedom to do things that he or she feels appropriate. In the meantime, each individual's "independent" work is synchronized to progress towards the team's common goal.

Another example of a new application domain for databases is computer-aided software engineering (CASE). In this case, again, it is important to have a database underneath other development tools. The common repository facilitates information exchange among the tools. It also provides reliable storage for information that is vital to every aspect of CASE: analysis, specification, design, coding, testing, maintenance, project management and configuration management.

While there is no great difficulty for people to recognize the fact that there is a need to extend the database technology and that the need is urgent, it is rather challenging a task to make such extensions. It is even more difficult for people to agree upon a common set of extensions that will solve the issues and problems satisfactorily. Our study will not be able to address all the issues. What we do hope to accomplish is to help identify critical success factors, improve our understanding of them, suggest our solutions and move one small step closer towards the grand goal — extending the database technology to support complex applications, efficiently and effectively.

In this thesis, we identify data-structuring techniques suitable to model complex data elements. We describe a command language for querying and updating complex objects. We explore the technique of deductive query processing in object-oriented databases. We propose an extension to database and query formalisms, to make it possible to store query patterns as objects. The theme of the dissertation, thus, is an investigation of the feasibility and the advantages of combining features from the object-oriented programming paradigm and the logic programming languages in data models.

This current chapter contains five sections. Section 1 is a short account of the basics, and Section 2 of the history, of the data processing technology. Recent work on databases and conceptual modeling is surveyed in Section 3. We also relate our study in that section to similar projects. Section 4 gives a brief description for some prominent features of our study vehicle, the TEDM data model. Section 5 contains an outline of the dissertation.

1.1. Background

To maintain a consistent terminology used in the thesis, key terms in the database world are supplied here. A word of caution: although these terms are ubiquitous in literature, they are nevertheless nonstandard. We also overview the basic characteristics of database systems, and contrast them to the more conventional file systems.

A *data model* is a collection of tools made available to the user by a computerized data-processing facility, typically for the purpose of defining, populating, manipulating and constraining application data. These tools usually come in some form of data-processing languages. The integrity constraint specification is often considered a part of the data definition, and the data population is a part of data manipulation. Hence, a "data model" is a collective term for a data definition language (DDL) and a data manipulation language (DML).

A *database* is a collection of application data and related control information, usually stored on a non-volatile device, with the condition that the data are structured according to a conceptual view, the *conceptual schema* of the database. The conceptual schema is also called the *intension* of the database; whereas the database itself is said to be an *instance* (or an *extension*) of the conceptual schema. (According to this definition, database language facilities are not part of the database. A major contribution of this dissertation lies in the formalization and mechanisms for making various database language constructs, such as compound commands and queries, part of the database.)

A *database management system* (DBMS) is the realization of a data model, by a piece of software, possibly with dedicated hardware support. In addition to implementing what a data model has to offer to its user, namely a DDL and a DML, a database management system often provides mechanisms for dealing with many other practical issues in a multi-user, time-sharing service environment, such as security, concurrency and recovery.

Database objects have a distinguishing property when compared with programming data: they are non-volatile, or *persistent*. In other words, their lifetime extends beyond the computing processes that manipulate them. An obvious way to attain this extended lifetime is through *files*. A file very often amounts simply to a byte stream without any attached semantics. In this scenario, it is entirely up to the program that uses and manipulates the file to interpret the raw data bytes, which typically involves the tedious task of counting bits and bytes within the file pages.

Databases promote data sharing among different applications. They support persistent objects and associative accessing. Navigational accessing is also possible and is expressed at much higher a level than a file can provide. Data are accessed by name rather than by explicitly interpreting the low-level representation. Therefore, databases also provide high-level data semantics and hide the representation details from the user.

1.2. A Bit of History

The hierarchical data model and the network data model were the two major conceptual frameworks for organizing and manipulating databases before the 70's. They have a common drawback — there is no distinction of the conceptual view of data from its physical implementation. Records are used to model objects in the applications, and pointers implement relationships among the objects. The user is required to have thorough knowledge of the storage structure in order to navigate along the pointer chains to get to the right place, and to gain access to the data objects stored there.

In 1970, the relational view for database organization and manipulation was proposed [Codd70]. Major contributions of the relational model include a rather simplified user view of the databases and a powerful high level query formalism on the simplified view. The relational model is founded on an existing body of knowledge in modern mathematics (mathematical relation and formal logic.) A rich theory of the relational databases, ranging from logical design to query optimization, has since been developed (see [Maier83, Ullman83].)

The ANSI/X3/SPARC [ANSI75] specification on database management system structures is a significant event. The ANSI/X3/SPARC report proposed a three-level architecture standard. A *physical* schema specifies storage format for physical records and their inter-references. A *conceptual* schema defines logical structure of database independent of any application programs. And an *external* schema describes application-specific data formats. Two mappings are also proposed to achieve a high degree of data independence. The mapping between a conceptual schema and a physical schema makes it possible to change the physical storage format without affecting services available at the logical database design level. Similarly, the mapping between a conceptual schema and external schema permits changing logical databases independent of applications, by redefining the mapping.

There have been many successful implementations of the relational data model. Early experimental systems include System R of IBM San Jose [Astrahan76], INGRES of UC Berkeley [Stonebraker76], MRDS of Honeywell [Honeywell80] and PRTV of IBM United Kingdom [Schmidt83, Todd76].

1.3. Recent Research and Related Work

As the need for databases started to extend to non-business type applications, it became clear that the traditional database technology was neither effective nor flexible. A severe shortcoming of the relational model is that it is incapable of directly capturing the semantics of the application data. Research on semantic data models addressed the issue. Abrial is one of the early pioneers in semantic models [Abrial74]. His work combines different approaches from the following three research areas: generalized database management systems, the relational data model and artificial intelligence. The first-order logic was used as a unified framework. Integrity constraints were expressed as predicate calculus formulas. An algorithm was also designed to translate general formulas into computer programs to evaluate the formulas. Chen proposed the *entity-relationship* (ER) data model [Chen76], as an attempt at providing a unified view for different data models to facilitate logical database design. The ER model can be regarded as an early effort in searching for more expressive semantic data models. Kent did a thorough analysis on limitations and drawbacks of the traditional record-based data models [Kent79]. His study identified many problems of record-based information structures. In particular, he pointed out that the assumption of information homogeneity was not well founded. There are many instances where information has neither horizontal homogeneity nor vertical homogeneity. In such cases, record-based structures become clumsy and do not lead to elegant representation schemes at all. He demonstrated that *entities* and records did not correspond well, though the former were often modeled using the latter in record-based systems. The representation of relationships in such systems was also problematic. For example, we can expect such systems to answer the question: "in which department does the employee named 'John' work?" (e.g., Marketing); but not the question: "how are the department 'Marketing" and the employee 'John' related?" (e.g., the latter works in the former).

Many other semantic data models also emerged in late 70's and early 80's. For example, the semantic data model, SDM, by Hammer and Mcleod [Hammer81]; functional data models by Buneman and Frankel [Buneman79], and by Shipman [Shipman81]; formal mathematical models by Abiteboul and Hull [Abiteboul87], by Hull and Yap [Hull84] and by Brodie [Brodie82]; the semantic association model, SAM*, by Su [Su83]; the entity-based data model, Taxis, by Mylopoulos et at [Mylopoulos80]; object-oriented data models by Copeland and Maier [Copeland84], etc.

There have also been extensions to the relational data model to overcome its own deficiencies. Smith and Smith identified two useful abstraction mechanisms for database systems [Smith77], *aggregation* abstraction and *generalization* abstraction. They point out that the relational data model only provides the aggregation abstraction. They introduced a *generic* type into the relational model to support a generalization hierarchy. Codd himself also proposed an extended relational data model, RM/T [Codd79], in which he suggests that objects be given surrogates for reference and that an *E-relation* be defined for grouping objects of a common type. The design of POSTGRES [Stonebraker86], which is the successor of INGRES, also incorporates the notions of complex objects, user-defined data types and active semantics such as alerts and triggers.

As we have mentioned, the demand on databases in areas other than business data processing has brought new concepts and new requirements to database systems, giving new challenges as well as new excitement to database researchers. Katz described a data management facility for VLSI chip design [Katz83]. The object-oriented approach to engineering database management systems also received wide acceptance, since a central issue in CAD/CAM databases is complex object representation [Lorie83]. Batory and Kim studied modeling concepts for VLSI CAD objects [Batory85]. They suggest using *molecular objects* [Batory84] enhanced with the versioning concept, to model VLSI design objects. Kemper et al. demonstrated that both behavioral aspects and structural aspects of complex objects are valuable assets in engineering databases [Kemper87]. They showed how an behaviorally object-oriented system can be constructed on top of a structurally object-oriented system, using a non-first-normal-form relational data model. Other similar work has also been reported in the literature [Bancilhon85, Dittrich86, Ege87, Kim87, Stonebraker86].

More importantly, there have been successful implementations of object-oriented database management systems. Several representative systems are GemStone [Maier86, 86a], Ontos [Andrews87], ORION [Banerjee87, 87a], Encore/Observer [Zdonik85] and Iris [Derrett85, Fishman87]. GemStone uses a general purpose database programming language, OPAL, as its DDL and DML. The syntax of OPAL resembles that of the Smalltalk-80. It supports navigational access as well as associative retrieval. Indexing and clustering on objects are also supported for performance tuning. Ontos is an object manager using C++ as its host language. Persistence is supported through a library of system classes. A generic type is provided to facilitate the translation of memory objects and disk objects. Using the type "TRef", objects are activated (brought into memory) automatically when needed. However, the support for updates on persistent objects is minimal: The user has to traverse a complex objects, issuing update request on each subobject. ORION, Encore/Observer and Iris are all research prototypes, from MCC, Brown university and HP Laboratories, respectively.

Extensible data models take a different approach to cope with the complexities of non-traditional applications [Carey86, 86a, Graefe87, Lindsay87, Paul87]. The theme here is that since no single data model can meets all the requirements of the new applications domains simultaneously, a reasonable compromise is for the system developers to provide a small yet powerful set of basic building blocks that can be configured, in a flexible way and with minimal effort, to meet the needs of different applications.

In most general terms, the database technology can be extended to accommodate new concepts and new requirements from new applications in three ways. One is to start from a programming language and incorporate database notions such as persistence, concurrency and recovery to suit the needs of database applications [Atkinson83, 84, 87, Buneman86, Cockshott84, Copeland84]. In this instance, the problem of *impedance mismatch* (bulk data types versus individualized types, hidden iteration versus explicit loop control, etc. [Copeland84]) must be addressed. A second approach is to start from a database system and build into it a rich type system and other useful programming notions such as recursion and higher order functions, to make the database system have enough power to cope with complexities inherent in engineering environments [Shoens79, Stonebraker84, 86a, 87]. A third approach is to define new database programming languages from scratch. These database programming languages typically are rich in data structuring capability, versatile in data persistence and complete in computing power [Albano85, Cardelli84, Ohori89].

Object-oriented programming languages and logic programming languages are of particular interests to us. Object-oriented programming approaches, such as Smalltalk-80 [Goldberg83] and C++ [Stroustrup86], advocate the notion of cooperative independent computing agents. In this computing paradigm, each object has its own state and behavior. The state of each object is internal to the object, and is completely hidden from other objects in the system. Communication, through message sending, is the only way to make an object show its behavior or perform computations. Another distinctive feature of object-oriented systems is their classification mechanism. Objects with similar structure and behavior are grouped into classes, which are tied together by a class hierarchy with inheritance of behavior and structure from super-classes to subclasses. Object-oriented programming methodology embeds the following three approaches to computation: computing via state transition, computing via communication and computing via classification. Its modeling power and encapsulation ability make it an ideal basis for the next generation database management systems.

Deductive databases [Gallaire84, Kowalski78] using the paradigm of logic programming [Emden76, Lloyd84, Maier87], on the other hand, can be a natural evolution from relational databases, considering that in mathematics, relational systems are used as interpretation spaces for first-order logic. Founded on first-order predicate logic, this programming paradigm views computation as a deduction process, a process in which proofs are established using axioms and inference rules. The logic approach to databases has a number of advantages. First, it has a sound underlying theory, which is an asset that is hard to get for most software systems. Second, it provides two views on databases and its supporting language facilities. So we can talk about query processing as a theorem proving activity as well as model-constructing (-finding) activity, depending on which one is more convenient. Third, the language in first-order logic proof theory is richer than the counterpart of its relational semantic space. Deductive databases have an expressive power exceeding that of relationally complete query languages. For example, with recursive queries, transitive closures can easily be computed.

TEDM has its roots in both object-oriented programming paradigm and logic-programming paradigm. Like many other recent semantic data models, TEDM is aimed at engineering applications that need management of complex design objects. It attempts to achieve its goal by adapting successful features from both object-oriented programming and from logic programming, and by adding other extensions geared at bridging the impedance mismatch between the current database technology and application demands.

Other related work is briefly described below. In PS-Algol [Atkinson83], persistence is added as an orthogonal property to data objects in the Algol-68 programming language. Any data object has the right to persist. The mechanism to achieve object persistence is by designating a distinguished persistent object, the persistent root. Objects reachable from the persistent root are also persistent. A persistent object manager (POM) is used for automatic translation between virtual memory addresses and external addresses. TEDM uses a similar idea to support object access. A root object is supplied; all objects are place in collections, which are reachable from the database root.

Galileo [Albano85] is a conceptual language designed from scratch for describing database applications. Galileo is a strongly typed language. Furthermore, it permits a type generalization hierarchy, and type checking takes the type hierarchy into consideration. The generalization hierarchy of the type system in TEDM is closely modeled after that of Galileo. But we use a declarational type system instead of a structural one.

OPAL is the database language for the GemStone database management system [Copeland84, Maier86]. GemStone is based on object-oriented programming paradigm with many extensions, such as concurrency, security and recovery, to suit the need of data intensive applications. It supports encapsulation and inheritance, both on structure and behavior. Furthermore, it has a rather rich environment for quickly building database applications.

The E programming language [Richardson87] is the C++ programming language extended with database classes, file classes, generic access methods. Unlike other languages we mentioned so far, the intended users of E are database implementors, instead of application builders.

TEDM also benefited from the work in deductive database research [Gallaire84]. The logic approach provides an ultimate unified view on different database concepts. In particular, we can view information stored by a database as axioms of a logic system. The drawbacks of this approach are the following: A deductive paradigm based on pure first-order logic lacks a classification mechanism, which is useful both for conceptual modeling and for efficient organizations. There is also a mismatch between the theorem proving paradigm and the query answering mechanism, namely, while the former looks for proofs, the latter looks for answers.

Ait-Kaci's work [Ait-Kaci84] is a step forward in bridging the gap between the logic paradigm and the database paradigm. He integrated the logic programming framework with a type hierarchy. But his type system does not distinguish instances from schemas. Nevertheless, the Ψ-term in his system has great influence on the design for the object definition language of TEDM.

The work of putting QUEL commands into the INGRES database [Stonebraker84] bears certain similarities with TEDM's view of database commands as objects. But the similarities hold only at a superficial level. The two approaches are fundamentally different in the way how the stored commands are treated and what we can do to them. In INGRES, a QUEL command is stored as a textual string or its compiled form. In either case, there is no way to look at the subparts of the stored command, which essentially limits the kind of manipulations one can do with the stored commands, such as sharing of code and dynamic construction of commands.

1.4. TEDM Overview

This section is an overview of TEDM, a data model designed and described in [Maier85]. Other related documents are [Anderson86, 89, Ecklund87, Maier89, Ohkawa87, Zhu86, 88, 89]. Features from both object-oriented methodology and logic methodology are integrated into the data model. Several other extensions are also made to enhance the model. In particular, it embeds such notions as object identities, object classifications and structural inheritance, from object-oriented programming. It adapts, from logic programming, such notions as one-way unifications, inference rules and deductive query processing. Other extensions contributing to the power of the model include a compound-command definition and execution mechanism, a nested workspace model for variable scoping and a semantic extension to logic variables.

A "good" data model should be based on a small set of primitive concepts. On one hand, these primitive concepts should correspond to their real world counterparts well, such that databases built with the model display a minimal conceptual distortion of the application being modeled. On the other hand, provision for conceptual composition from primitives is needed, since much design activity involves conceptual composition in typical engineering applications. Although it is impossible to formalize what it means for a conceptual entity to model a real world entity without distortion, object-oriented programming methodology is well received as one of the best paradigms for this kind of modeling task.

Objects are the basic blocks for building TEDM databases. Conceptually, a database object represents a real world entity of some sort. Structurally, such an object resembles a nested-record structure that bottoms out at the so-called *simple* objects. There are subtle differences. Complex objects possess unique *object identities* (OIDs). Consequently, objects are referenced by identifiers rather than by values. Since the structure and the identity of an object are two orthogonal properties of its representation, the existence of unique object identity makes it is possible to distinguish objects without referring to or depending on their structures, which is important in CAD/CAM applications. For example, in a typical design of an electronic device, several IC chips with identical physical and electrical parameters may be needed. In this case, any such chips can be interchanged without affecting the behavior the circuit: none is distinguishable from the others by its own properties. But when the design is stored by a database, it is highly desirable to be able to distinguish them for such purposes as simulation, manufacture, and design change management.

Types are collections of objects, usually with similar or closely related structure. Several highlights of the TEDM type system are the following. First, *structural inheritance* allows subtypes to acquire structures from their supertypes. Second, *prescriptive typing* means an object conforms to a type if the object has all the structure specified by the type, but it may have more. Third, a distinction is made between intentions and extensions. That is to say, instances of a single type may participate in multiple collections, as opposed to just one relation as is the case in the relational data model. Finally, type memberships are separated from type conformities, which means object structure alone does not automatically assign type memberships to an object.

Database commands bring dynamics to databases. All traditional database access functions are fully supported by commands. These database commands in many ways resemble rules of a production system. A *pattern* and an *action* make up a command, with the semantics that the operations denoted by the action is performed on all objects that are retrieved by successful pattern matches against the database. This approach separates data retrieval from data viewing, and has the advantage that viewing operation and other database operations have uniform structure and semantics. It also separates sequential imperative part of a command from its non-sequential declarative part. In addition, database commands are stored as objects. They can be queried and manipulated in the same way as any other ordinary objects.

Rules enhance both the modeling power and the query processing power of the data model. Essentially, a rule specifies an assertion that the existence of certain objects (or internal structures of objects, or memberships of objects) can be deduced provided certain other objects (or other object structures, or other object memberships) exist also. With the presence of database rules, a database not only contains objects (object structures, object memberships) that have physical representations in the database, but also objects (object structures, object memberships) that are derivable with rules. These derivable data are considered *virtual* data. When coupled with query processing engine, database rules make the computation of transitive closures, and other recursive queries, possible.

1.5. Thesis Organization

This thesis is organized into 11 chapters. Chapter 2 describes an example database design using TEDM. The purpose there is to present most of the features of the data model informally, using an integrated example. The example application is taken from the domain of VLSI designs. We present a design database to maintain design information with a varying levels of abstraction. The database will allow design objects ranging from gate level to register-transfer level. Therefore, it is mostly suited for simulation support and similar tasks that are important in an iterative design cycle.

Chapter 3 discusses database objects, object definition languages and related topics. An object definition language (ODL) is proposed and its semantics are studied. ODL uses a first-order-term-like syntax as object constructors and adds to its interpretation space abstract objects, stored counterparts of pure placeholder variables. ODL also extends first-order term languages in a number of ways. It allows a variable number of arguments and variable ordering of arguments in terms. It splits the notion of variable into that of pure syntactical placeholders and that of symbols with abstract objects as interpretations. It also permits a variable to appear in internal node of a term graph to meet the need of specifying patterns sharing in complex structures.

Database types and a type definition language are discussed in Chapter 4. The chapter starts out with some comments on the notion of types in general, and TEDM's view of types in particular. A type definition language (TDL) is proposed in this chapter. The most important aspects of type semantics are characterized using four relations. We define and discuss each of the four relations in detail. The problem of representing types internally is also examined.

Chapter 5 discusses TEDM commands, command definition languages, command translation issues and query processing semantics. A command definition language (CDL) is defined and command definition semantics are explored. Syntactically, commands are similar to production rules of expert systems, having a conditional part and an action part. However, the analogy does not carry into semantics. In particular, the conditional part of a command is not just a simple predicate expecting a "true" or "false" truth-value as answer. It also has the power of pattern matching on databases and retrieving objects that contributed to a "true" truth value result. As such, the conditional part of a command is called a *pattern*.

Chapter 6 focuses on the abstract object extension to TEDM. We discuss the role of abstract objects as interpretations of variables in the command language. We attempt to make more precise the meaning of structure-matching in the semantical space. Our approach visualizes semantical objects as graphs and uses link-preserving functions that maps nodes of one graph to nodes of another graph. Two structures match if such a mapping can be found. Two kinds of pattern-matching, abstract-concrete and abstract-abstract, are studied. The chapter also describes how abstract objects are employed in the representations of type-defining objects and of command-defining objects.

Chapter 7 discusses the compound-command execution mechanism of TEDM. We study operational semantics of compound commands and describe implementation. We examine in detail TEDM's parameter passing scheme for the execution of compound commands.

Chapter 8 describes a prototype implementation of TEDM. The prototype is a main-memory based implementation written largely in Prolog. We briefly discuss overall program organization and each major functional component and its realization.

Chapter 9 proposes extending the data model with computational objects. We review basic functional language implementation techniques. We are particularly interested in graph reduction techniques and an abstract graph-reduction machine, the G-machine. We point out that the it is possible to incorporate the mechanics of graph reduction and TEDM's command processing engine, and investigate the semantics of mixing computations with pattern-matching.

Chapter 10 describes the data derivation mechanism in TEDM. Syntactically, *rules* extend a combination of the object definition language and the command language. They are well formed formulas in this extended formalism and form a basis for deriving logical consequences of the database. From a proof-theoretic viewpoint, derivable data are not very different from stored data, both of them are theorems in the system. On the other hand, using a model-theoretical viewpoint, we can think of a database as a closure of stored models, with respect to all well formed query formulas known to the system. In terms of operational semantics, rules can be thought of as deferred database commands.

Chapter 11 contains general discussions on future directions, some concluding remarks and a summary of the dissertation.

CHAPTER 2

AN EXAMPLE APPLICATION

This chapter introduces the TEDM data model through a series of examples. The illustrative examples are all based on a schema defined in TEDM for an electronic CAD database, and emphasize the conceptual modeling aspects of the data model. In this application, electronic design objects and information about their inter-connections are stored as a TEDM database. When these electronic component objects are composed using connection objects, or wires, larger circuits with various functionalities can be formed. It should be noted that this is only a paper exercise; the result of the design has not been tested.

This chapter is organized as follows. Section 1 reviews general issues and problems in the area of CAD/CAM research. Section 2 describes an example database intended for an electronic CAD application. It explains general ideas in the CAD database design and representation of objects in this particular application domain. Section 3 contains a schema design for the CAD database. Each type is explained and is given a type definition in TEDM's type definition language. Section 4 shows how to represent digital designs using the CAD database; we will build a sequential digital circuit, a four-bit-adder, using components such as logical gates and registers. A fully populated design instance of the four-bit-adder is included as an appendix. Section 5 introduces TEDM's database commands and explains basic ideas behind command processing, with a number of examples for querying and manipulating the CAD database.

One of the distinguishing features of the TEDM model is that it is not tied to any single concrete syntax. While there are default languages for objects, types and commands, the system architecture makes it easy to provide alternative syntax to describe the same thing or a subset of it. We can freely introduce specially tailored syntax suited for particular application domains. However, the underlying object representation will remain the same. This idea will be expounded in future chapters, and in the appendix for this example application.

2.1. Fundamentals of CAD/CAM

We review issues and problems in the area of design support for electronic circuit designs. The goal of such a design effort is to produce circuit schematic that meets various requirements, such as I/O behavior, timing constraints, power dissipation, etc., given by a specification. Of the four alternative ways to verify whether a design satisfies a specification, hand execution (using paper and pencil), program simulation (using software and computer), hardware prototyping (actually making a chip) and automated theorem-proving techniques (using formal specification and verification), program simulation has been most widely used. The method of hand execution is perhaps no longer a valid alternative for human engineers, due to the enormous circuitry complexity of today's VLSI technology. Hardware prototyping, on the other hand, is likely the least appealing to most people, simply because the cost involved does not measure up with the value of the prototype products. Hardware prototyping is not conclusive to mixed-mode text-and-fix development, as are the others. One hardware bug may mask other bugs, meaning many prototypes. Although the formal approach is appealing, in that the correctness of designs can be verified mechanically using a theorem prover, the required computation cost prohibits its application to designs of the size of the current generation of processors. People are still actively seeking efficient formalisms for design specification and verification.

Thus, a typical VLSI design process consists of an initial design followed by one or more iterations between design simulation and design change. In this context, a design is "conveniently" represented as a program in some high-level language (HLL), such as the C programming language with BitSim, a simulation environment based on C [Hammerstrom86]. Given this correspondence, an initial design becomes the initial coding of a program, design simulation is the execution of the program, while design change corresponds to program modification, as shown in Figure 2.1.

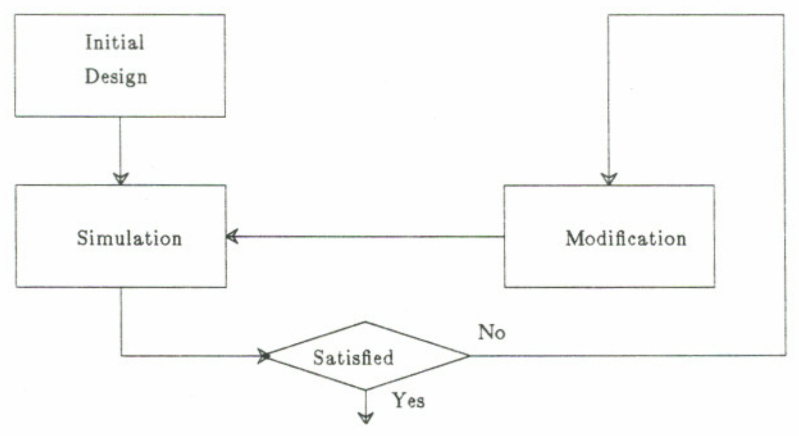

Figure 2.1 A Design Process

While language representation for designs is good for simulation, it is hard to maintain changes, since design objects and connections are captured implicitly in expressions and shared variables. On the other hand, simulating off of structures is not efficient. A good middle ground is to use a database to represent designs, and generated HLL simulators from the database representation for the designs. A good database language would also help considerably the task of generating simulators.

Given a piece of circuit, as shown in Figure 2.2 (a), the following code segment (in C notation) may be a reasonable program representation (a *design program*) of it:

```
D = ~ A;
if (CLK) Q = D;
```

Suppose we change the design to Figure 2.2 (b), then the code segment accordingly becomes

```
D = ~ (A & B);
if (CLK) Q = D;
```

This example illustrates how changes in the design dictate the changes in the simulation code.

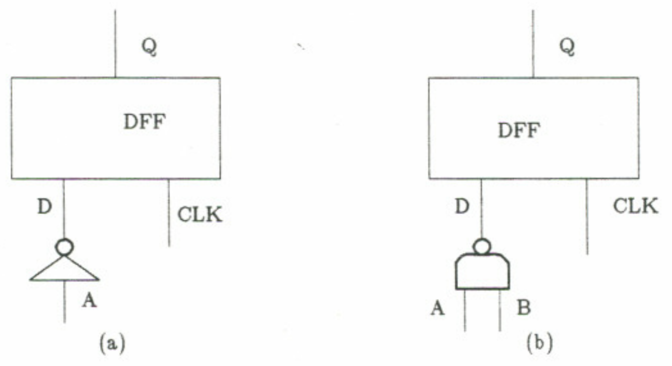

Figure 2.2 Two Circuit Elements

HLLs support abstractions for computation, but not for design simulation or description. For example, hardware description languages (HDLs) have abstractions that are more domain specific. Although programs representing simple designs can easily be constructed with maintainable complexity, it is entirely a different story for complex design tasks. The complexity of a large design can grow quickly, to the point where the HLL is no longer a tool but a burden to the hardware designer. In parallel to using general purpose HLL, many specialized design description languages have been used. These hardware description languages (HDLs) are tailored specifically for the application domain of electronic circuit designs. They typically include linguistic constructs meaningful in this domain. For example, instead of saying

D = ˜ A,

we may use

signal A Inverter **to signal** D

Similarly we can use

on signal CLK **signal** D D-flip-flop **to signal** Q

to replace

if (CLK) Q = D;

Although HDLs are an improvement, they still cannot cope with the ever-changing hardware technology. Also, connections are still via variables; and it is not easy to find all uses of "Q" in the program, for example. New alternatives, such as database support, are still needed. The primary goal of CAD/CAM databases is to describe and represent hardware designs and as database objects and to computerize the design and manufacturing processes.

CAD/CAM databases emerged to complement the HDL's to cope with the complexity in hardware design. As design tasks becomes more and more complex, HDLs alone as the tools for design description become not only inefficient, they also create new difficulties. One possible explanation is the following. HDLs are no more than formal languages that can be utilized to describe circuit components and their interconnections. Hardware designs expressed with an HDL are analogous to source programs written in an HLL. They are compiled into different representations, depending on tools involved and design aspects studied by the different phases in a design process. There are also problems in this approach. For example, a layout tool understands a layout description of a chip, but a simulation description of the same chip may totally be foreign to the layout tool. It is also very hard to keep connections between different abstraction layers.

Recent CAD/CAM research calls for a shared design database to meet the new challenges in design and manufacturing applications. The premise is that if all design tools are built from a common ground and rely on a central standard information repository, we would have a better chance to make the tools understand each other. As such they can work in step and cooperate toward solving problems, rather than creating problems. We would also have a better chance to cut the cost of managing changes in design processes. Finally, database support can open up new possibilities for design tool generators, too. For example, a hardware design stored in a database can be the input to a simulator generator, that produces the code for a functional simulator of the design, as opposed to maintaining the functional design as simulator code.

2.2. An Example Application

The example application we choose is from the domain of electronic circuit design in general, and VLSI design in particular. The goal of the database design is a schema for representing VLSI design objects, mostly at the level of the register-transfer logic.

In this application, a design is a complex object containing component objects and information about interconnections among components. The database should be able to store various kinds of design component objects and connection (or conducting wire) objects, and compositions of these objects that represent larger circuits with certain prescribed functionalities.

Even though the primary motivation of this example is to introduce and illustrate basic data modeling capabilities of TEDM, our design nevertheless is sufficiently general and can be used as a basic framework for real CAD/CAM applications. For example, one possible extension is to include code segments with design component objects that simulate their behavior. Then, a code extractor can be constructed as a database application, which traverses circuit objects and collects code segments to produce simulation programs for the circuits. In other words, a simulator generator can be built on top of this design database. Another possibility for extension is that formal functions can be incorporated into the schema that describe the formal behavior of the design components, and a design verifier can then automatically derive or verify the functional specification of the resulting circuits.

In designing a schema for the example CAD application, we take into consideration support for modularized, hierarchical design methodology, and permit design objects at different abstraction levels to coexist in a single design project. In our view, electronic design objects are database representations of detailed plans for constructing VLSI circuits. Such an object describes the circuit components needed to manufacture a chip, as well as their interconnection topology. Thus, design objects represent circuit schematics. A complete design is constructed from smaller building blocks — design components or cells. Primitive components and pre-existing designs can form design libraries, and objects in design libraries appear in new designs to avoid repeated efforts and reduce complexity. Depending on the nature of a design task, libraries of different abstraction levels can be selected, permitting the same device to be viewed differently.

Once a design is completed, it is available for reuse in the form of library objects, Therefore, a design activity can also be viewed as an effort to make new library cells. To support this library paradigm, a design specification must be equipped with two distinct views, an external view and an internal view. The external view is the interface to the outside world and has ports for connecting to other circuit components. The internal view is the implementation, which details internal configuration and realizes the external (interface) behavior of the new cell.

2.3. Schema Design for A CAD Database

The format of the design objects can be understood by looking at the schema of the CAD database. In general, a schema is a collection of type definitions. Typically, a TEDM type definition is given as a list of structures, field names and corresponding field types. Each type defines one kind of object that is permitted in the database. Thus, the schema for a design is a conceptual abstraction of the design entities.

There are nine types in the schema for the CAD application. They are "Design", "Library", "Cell", "Port", "ImplementedCell", "Schematic", "Component", "Connection" and "PortRef".

The Type "Design"

"Design" objects are the top level objects, representing logical plans for realizing functional circuits. A "Design" object is the goal that a design activity wishes to achieve. In order to describe a design efficiently, "Design" objects make use of available libraries, instead of always starting from primitive components, such as FETs. For example, if certain design requires a one-bit half adder, it may directly get the adder as an existing component from some library, instead of using four nand gates to construct the adder circuit explicitly. The detailed circuit connectivity is represented by objects of type "ImplementedCell". Thus, we have the following TEDM type definition for "Design".

$$\begin{aligned}\text{Design} = (&\text{usesLibrary} \twoheadrightarrow \text{Library}\\ &\text{implementation} \rightarrow \text{ImplementedCell})\end{aligned}$$

The definition says that a "Design" object has two kinds of fields, those labeled by "usesLibrary" and taking values from "Library" objects, and exactly one labeled by "implementation" and taking its value from "ImplementedCell" objects. The notation "$\rightarrow$" indicates a *single-valued* field, a field permitting exactly one value, and the notation "$\twoheadrightarrow$" denotes a *multiple-occurrence* field, a filed permitting zero or more values.

The Type "Library"

Information kept by a design "Library" object includes a name, "libraryName", for identifying the library, and a type, "libraryType", indicating whether the library is global (standard) or local (nonstandard). The third component of a "Library" object is a multiple-occurrence field "libraryCell", which keeps a list of known devices available in the library. This type definition is given below:

$$\begin{aligned}\text{Library} = (&\text{libraryName} \rightarrow \text{String},\\ &\text{libraryType} \rightarrow \text{String},\\ &\text{libraryCell} \rightarrow\!\!\!\rightarrow \text{Cell})\end{aligned}$$

The type for both "libraryName" field and "libraryType" field is the same, "String", which is a *primitive* type. Primitive types are provided by system and cannot be redefined. We assume using two string valued objects, "global" and "local" as the values for the "libraryType" field. The objects of the field "libraryCell" are of the type "Cell". These library "Cell" objects represent electronic components that are directly available to the designers to make new circuits.

The Type "Cell"

A "Cell" object represents digital circuitry with well defined functionality, such as various logic gates, flip-flops, etc. Each such object has a name, "cellName", for external reference, and a multiple occurrence field, "cellPort", which maintains information about the cell interface to the outside world. The "Cell" is defined by the following type definition.

$$\begin{aligned}\text{Cell} = (&\text{cellName} \rightarrow \text{String},\\ &\text{cellPort} \rightarrow\!\!\!\rightarrow \text{Port})\end{aligned}$$

At this level, a "Cell" object is pretty much a black box with a certain I/O function. "Cell" objects only contain interface information, a list of connection points through which the outside world can make use of the well defined cell function. We assume library cells are correctly implemented, and are not concerned at all how they are implemented.

The Type "Port"

A "Port" object represents a connection point, or a pin, of an electronic component. It has a name field, "portName", for the purpose of pin identification, and a direction field, "portDirection", indicating the direction of information flow on the pin. The type definition is thus

$$\begin{aligned}\text{Port} = (&\text{portName} \rightarrow \text{String},\\ &\text{portDirection} \rightarrow \text{String})\end{aligned}$$

Note in practice, pins are usually identified using numbers, such as pin 1, pin 2, etc. We use the type "String" instead.

The Type "ImplementedCell"

The next type definition uses a new notation — a type name followed by a structure list. This notation relates a newly defined type to one or more existing types by a subtype-supertype relationship. The new type is defined as a subtype of the type(s) preceding the colon. A subtype inherits all the structures of its supertype(s).

$$\text{ImplementedCell} = \text{Cell:}(\text{internal} \rightarrow \text{Schematic})$$

Thus, "ImplementedCell" is a subtype of "Cell". As such, an "ImplementedCell" object will also have a name, and a number of connecting ports just like a "Cell" object. In addition, an "ImplementedCell" object is also required to have a "Schematic" object to provide internally detailed implementation information, such as components needed and connection topology among the components. Also, because the type "ImplementedCell" is a subtype of the type "Cell", an ImplementedCell object can be used where a Cell object is expected.

The Type "Schematic"

Detailed design implementation information is completely described by a part list, which are the components needed to make a new device, and a connection list, which are wires for connecting the components, as is reflected in the following definition.

$$\begin{aligned}\text{Schematic} = (&\text{part} \twoheadrightarrow \text{Component},\\ &\text{wire} \twoheadrightarrow \text{Connection})\end{aligned}$$

The Type "Component"

A "Component" object represents an instance of a physical circuit element used in the design to achieve certain electronic behavior. It is a logic symbol appearing in a schematic, or a component with physical existence required when the design is actually to be manufactured.

$$\begin{aligned}\text{Component} = (&\text{compoName} \rightarrow \text{String},\\ &\text{cell} \rightarrow \text{Cell},\\ &\text{library} \rightarrow \text{Library})\end{aligned}$$

The information in a "Component" object is basically for identification purposes, that is, where to find, when needed, and further descriptions of a component. Hence it has three fields, a "compoName", a "cell" and a "library". In practice, component names correspond to the component numbering in a schematic.

The Type "Connection"

An object of type "Connection" represent a single electrical connection point in the schematic. It is these objects that group together various components to perform a given digital function. The definition for "Connection" is:

$$\text{Connection} = (\text{port} \twoheadrightarrow \text{PortRef})$$

Thus, a "Connection" object consists of a list of port references (of type "PortRef"). a connection is made by tying all cell ports that participate in the connection to the same place. Again, we may use ports instead of port references in the definition.

The Type "PortRef"

A "PortRef" object represents a single wire originating from a port of a cell instance of the type "Component". "PortRef" objects are used by the "Connection" objects. Notice, in our model, that objects of type "Cell" or type "Port" are used as templates, and that designs use instances of these templates. The definition of "PortRef" is the following:

```
PortRef = (port → Port,
           component → Component)
```

Obviously, using the "port" of a cell instance and the "component" of the instance, we can uniquely identify to which circuit component instance this port belongs. There would be ambiguities had we used component names and port names, as two components may have the same name.

We should mention that not all type definition facilities available in TEDM are demonstrated by this example application. There are places in the schema where finite enumeration types would be more appropriate. We use a minimal TEDM model, which only supports "String" and "Integer" as primitive types, for simplicity.

2.4. Example Objects in the CAD Database

This section provides example objects for the CAD database schema. The examples will be drawn from a simple circuit design for a four-bit adder. A complete object representation in the CAD database for this design is included as Appendix I. The logic diagram for the four-bit adder is shown below (Figure 2.3). The logic diagram is constructed by the author from a BitSim program provided by Dan Hammerstrom [Hammerstrom86].

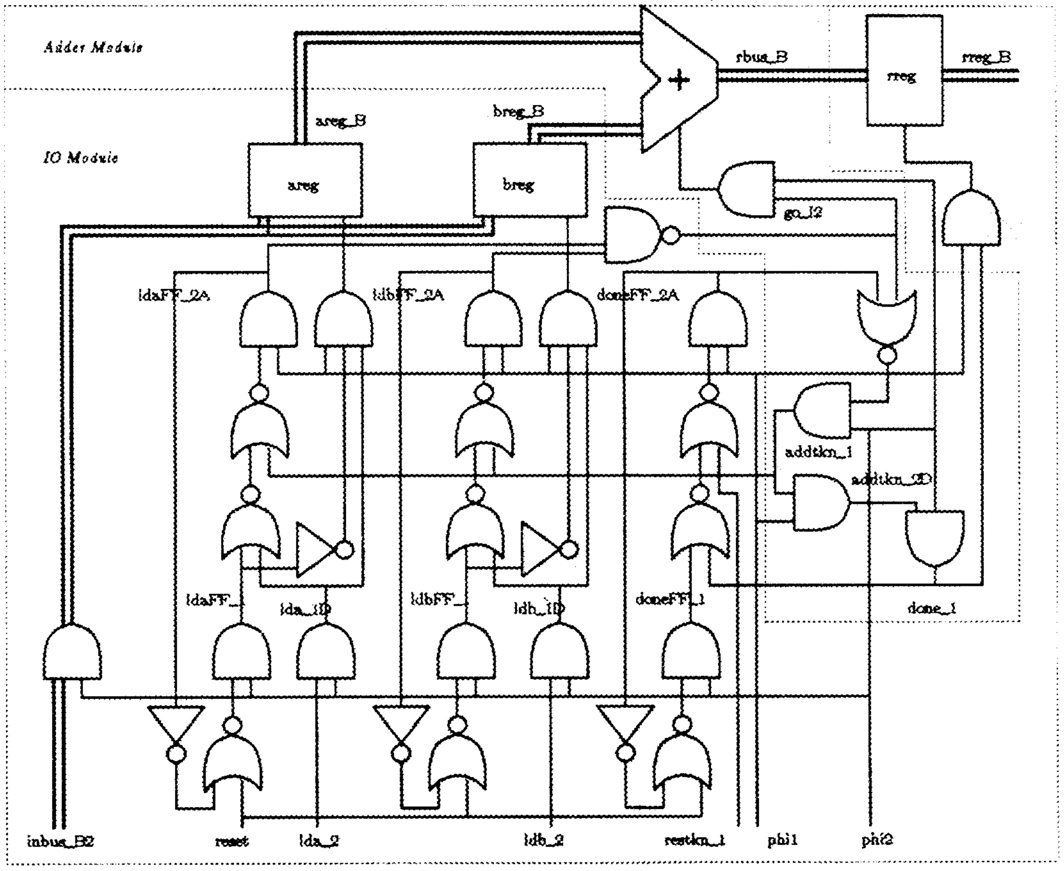

Figure 2.3 A Design Object for A CAD Database

In the diagram, double bold lines represent a data bus of width 4, which is simply captured here as 4 wires, though other alternatives exist. For example, we could add a "width" field in the schema for objects of type "Port". The rest are all standard logical symbols. This design uses the two-phased logic, with clocking signals labeled "phi1" and "phi2", respectively. Operands are loaded into two registers, "areg" and "breg", from input bus "inbus_B2" and controlled by "lda_2" and "ldb_2". The add operation is done by a four-bit full adder, and the result is saved in result register "rreg".

Example 1: An object representing a two-input and-gate of a design library:

```
Cell:(cellName → 'and_2',
      cellPort → Port:(portName → 'pin_1',
                       portDirection → 'input'),
      cellPort → Port:(portName → 'pin_2',
                       portDirection → 'input'),
      cellPort → Port:(portName → 'pin_3',
                       portDirection → 'output'))
```

□

While the examples we show here may seem verbose, we are not restricted to any specific surface syntax, as we pointed out. We will introduce application-specific syntax in Appendix I, where a complete design instance is presented.

Notice how a multiple occurrence field occurs in the previous example, where "cellPort" repeated three times, but with distinct object values. A short hand for multiple occurrence field is possible, as is shown in the next example.

Example 2: An object representing a two-input nand-gate:

```
Cell:(cellName → 'nand_2',
      cellPort → Port:(portName → 'pin_1',
                       portDirection → 'input')
               & Port:(portName → 'pin_2',
                       portDirection → 'input')
               & Port:(portName → 'pin_1',
                       portDirection → 'output'))
```

□

Instead of repeating field labels multiple times, an ampersand symbol "&" is used as a connective for values of multiple occurrence fields. The next example is for a D register cell.

Example 3: An object representing a D register:

```
Cell:(cellName → 'd_register',
     cellPort → Port:(portName → 'pin_1',
                      portDirection → 'input'),
               & Port:(portName → 'clock',
                      portDirection → 'input'),
               & Port:(portName → 'pin_2',
                      portDirection → 'output'))
```

□

The example cells we have seen so far are all standard library cells. They only have external interface descriptions, which are sufficient to make use of them. The functionality of the standard cells can be derived from their internal implementations, or be given as Boolean equations for primitive components. Note that in the BitSim representation, the functionality is implicit in the expressions of the program. In general, the following places are likely candidates for keeping this information: people's heads, simulation programs, additional fields in cells and other parts of the DB.

The next example shows a nonstandard cell with implementation information. The example illustrate the concept of *object identities*. In TEDM, every object has an identifier, called the identity of the object. Object identities are uniquely assigned to the objects on their creation. No two objects ever have the same identity. We use symbolic names, such as "AND_2", "anAnd_2_1", "anAnd_2_2", etc., to represent object identifiers.

Example 4: This object represents a method for making a four-input and gate out of three two-input and gates. Notice how placeholders are used in this example to indicate subobject sharing.

```
ImplementedCell:
 (cellName → 'and_4',
  cellPort → Port:(portName → 'pin_1',
                   portDirection → 'input'),
           & Port:(portName → 'pin_2',
                   portDirection → 'input'),
           & Port:(portName → 'pin_3',
                   portDirection → 'input'),
           & Port:(portName → 'pin_4',
                   portDirection → 'input'),
           & Port:(portName → 'pin_5',
                   portDirection → 'output'),
  internal → Schematic:
   (part → Component: anAnd_2_1
     (compoName → 'and_2_1',
      cell → AND_2',
      library → DefaultLib)
           & Component: anAnd_2_2
     (compoName → 'and_2_2',
      cell → AND_2,
      library → DefaultLib)
           & Component: anAnd_2_3
     (compoName → 'and_2_3',
      cell → AND_2,
      library → DefaultLib))
    wire → Connection:
     (port → PortRef:(port → aPin_3,
                      component → anAnd_2_1)
            & PortRef:(portName → aPin_1,
                      component → anAnd_2_3))
           & Connection:
     (port → PortRef:(portName → aPin_3,
                      component → anAnd_2_2)
            & PortRef:(portName → aPin_2,
                      component → anAnd_2_3)))))
```

□

In order to save space, we omitted five wires in the last example, which are used to establish the correspondence between the ports of the new cell, a four-input and gate, and the ports of its components, three two-input and gates.

2.5. Example Queries on the CAD Database

Finally, we furnish a number of example queries on the CAD database to finish our introduction to TEDM.

Example 5: This command finds and displays all ports that a four-bit adder has.

$$\text{View}[P] \Longleftarrow \text{cells} \rightarrow \text{Cell:C(cellName} \rightarrow \text{`adder4', cellPort} \rightarrow \text{Port:P)}$$

□

The construct on the right-hand side of the symbol "<==" forms a *pattern* or *template*. The construct "cells →" on the right-hand side of the symbol "<==" means a collection of database objects, reachable from the database root. Here "cells" refers to a collection of "Cell" objects, which serves the purpose of defining a starting point or scope for matching against databases. Conceptually, collections separate intension from extension, in that there can be multiple collections over a type. The command processor searches through databases and uses the pattern in the command to match objects on the "cells" collection. The objects that match the pattern are retrieved and are assigned as values of the variables in the pattern, which are then processed by operations indicated by the construct to the left hand side of the arrow symbol. In this particular example, all port objects of cells named "adder4" are retrieved by the pattern and are displayed by the "View" command procedure.

Example 6: This command creates a new library of Cell objects for each ImplementedCell object. The cells in the library object are those cells used in the implementation of the ImplementedCell object.

```
library → Library:*(libraryName → N,
                    libraryType → 'global',
                    libraryCell →→ C) <==
        cellUsed → ImplementedCell:(cellName → N,
                                    internal → Schematic:(part →
                                        Component:(cell → C))),
        cells → C
```

□

The asterisk "*" in the last command indicates an object creation operation. In this case, objects of type "Port" are created. The newly created objects all have the same internal structure. Nevertheless, they are distinct objects since they have unique identities. Note also that the notation →→ is used to collect all objects bound to the variable "C" in a multiple-occurrence field.

Example 7: This command adds "neighbor" field to Component objects. The values of this field of a Component object are the Component objects that are directly connected to the object.

```
C(neighbor → C1) <== connections →
        Connection:(port → PortRef:(component → Component:C),
                         & PortRef:(component → Component:C1))
```

□

2.6. Chapter Summary

The basic modeling aspects of TEDM and its command language are introduced in this chapter through an example application. The CAD database example provides a good exposition to the data model, although it only uses a limited number of constructs of TEDM. More thorough discussions are given in the next few chapters.

CHAPTER 3

DEFINING OBJECTS

The language aspects for describing objects are studied in this chapter. Although we define different languages, our ultimate goal is to do away with these auxiliary languages. In the end, we would use a single uniform (or canonical) language for all purposes, such as data definition and data manipulation. How can such a canonical language be possible? One approach would be to define the canonical language as the union of all the auxiliary languages whose functionalities the resultant language intends to incorporate. The union method solves the problem, but in a clumsy way. It also means that future inclusion of a new language by the data model amounts to redefining the canonical language by one more union component. In other words, the canonical language obtained by the union method can not easily be extended. We approach the problem in a different way: We do not rely on inclusion of syntactic constructs from different languages for obtaining a language with more power. Instead, we do so by extending the databases, to include objects that are semantically richer than application data. Also, we only require a minimal syntactic extension to the object language (when compared to similar languages in other systems).

The result of this convergence in language is a total objectification in the following sense: The object language becomes the canonical language, whose sentences denote database objects. The expressions of the other languages become sentences of this canonical language. For example, the language for defining types and the language for defining commands are all sublanguages of the canonical object language. They denote special classes of database objects, namely, type-defining objects and command-defining objects. It is also convenient to add new sublanguages, for example, for describing displays. The new language would be similarly interpreted as database objects. Although the new language may have a special syntax, its semantics remain unchanged as database objects, and can be described by the canonical language. Consequently, every sentence in the language has an object representation. And conversely, every database object can be described by a sentence.

We will make a distinction between an *abstract* object and a *concrete* object. Concrete objects are objects with identities and structures. They are the basic building blocks for application modeling. From the database user's perspective, an object is a conceptual representation of certain aspects of a real world entity from an underlying application domain. From the system's perspective, the representation of a database object consists of an object identity and a collection of fields. Abstract objects are a different class of objects. Their presence improves the data model in its ability to represent things other than application data. They have special semantics, which form the basis for representing programs as data, and the basis for a canonical data description language.

Abstract objects are used to represent patterns in commands. Their function is similar to that of variables in logic formulas; they are templates that can match concrete objects. Structurally, an abstract object also consists of an object identity and a collection of fields. The major differences between abstract objects and concrete objects lie in their semantic connotations. The system treats them as fundamentally different categories of objects, and is able to distinguish them by their representations.

This chapter contains four sections. Section 1 discusses motivations and the rational for extending object spaces. It discusses the requirements and the consequences of a total objectification. Section 2 presents related syntactic issues. It suggests splitting the meta-concept *variable* into two parts: one being *placeholders* and the other being *object tags*. The former continue to be meta-entities that range over individuals of the object spaces. The latter (object tags), however, become non-logical symbols that require interpretations. A syntax for an object definition language (the canonical language for TEDM) is also proposed. Section 3 associates meanings with expressions of the object language. Section 4 provides informal discussions for a few of the additional features of the object definition language.

3.1. A Canonical Data Description Language

We elaborate on what we mean by the term *total objectification.* A data model is *representationally complete* if it provides a unified view and representation for application data, their definition, manipulation, inference and displaying. If an object-oriented data model is representation complete, we say that it is totally objectified. We consider the definition, manipulation, inference and displaying of data as the most important activities for database management systems, from an end user's perspective. They are singled out in this thesis as the criteria for the informal notion of representation completeness. But others (such as constraints and access restrictions on objects) can be added to this criterion list, too.

With this notion of totally objectification, data definition, manipulation, inference and displaying all become part of an application database. They are all represented uniformly as objects. Consequently, they can be queried and manipulated, in the same way as the application data can, by the end users or the application developers, using the tools and facilities provided by the database management system. Special predefined types are provided for representing data definition, manipulation, inference and displaying as objects. With this approach, types and commands can be manipulated in the same way other objects are manipulated. We do not exclude the possibility of imposing certain restrictions on how these special classes of objects can be manipulated. For example, we may restrict updating on data definition objects to control the complexity in schema evolution.

For each category of these special objects, TEDM provides a specification language for the users' convenience. In particular, we have a language for creating objects, a language for defining types, a language for querying and manipulating existing data, and a language for deducing information. We anticipate that other languages would be added in the future. For example, a language for display format specification for objects, as proposed in [Anderson86, Flynn88]. But as we pointed out, the language for object definition (creation) will subsume the capabilities of the other languages combined. In other words, although on the surface we provide different languages to describe different categories of database objects, only the object definition language is fundamentally indispensable. The remaining ones are syntactical variations and are supplied for the users' benefit. With this canonical language, we are able to tailor individual interface languages to best suit special needs. Hence, the system provides *tailorable language interfaces* in a clean way: Addition of a dialect amounts to introducing a simple dialect for the object definition language, whose semantics are already understood by the current system. (Of course, the introduction of a new language with new semantics would require new evaluation modules to be added.) Furthermore, new language dialect can be added in a fairly modularized fashion. We can also define multiple dialects for certain special objects. For example, either a textual language or a graphical language or both can be used for defining compound commands, or for specifying display formats.

Objects and Their Structure

There are two generic classes of objects, *simple* objects and *complex* objects. Simple objects are *atomic*, namely, cannot be further decomposed. Simple objects have no internal structure and are *immutable*: One cannot change the state or value of simple objects. But when simple objects occur as part of complex objects, they can be replaced by other simple objects of a similar type, say as the result of an update operation on the complex objects.

Example 8: *String*, *Integer* and *Real* are a few typical simple object types. *'Oregon Graduate Institute'*, *'Beaverton'*, *19600*, *1989*, *5.5*, etc. are examples of simple objects. The notion of types is discussed in detail in the next chapter.
□

Complex objects, on the other hand, has unique identities, possess structures, and are classified into different types. A complex object is a composite entity consisting of one or more fields. The values of object fields are themselves objects, simple or complex.

Example 9: A Point object has an x-coordinate and a y-coordinate, which are Integer objects. P1 and P2 *represent* the unique identifiers that are assigned to the two point object, respectively (but the actual identifier values are system generated and not visible external to the system).

$$\text{Point:P1}(x \rightarrow 2,\ y \rightarrow 2)$$
$$\text{Point:P2}(x \rightarrow 4,\ y \rightarrow 4)$$

A DirectedLine object has a start point and an end point, which are Point objects: DirectLine:L(startPoint → Point:P1, endPoint → Point:P2). □

Complex objects need not always be tree-structured, as in the last example. Since objects have unique identities, structure sharing is easily supported.

Example 10: When two directed lines have a common start point, an angle is formed (This example uses objects from the previous example):

```
Angle:A(startEdge → DirectedLine:L
        endEdge → DirectedLine:L2(startPoint → Point:P1
                                  endPoint → Point:P3(x → 2,
                                                      y → 4)))
```

□

Similarly, cycles in the object structure graphs are also possible. For example, in an organization database, a department has a chairperson who works in the department. Figure 3.1 is an illustration of the angle object described in Example 10. The components, lines and points, are labeled by their symbolic object identifiers.

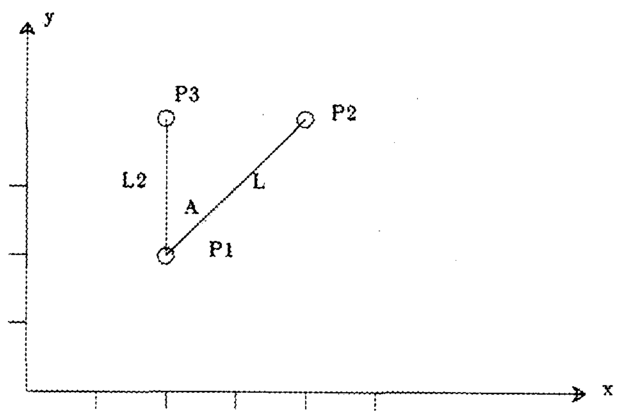

Figure 3.1 A Complex Object

In summary, objects in TEDM are structurally homomorphic to (can be mapped into) directed graphs. The results of the mapping are *object structure graphs*. The nodes in an object structure graph represent objects, and are labeled by object identities (simple object identities coincide with their values). The arcs represent the field structure of objects, and are labeled by field names. The mapping starts from simple objects. (If there are no simple objects, pick an arbitrary complex object to start.) Simple objects are mapped to isolated nodes. Consider each already mapped node in turn, if the corresponding object occurs in another object as a field value, and there is no arc labeled with field name connecting the two nodes (if the second object is not mapped, create a node for it first), add such an arc from the node for the second to the node for the first. This procedure results in a directed graph, with labeled nodes (by object identities) and labeled arcs (by field names).

Describing Complex Structures

Convenient notations exist in the mathematics for describing complex structures. Examples include the following:

1) The set notation: SET. SET is the basis for studying graphs abstractly. A directed graph can be described as a pair of sets, a set of nodes, and a subset of the self-product of the node set as the set of edges in the graph. For example, the set pair

$$S = (\{ a, b, c, d \}, \{ (a, b), (a, c), (b, c), (b, d), (d, a) \})$$

denotes the graph in Figure 3.2.

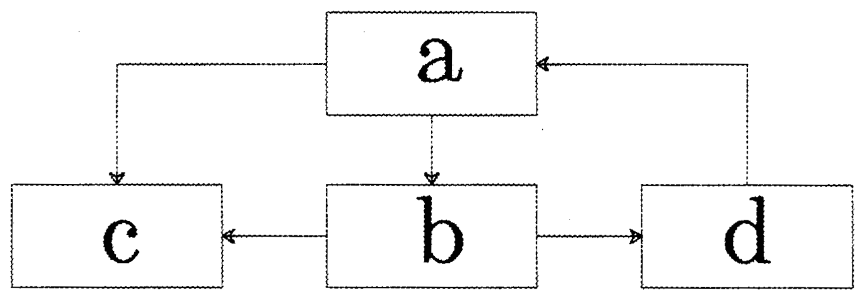

Figure 3.2 A Directed Graph

2) Variable-free terms: TERM. The TERM notation is equivalent to *rooted directed trees*, directed acyclic connected graphs with two properties: a) The number of edges is one less than the number of nodes; and b) there is a unique distinguished node (the root) that has no incoming edges. In formal logic, terms are an important language construct. Ground terms, terms without variables, denote precisely

the class of graphs of the rooted directed trees. For example, the ground term,

a(b, c(d, e))

represents the tree in Figure 3.3

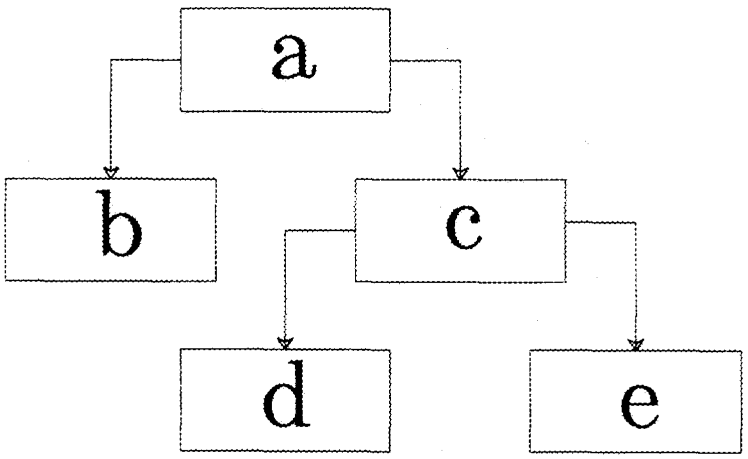

Figure 3.3 A Rooted Tree

3). Terms with variables: TERM/VAR. This notation is an immediate extension of TERM. Although variables do not have meaning by themselves alone, they give rise to a class of formulas, quantified formulas, and their meaningful interpretations. Constants are substituted for variables before the truth value of a formula is determined. Such substitutions are required to be consistent. That is, they must be functions from variables to constants. Interestingly, this requirement leads to a desirable extension to the graphical interpretations for TERM: TERM/VAR effectively represents rooted directed acyclic graphs (see Figure 3.4). For example, the term

a(X, c(d, X))

describes the following DAG.

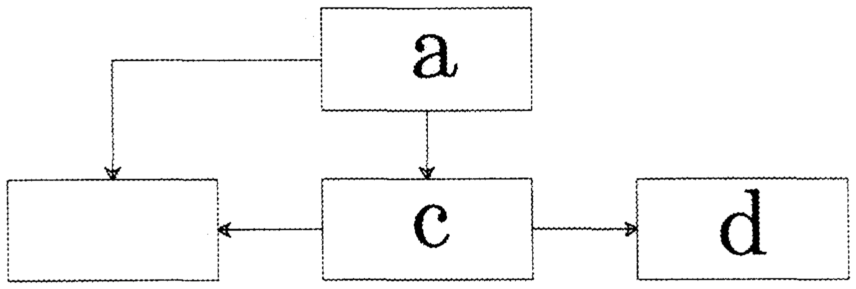

Figure 3.4 A Directed Acyclic Graph

Unfortunately, none of the three notations provides a complete solution to our quest for a powerful specification language for object structure diagrams. The paragraphs to follow discuss why we need more.

As is evident from the last section, when their semantics are ignored, object structure graphs are precisely rooted directed graphs. But when taking semantic information into consideration, we find that not all such general graphs are well-formed object structure graphs. For example, if we consider the fact that objects are typed, then possible edges emanating from a given node are restricted to those labeled by field names defined in the object's type. In particular, there should never be edges emanating from nodes for simple objects.

The language TERM is too weak to be a canonical language. Since TERM describes only the class of structures that are rooted directed trees, it cannot be used to specify structure sharing, which is important to the TEDM object model. The language TERM/VAR, on the other hand, supports a limited version of object sharing. In

TERM/VAR, multiple occurrences of the same variable must be substituted consistently, i.e., by the same object, which amounts to sharing the substituting object by multiple containing objects. Object sharing in TERM/VAR is limited in the following sense: Although they can be substituted by complex terms, the variable themselves can only occur as leaves. In other words, TERM/VAR (before substitution) is structurally equivalent to the class of *tree DAG's*, rooted directed trees whose leaf nodes may have multiple paths from the root. Furthermore, if the language TERM/VAR is used in a system where only 0-arity terms (that is, constants) are permitted in substitution for variables, TERM/VAR denotes precisely the class of tree DAG's. There is still another deficiency in this language: terms have fixed arity — they do not allow extra fields. Therefore, TERM/VAR is still too weak to be TEDM's canonical language.

The language SET is strictly more powerful than the graph formalism. It is capable of specifying any graph, at a relatively low level. It may be good as an internal representation language. For example, the *2-3 storage model* in [Zhu86] is basically a set notation. Readability and intuitiveness are important qualities of languages. Hence, the language SET needs to be extended with richer syntactical constructs, to become a useful specification language.

Extending the Language TERM/VAR

The language TERM/VAR can be extended to support a more general notion of structure sharing. The basic idea for more generalized sharing is still the consistent substitution for object variables. The semantic basis that makes the extension possible, however, is the assumption that each complex structure is associated with a common identity. Object variables are mapped to object identities. If an object variable has multiple occurrences, then they are mapped to a unique object identity, referencing the same complex structure. In fact, not only genuine rooted DAG's can be specified, cycles can also be introduced into object structure diagrams under the scheme.

We call the result of this section's extension TERM/VAR*. Each sentence, or term, in TERM/VAR* corresponds to a primary object, and possibly a set of subordinate objects, in the database. The language TERM/VAR* by itself alone, however, is still insufficient as TEDM's canonical language. We will deal with the remaining deficiencies in the next section.

The language TERM/VAR* is obtained from TERM/VAR by the following relaxations on syntax: The order of argument terms can be rearranged without affecting the meaning (which is possible since the terms are labeled), terms may contain a variable number of argument subterms, and variables may be associated with the term or subterms. The resulting language, TERM/VAR*, is essentially the language of the O_terms of [Maier86], and similar to feature structures in unification grammars [Porter88].

The ordering of arguments in TERM/VAR* becomes insignificant, since field labels are used to associate the primary term with the subterms. This simple addition also improves the readability to some extent. Consider the description of a schematic for an RS-Flip-Flop, shown in Figure 3.5. Points in a Cartesian coordinate space are employeed to convey the connection information among the pins of logical components. When two pins refer to the same point, they are connected with each other.

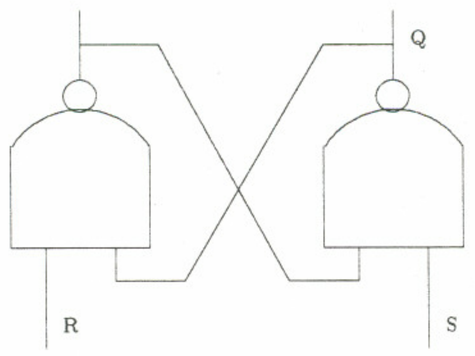

Figure 3.5 An RS Flip-Flop

Using TERM/VAR, we can describe the schematic object as follows:

```
rsFF(nand2(point(2, 0), point(1, 4)),
     nand2(point(1, 4), point(2, 0)))
```

Some information is implicit in this description: We assume that the first argument in "point" represents the x-coordinate of a point, while the second represents the y-coordinate, and so on. Coordinate values are used to determine whether two points are actually the same.

Using field labels, we can rewrite the description for the RS-Flip-Flop as the following:

```
rsFF(leftNand→nand2(rightPin→point(x→2, y→0), outPin→point(x→1, y→4)),
     rightNand→nand2(leftPin→point(x→1, y→4), outPin→point(x→2, y→0)))
```

where field labels are separated from subterms by "→". The situation is improved a little, but we still have to rely on value equality to refer to the same object. To solve this problem, we make use of *object variables*, variables that are bound to object identities. As a result, we can associate object variables with complex objects, and express structure sharing readily. For example, the fact that two point objects in the RS-Flip-Flop are shared can be specified as follows:

```
rsFF(leftNand→nand2(rightPin→P1 = point(x→2, y→0),
                     topPin→P2 = point(x→1, y→4)),
     rightNand→nand2(leftPin→P2, topPin→P1))
```

This description captures the fact that the right pin of the left nand-gate and the top pin of the right nand-gate are connected to the same grid point. Similarly, the left pin of the right nand-gate and the top pin of the left nand-gate are connected to the same grid point. In the description, "P1" and "P2" are object variables. Notice we no longer need to describe the coordinates for the rightNand to capture the shared connections.

A term is most often characterized by its name (functor) and its arity (the number of arguments). A functor may be associated with several arities and different <functor, arity> pairs refer to distinct functions. In our use of TERM/VAR (including extensions we have discussed), functors are construed as classifiers rather than functions. For classification purposes, it is useful to define an order on data records, based on the amount of information they contain (or the generality), as exemplified by the following:

```
name(first → 'Joe') ≥ name(first → 'Joe', last → 'Joy'), and
name(first → 'Joe', last → 'Joy') ≥
        name(first → 'Joe', last → 'Joy', middle → 'Jay')
```

The arity of terms in TERM/VAR* can be a variable quantity. In other words, the functor alone determines the category of the data records, although some records may contain more information than the others. The fact that no particular order is required on term arguments also contributes to the flexibility of adding or dropping sub-terms. Continuing the RS-Flip-Flop example, we add two input pins and an output pin to its description:

```
rsFF(leftNand→nand2(rightPin→point(x→2, y→0), topPin→point(x→1, y→4)),
     rightNand→nand2(leftPin→point(x→1, y→4), topPin→point(x→2, y→0)),
     leftPin→inPin(pinName→'R'),
     rightPin→inPin(pinName→'S'),
     topPin→outPin(pinName→ 'Q'))
```

Notice that both an RS-Flip-Flop and a two-input nand gate have a left pin ("leftPin"). In this case, the two fields contain two distinct objects, although we may expect that the types of the two objects are related, e.g., external pins should have symbolic names.

3.2. Placeholders and Object Tags

Traditionally, variables are purely placeholders in a logic system. They are used to help form abstractions with quantification. Usually, a term with free variables is hard to interpret and most logic systems assign no meaning to such terms. The extended language with arbitrary co-reference variables still only yields ground terms. Therefore, in addition to co-reference variables, we add another class of variables whose free occurrences permit meaningful interpretations. This new class of variables are called *object tags* (or simply *tags*), and they are interpreted by abstract objects.

Objects tags are closely related to placeholders: Both are related to the notion of substitution of one entity by another. The main difference is: placeholders must be bound to constants (including object identifiers), when occurring in a term. But object tags in a term are not handled as variable substitution when the term is interpreted. Instead, an interpretation function shall map object tags to abstract objects. In other words, when a syntactical construct, e.g., a term, is interpreted by a semantic object (or translated into an internal representation), placeholders disappear in the representation, whereas object tags shall remain.

The significance of object tags is that they make it possible to define inverse maps under which object tags become variables, or better yet, to define directly bindings of one type of semantic object with another. The inverse maps allow us not only to recover data stored (converting the internal representations to the external forms), but also to recover commands, etc., up to renaming of placeholders.

The use of placeholders is consistent with their traditional roles. They are syntactic symbols that are not interpreted directly in a model. As in logic, nonclosed formulas rely on variable assignments to determine their truth values. Truth values thus obtained are relative to variable assignments. The treatment of placeholders in our system is only slightly different. Object terms define the assignment to placeholders occurring in them. Similarly, the notions of free occurrences and bound occurrences of placeholders are used to define *closed* terms.

Example 11: In the following term, F and L occur free; and E and N occur bound.

$$\text{Employee:E(name} \rightarrow \text{PersonName:N(first} \rightarrow \text{F, last} \rightarrow \text{L))}$$

They are bound to the objects (object identifiers) denoted by the terms Employee:(name → PersonName:(first → F, last → L)) □

We augment TERM/VAR* with the distinction between placeholders and object tags. We also refer to the sentences in the resulting language as *object terms.*

Notation: The following conventions are used in discussion. $\mathbf{T}$, $\mathbf{T}_1$, $\mathbf{T}_2$, ..., $\mathbf{T}_n$ are type names. $\mathbf{f}$, $\mathbf{f}_1$, $\mathbf{f}_2$, ..., $\mathbf{f}_n$ are field names. $\mathbf{V}$, $\mathbf{V}_1$, $\mathbf{V}_2$, ..., $\mathbf{V}_n$ are placeholders. $\mathbf{P}$, $\mathbf{P}_1$, $\mathbf{P}_2$, ..., $\mathbf{P}_n$ are object tags. $\mathbf{c}$, $\mathbf{c}_1$, $\mathbf{c}_2$, ..., $\mathbf{c}_n$ are constants. We also use $\mathbf{w}$, $\mathbf{w}_1$, $\mathbf{w}_2$, ..., $\mathbf{w}_n$ to denote object terms.

Definition 1: Object terms are defined recursively as follows:

1 Each of the following is an object term
  1.1 **c** (a constant).
  1.2 **V** (a placeholder).
  1.3 **T**:**P**? (an object tag).

2 Given that $\mathbf{w}_1, ..., \mathbf{w}_n$ are object terms, then so are these:
  2.1 $\mathbf{T}:\mathbf{V}(\mathbf{f}_1 \rightarrow \mathbf{w}_1, ..., \mathbf{f}_n \rightarrow \mathbf{w}_n)$ (a complex object with a placeholder).
  2.2 $\mathbf{T}:\mathbf{P}?(\mathbf{f}_1 \rightarrow \mathbf{w}_1, ..., \mathbf{f}_n \rightarrow \mathbf{w}_n)$ (a complex object with an object tag).

3 **T**, **w** is an object term (a multiply typed object).
4 Nothing else is an object term.

Each placeholder **V** must occur exactly once in a 2.1 term, and zero or more times in a 1.2 term. Placeholder occurrences are unique. □

In this definition, the first set of clauses say that data constants, placeholders and typed object tags are object terms. The second set of clauses say that more complex object terms are constructed by grouping simpler ones using field labels. In particular, Clause 2.1 describes an object of type **T** with n fields. In addition, the identifier of the object is bound to a placeholder **V**. Clause 2.2 is similar to Clause 2.1 except that an object tag is used, indicating an abstract object. The third clause says there can be more than one type symbol in an object term, indicating *multityping* (one object being a member of more than one type.) We also use the notation "$\mathbf{T}:(\mathbf{f}_1 \rightarrow \mathbf{w}_1, ..., \mathbf{f}_n \rightarrow \mathbf{w}_n)$" as a shorthand when the placeholder is unique.

Example 12: *12345* and *'Joe'* are constant symbols denoting simple objects. (Clause 1.1). *Person:P?* is an object term denoting an abstract object that matches concrete objects of type *Person* (Clause 1.2). □

Example 13: The term

PersonName:(first → 'Joe', last → 'Dow')

denotes a complex object representing an individual's name (Clause 2.1). The term

Person:V(name → PersonName:(first → 'Joe', last → 'Dow')
addr → Address:(strNo → 1234, strNm → 'First Ave.'))

denotes a complex object corresponding to a person, which consists of a PersonName subobject and an Address subobject (Clause 2.2). The placeholder V in this case is bound to the identifier of the object. In general, more than one occurrence of V may occur in a term indicating sharing of objects, which is a more useful scenario. The term

Person:P?(name → PersonName:N?(last → 'Dow'))

denotes an abstract object, which contains an abstract PersonName object (Clause 2.3). The abstract PersonName object matches concrete PersonName objects whose last name field has a value of a simple object, 'Dow'. The abstract Person object matches concrete Person objects whose last name is 'Dow.' □

Example 14: The term

Stockholder, Manager:(name → PersonName:(first → 'Joe',
last → 'Dow'),
addr → Address:(strNo → 1234,
strNm → 'First Ave.'))

denotes a complex object with multiple types. □

We require all objects be *properly typed*. (The meaning of proper typing will be made explicit in the next chapter.) Types are always explicitly given when writing object terms, except for simple objects, whose types are self-declaring.

3.3. Semantics

Object terms or groups of object terms are specifications (and constructors as well) for database objects. Technically, they are formulas interpreted by (or mapped to) objects — the objects provide meanings for the terms. Designing a mapping (an interpretation) from formulas to objects is usually referred to as defining the semantics of a language. To accurately reflect the meaning of the syntactical objects, certain constraints are always to be observed by interpretations. For example, an interpretation for the first-order logic must map constants to individuals of a data domain, a function symbol to a function (of proper arity) over the data domain, and a predicate symbol to a relation over the data domain, etc. In our case, each simple term is interpreted by one and only one atomic data value, while a complex term can be mapped to any one of an infinite number of complex objects (they are isomorphic nevertheless). Other interpretation rules are detailed below.

We use structured spaces for interpreting the object definition language. Elements (or objects) of such a space are rooted directed graphs, which we called object structure graphs earlier. We utilize a textual format to describe elements of the interpretation spaces.

Notation: **G** denotes an interpretation space. Elements of **G** are graphs, denoted as $\mathbf{g}$, $\mathbf{g}_1$, $\mathbf{g}_2$, ..., $\mathbf{g}_n$. The following three disjoint sets are assumed: $\mathbf{ID}_D$ is a set of nodes labeled by primitive data values, whose elements are $\mathbf{v}$, $\mathbf{v}_1$, $\mathbf{v}_2$, ... $\mathbf{v}_n$. $\mathbf{ID}_C$ is a set of nodes labeled by concrete object identities. $\mathbf{ID}_A$ is a set of nodes labeled by abstract object identities. Unions of the three sets are denoted with proper subscripting. For example, $\mathbf{ID}_{AC}$ denotes the union of $\mathbf{ID}_C$ and $\mathbf{ID}_A$; $\mathbf{ID}_{ACD}$ denotes the union of $\mathbf{ID}_D$, $\mathbf{ID}_C$ and $\mathbf{ID}_A$, with elements are $\mathbf{i}$, $\mathbf{i}_1$, $\mathbf{i}_2$, ..., $\mathbf{i}_n$. $\mathbf{E}$ denotes a set of edges labeled by field names, whose elements are $\mathbf{e}$, $\mathbf{e}_1$, $\mathbf{e}_2$, ..., $\mathbf{e}_n$ ($\mathbf{e}_i$ labeled by $\mathbf{f}_i$). In addition, all nodes in an object structure graph are tagged by type names.

Definition 2: A structure space, $\mathbf{G}$, is constructed as follows:

1). if $\mathbf{v} \in \mathrm{AD}$, then $(\ \mathbf{v}, \{\ (\text{type:}v_T)\ \}\ ) \in \mathbf{G}$, where v_T denotes the type of v
2). if $\mathbf{i} \in \mathbf{ID}_{AC}$ and $\mathbf{g}_i \in \mathbf{G}$, then $\mathbf{g} \equiv (\mathbf{i}, \{(\mathbf{e}_1 : \mathbf{g}_1), \ldots, (\mathbf{e}_n : \mathbf{g}_n), (\text{type:}\mathbf{i}_T)\}) \in \mathbf{G}$;
where $\mathbf{i}_T$ denotes the type of $\mathbf{i}$
we say $\mathbf{g}$ has as its components each $\mathbf{g}_i$ and all of $\mathbf{g}_i$'s components;
and any of the $\mathbf{g}$'s components can be $\mathbf{g}$ itself

□

The first statement of the definition covers the basis: graphs of simple data values or object identifiers are elements of a structure space. The notation ($\mathbf{e}$: $\mathbf{g}$) means there is an edge $\mathbf{e}$ going into the root of the subgraph $\mathbf{g}$. The second statement says that a new element can be constructed from simple ones: $\mathbf{i}$ is the root of $\mathbf{g}$; and the root has (n + 1) edges, $\mathbf{e}_1$, ..., $\mathbf{e}_n$ and $\mathbf{e}$, directed to the roots of the subgraphs $\mathbf{g}_1$, ..., $\mathbf{g}_n$ and $\mathbf{i}_T$, respectively. Note that the field names need not be distinct — there may be two or more edges with the same label going out from the root.

An interpretation maps object terms to object structure graphs in $\mathbf{G}$, and works as follows (refer to Definition 1).

1). Each of the two clauses is mapped to a node in the resulting object structure graph. A constant **c** is mapped to a concrete node labeled by the value of the constant, and tagged by the type of the value. An object tag placeholder **T**:**P**? is mapped to an abstract node from $\mathbf{ID}_{\mathbf{A}}$, labeled by a unique object identity and tagged by the type of the abstract object id.

2). For each of the three subcases, a node is selected or created; and from the node, (n + 1) edges (labeled with $\mathbf{f}_1$, ..., $\mathbf{f}_n$) are added, leading to the roots obtained by mapping $\mathbf{w}_1$, ..., $\mathbf{w}_n$, respectively. For case 2.1, a concrete node (of type **T**) with a new object identifier is created. Handling of the case 2.2 is similar, except that in this case, a symbol table containing binding information for placeholders needs to be consulted or updated to deal with co-reference. (Recall this form defines the binding of the placeholders to the object identity.) For case 2.3, a new abstract node is introduced. An additional node (of type String) is created to hold the external names for the abstract node; an edge labeled by *tag* is used to connect the additional node to the abstract node.

3). For this case, a new type **T** is attached to the root node of the object graph for **w**.

The following definition restates the interpretation rules in a formal manner.

Definition 3: The semantic function, Σ, maps object terms into objects. In doing so, it uses an environment, Env, that maps types to type defining objects (see the next chapter). The environment is also used to maintain information about placeholder and object tag bindings. We assume that the Env maps all placeholders and object tags to unique abstract and concrete ids. The interpretation of an object term is unique up to selection of these ids.

1.1) $\Sigma [\![\ \mathbf{c} \ ]\!] \, \mathrm{Env} = (\mathbf{c}, \{(\mathrm{type}{:}\mathrm{Env}\ \mathbf{c}_T)\}$

1.2) $\Sigma [\![\ \mathbf{V}? \ ]\!] \, \mathrm{Env} = (\mathrm{Env}\ \mathbf{P}, \mathrm{S}),$
where S is the set of edges defined by the unique occurrence of V in a 2.1 term

1.3) $\Sigma [\![\ \mathbf{T}{:}\mathbf{P}? \ ]\!] \, \mathrm{Env} = (\mathrm{Env}\ \mathbf{P}, \{(\mathrm{type}{:}\mathrm{Env}\ \mathbf{T}), (\mathrm{tag}{:}\mathbf{P})\})$

2.1) $\Sigma [\![\ \mathbf{T}{:}\mathbf{V}(\mathbf{f}_1 \rightarrow \mathbf{w}_1, \ldots, \mathbf{f}_n \rightarrow \mathbf{w}_n) \ ]\!] \, \mathrm{Env} =$
$(\mathrm{Env}\ \mathbf{V}, \{(\mathrm{type}{:}\mathrm{Env}\ \mathbf{T}), (\mathbf{f}_1{:}\Sigma [\![\ \mathbf{w}_1 \ ]\!] \, \mathrm{Env}), \ldots,$
$(\mathbf{f}_n{:}\ \Sigma [\![\ \mathbf{w}_n \ ]\!] \, \mathrm{Env}))\})$

2.2) $\Sigma [\![\ \mathbf{T}{:}\mathbf{P}?(\mathbf{f}_1 \rightarrow \mathbf{w}_1, \ldots, \mathbf{f}_n \rightarrow \mathbf{w}_n) \ ]\!] \, \mathrm{Env} =$
$(\mathrm{Env}\ \mathbf{V}, \{(\mathrm{type}{:}\mathrm{Env}\ \mathbf{T}), (\mathrm{tag}{:}\mathbf{i}),$
$(\mathbf{f}_1{:}\Sigma [\![\ \mathbf{w}_1 \ ]\!] \, \mathrm{Env}), \ldots, (\mathbf{f}_n{:}\Sigma [\![\ \mathbf{w}_n \ ]\!] \, \mathrm{Env}))\})$

3) $\Sigma [\![\ \mathbf{T}, \mathbf{w} \ ]\!] \, \mathrm{Env} = (\mathrm{type}{:}\mathbf{T}) \cup \Sigma [\![\ \mathbf{w} \ ]\!] \, \mathrm{Env}$

□

Clause 1.1 states that a constant is simply mapped to its value, which is itself. In Clause 1.2, we used the notation "Env <name>" to denote looking up the <name> from the environment. Type information is recorded via a distinguished edge from every node. An object tag name is also recorded as a field.

Clause 2.1 creates a concrete object with the identity assigned to **V**, and fields described by the object term. Clause 2.2, creates an abstract object in a similar fashion. The interpretation for Clause 3 attaches an additional type to an object.

3.4. Other Considerations and Chapter Summary

Collection objects are not directly available in TEDM. Instead, they are supported indirectly via *multiple occurrence fields*, fields that may have more than one value (including none). In object graphs, an object having a field with more than one value means there are multiple edges with the same label emanating from the node for the object.

A shorthand syntax is provided for indicating multiple occurrence fields; this is done by conjoining subterms with an ampersand (&). The following example illustrates the use of multiple occurrence fields.

Example 15: In VLSI design, a polysilicon rectangle across a diffusion region forms a transistor.

```
FETLayout:(fet → SimpleLayout:
                    (layoutunit → LayoutUnit:
                                    (rect → Rectangle:
                                               (width → 2,
                                                height → 6),
                                     color → 'GREEN'),
                     position → Point:(x → 2, y → 0)))
                 & SimpleLayout:
                    (layoutunit → LayoutUnit:
                                    (rect → Rectangle:
                                               (width → 6,
                                                height → 2),
                                     color → 'RED'),
                     position → Point:(x → 0, y → 2)))
```

□

Closely related to multiple occurrence fields are indexed fields, which are indicated by index values delimited using square brackets. Like field labels, we assume index values come from a fixed underlying domain (the positive integers.) Indexed fields are useful for applications where the order among a group of similar objects needs to be be maintained.

Example 16: In VLSI layout design, it is convenient to place a number of layout units as a group.

```
LayoutArray:(region →[1] SimpleLayout:
                          (layoutunit → LayoutUnit:
                                          (rect → Rectangle:
                                                     (width → 2,
                                        height → 6),
                              color → 'GREEN'),
                  position → Point:(x → 2, y → 0)))
          &[2] SimpleLayout:
                (layoutunit → LayoutUnit:
                              (rect → Rectangle:
                                          (width → 2,
                                           height → 8),
                              color → 'GREEN'),

                 position → Point:(x → 6, y → 0)))
```

□

Example 17: A connection on a printed circuit board can be represented as a connector object shared by pin objects to be connected together. In the following pullup transistor, the gate pin and the drain pin are tied to the same connector object, as indicated by the placeholder C.

```
Pullup:(g → Terminal:
               (termName → 'gate',
                connectTo → Connector:C
                               (position → Point:(x → 0, y → 0),
                                signal → Signal:(sigName → 'T3D',
                                                 sigVal  → 'unknown')),
        d → Terminal:
               (termName → 'drain', connectTo → C),
        s → Terminal:
               (termName → 'source',
                connectTo → Connector:
                               (position → Point:(x → 10, y → 10),
                                signal → Signal:(sigName → 'T3OUT',
                                                 sigVal  → 'unknown')))
```

□

To sum up, we have described a language for defining objects in TEDM. More importantly, we have studied in detail several ways for describing complex structures, pointed out their strength and weakness. The study, combined with our goal to achieve total objectification, leads to the decision of splitting traditional variables into two kinds, placeholders and object tags. Finally, formal syntax and semantics of the object definition language are proposed. It would be too hard to extend semantics to allow duplicate object tags, making them function as both tags and co-reference placeholders.

CHAPTER 4

DEFINING TYPES

Types define and enforce the structure of objects: What fields an object may have and what values these fields can take are stipulated to a large degree by the type of the object. We allow an object to be *multiply typed.* In this case, the object is constrained to satisfy the structural requirements from each of its types. Types also play importants roles in conceptual modeling: They naturally correspond to prototypical concepts and can be used for classification purposes (dividing objects into conceptual classes). The type system of TEDM is designed to allow these dual roles, constraining object structure and representing prototypical concepts, to be easily blended.

Several important concepts for typing are *types*, *subtypes*, *conformity* of objects to types, *membership* of objects participating in types (*typesets*), and the classes of *object sets* (*collections* or *domains*) definable from the typesets using the usual set operations such as intersection, subsetting, and so on.

The static aspect of a type defines a set of fields, and their respective domains, as a necessary condition for its members: the *conformity condition* of a type. Such a conformity condition states that, in order for an object to be a member of the type, it must contain at least as many fields as there are defined for the type, and each field has a legitimate value as specified in the type definition. An object conforming to a type can become a member of the type (meaning an element of the typeset). We require all objects belong to at least one type. We also require type definitions be given before its instances can be created.

This chapter discusses what types are and how they are defined in TEDM. For these purposes, we briefly overview the general notion of types, and describe the syntax and the semantics of a language for defining types. In Section 2, the notion of typing in TEDM is described, along with four relations that characterize how the types are related to each other, and how objects and types are related. Section 3 presents the syntax of a language for defining types. Section 4 discusses the semantics of the language. The chapter is summarized in Section 5.

4.1. What Is a Type?

The concept of types has been used extensively in programming languages. Types in programming languages are used in the following two ways. They provide a high-level abstraction on data, allowing the construction of complex data structures that are not directly representable by the hardware (for example, a record type); they give rise to an important feature in modern programming tools (for example, compilers), *type checking*. Most compilers can detect obvious type errors, such as adding a Boolean value and a Real value, or assigning an Employee record to an Account variable. To detect more subtle errors, such as division by zero, type systems more comprehensive than the ones in conventional programming languages are needed. (For example, one which can define a domain "PositiveNumber" as "Number" minus the zero.)

A subtle issue regarding type checking is *type-equivalence*: When are two types considered identical and when are they are? There are approaches: equivalence by declaration, and equivalence by structure compatibility. In the first approach, two types are considered distinct even if they have identical definitions. Hence, assigning a value of one type to a variable of the second type always results in a type error. In the second approach, two types are equivalent if they are structurally compatible. The structural compatibility can be determined by the compiler at compile time.

The type systems of early languages such as Pascal tend to be rather rigid. The rigid requirements often become a burden rather than a convenience. Consider as an example the identity function ($\lambda x.x$). Apparently, this function has a well defined type, $\forall T.(T \rightarrow T)$, where T is a type variable that can be replaced by any type. However, this function is not implementable in PASCAL. Object-oriented (OO) programming languages have statically typed versions (such as C++ [Stroustrup86] and Trellis/Owl [Schaffer86]) and dynamically typed versions (such as Smalltalk-80 [Goldberg83] and Objective-C[Cox86]). OO languages support limited polymorphism with their type hierarchies, where objects of a subtype can be used wherever objects of its supertype(s) are expected. In addition, types in OO systems naturally correspond to the conceptualization of the real world entities. For example, an object of type Person in an OO system is likely intended to be an abstraction of a real person. Although static typing is advantageous in many respect, the flexibility of dynamically typed OO languages makes certain data structure and operations extremely easy, heterogeneous collections or sets being such an example. On the other hand, supporting late binding of messages to methods may require substantial compile-time analysis of run-time behavior. (Consider an object of a subtype being assigned to a variable of a supertype, which is then sent a message that is redefined by the subtype).

The conceptually oriented semantics of types are especially appealing to database systems, as conceptual modeling has always been an important part of developing database applications. Before OO systems, there was a clear distinction between types in programming languages and those in conceptual schemas. The distinction has since blurred and is eroding continually, as is evident from the large amount of work in the area of database programming languages [Hull89]. It is interesting to note that the semantics of the type systems have been the driving force for the convergence, and at the same time, still a major source where the type systems of programming languages and the type systems of databases differ. For example, one of the differences between type semantics in OO programming languages and OO databases lies in the treatment of subtype instances: are they also instances of the supertype? To this question, most OO databases' answer is probably "Yes", whereas most OO programming languages a "No" or "Don't know". (Free substitution by subtype instances does not necessarily give rise to the containment semantics, the instances of a subtype being those of its supertype or supertypes.) This difference in semantics becomes important for determining function types For example (see [Breazu-Tannen89]), what is the type for λx:Person.x?

4.2. Typing in TEDM

TEDM uses types to organize objects according to their structure. Objects of the same type have a "similar" structure, and we say that the objects *conform* to the type. In general, an object conforming to a type also conforms to its supertype(s). Nevertheless, it is always possible to define a unique type (up to structure isomorphism) that bounds an object from below in a type hierarchy. This unique (boundary) type is said to *minimally* define the object structure, in that any field dismissal from the object structure would result in the object no longer conforming to the type. The most general type, "All", is furnished by the system, which defines an empty structure, and to which every object conforms. Consequently, the types of an object, though not unique, are always bounded both from above and from below.

There is a *typeset* associated with each type. The typeset of a type contains objects conforming to the type. However, the membership of an object in a typeset is not automatic; request for inclusion into a typeset has to be made explicitly. In other words, although the notion of conformity of an object to a type is determined solely by structural compatibility, the notion of membership of an object in a typeset is dependent on explicit declarations. The typeset defines the *active domain* for the type. An object can be a member of multiple active domains.

Similar to the notion of conformity defined on objects and types, we define a relation to characterize structural compatibility among types. A type $\mathbf{T}_1$ is said to *specialize* a second type $\mathbf{T}_2$ (or $\mathbf{T}_2$ *generalizes* $\mathbf{T}_1$), if the conformity condition of $\mathbf{T}_1$ is stronger than that of $\mathbf{T}_2$. (Here $\mathbf{T}_1$ has a stronger conformity condition means $\mathbf{T}_1$ defines a superset of fields of what $\mathbf{T}_2$ defines.) If $\mathbf{T}_1$ specializes $\mathbf{T}_2$, $\mathbf{T}_1$ can be declared to be a *subtype* of $\mathbf{T}_2$. The *subtype* relation establishes an explicit type hierarchy, with each element in the relation corresponding to a parent-child link in the diagram for the types.

While it may seem that the typeset for a type is an extension of the type, it is not maintained as such. Typeset membership is associated with the element objects, not with the type (-defining object). An extension of a type is an object set constructible from the the active domain of the type. In other words, there can be many collections of the objects of a given type. This is useful for dealing with types that need multiple incarnations, for example, books can be technical or nontechnical, employees can be still in service or retired, etc. Although a typeset is similar to a table (or a relation) of the relational model, the notion of extension in TEDM is different from what it is in the relational model. In the relational model, each scheme (an intension) defines one and only one table (an extension). The separation of the intension of a type from the extension of the type has a number of implications. For example, more efficient query processing is possible, as each collection in the extension provides a reduced search space. Also, the separation makes it easy for the objects to migrate among the sets within the family definable from a typeset. In this case, no special processing on the object type is necessary. Objects are likely to move from one set to another more often than to change from one type to another. For example, a person may get hired, becoming a current employee; the same person may retire in the future, becoming a retired employee, etc.

Continuing with the general discussion, we define two relations between objects and types. (In the sequel, we use **TI** to denote type intension, and **TE** type extension.) The relation $\in^d \subseteq \mathbf{G} \times \mathbf{TE}$ (read as "defined member of") describes object's membership in an active domain for a type. The meaning of $\mathbf{o} \in^d \mathbf{T}$, where $\mathbf{o}$ is an object and $\mathbf{T}$ a type, is that $\mathbf{o}$ is an element of the active domain of $\mathbf{T}$ ($\mathbf{o} \in \mathit{adom}(\mathbf{T})$). The relation $\in^i \subseteq \mathbf{G} \times \mathbf{TI}$ (read as "inferred member of" or "conforms to"), on the other hand, is used to characterize object's structure with respect to types. The meaning of the expression $\mathbf{o} \in^i \mathbf{T}$ is that the object $\mathbf{o}$ conforms to the type $\mathbf{T}$ ($\mathit{struct}(\mathbf{o}) \supseteq \mathit{struct}(\mathbf{T})$).

Informally, the former relation states a membership fact, that an object *is* of a type; while the latter relation states a membership possibility, that an object *can be* of a type. The two concepts are related as follows:

$$\mathbf{o} \in^{d} \mathbf{T} \Longrightarrow \mathbf{o} \in^{i} \mathbf{T}.$$

Also, when an object acquires a type, it acquires all declared supertypes.

The characterization of subtyping aspect of TEDM's type system is similarly handled. The relation $\leq^{d} \subseteq \mathbf{TI} \times \mathbf{TI}$ (read as "defined subtype of "), like $\in^{d}$, captures the declarational aspect: the expression $\mathbf{T1} \leq^{d} \mathbf{T2}$, where both **T1** and **T2** are types, states that **T1** is an *immediate* subtype of **T2**. The relation $\leq^{i} \subseteq \mathbf{TI} \times \mathbf{TI}$ (read as "inferred subtypes of"), on the other hand, emphasizes the structural aspect: $\mathbf{T1} \leq^{i} \mathbf{T2}$ means *struct*(**T1**) $\supseteq$ *struct*(**T2**)).

In parallel to the relationship between $\in^{d}$ and $\in^{i}$, the following holds between $\leq^{d}$ and $\leq^{i}$:

$$\mathbf{T1} \leq^{d} \mathbf{T2} \Longrightarrow \mathbf{T1} \leq^{i} \mathbf{T2}$$

The relation $\leq^{d}$ must contain no cycles: a type cannot transitively be a subtype of itself. The relation $\leq^{i}$ is required to be a partial order: it is reflexive, transitive and antisymmetric. Furthermore, $\leq^{d*} \subseteq \leq^{i}$: the reflexive and transitive closure of $\leq^{d*}$ is a subset of $\leq^{i}$.

The relation $\leq^{d*}$ establishes the type hierarchy of a database. We mentioned that the top element of the type hierarchy is a predefined type "All". The type definition for "All" is simply

$$\text{All} = ().$$

As "All" defines a null conformity condition, every object in the database is $\in^{i}$-related to "All". Furthermore, every other type in the database is $\leq^i$-related to "All".

When a type is defined in TEDM, several things happen. First, an object of type "TypeDef", a type-defining object, is created as the internal representation of the type. The precise definition of "TypeDef" will be given in a later chapter. Second, the new type is related to the existing types through the $\in^d$ relation, which determines the relative position of the new type in the hierarchy. The rule for finding a position for the new type is: if the type definition explicitly specifies a supertype (or multiple supertypes), then the new type is considered $\in^d$-related to the supertype(s); otherwise, a default supertype of "All" is assumed. Third, a structure template is created with which the system can determine, given an object, the conformity of the object with respect to the type. Finally, the typeset for the new type also comes into existence.

One uses *value specifications* (**VS**) and *type specifications* (**TS**) to define types in TEDM. A type specification defines a type. Value specifications provide a flexible way to define the domains for fields of a type. In a value specification, expressions can be used to define new domains. The implications are twofold. First, the domain of a field does not have to be a type, it can be complex expression composed of types or others similar set-defining constructs. Second, the language for value specifications is strictly more powerful than the language for type specifications. Semantically, each expression in **VS** defines a predicate, characterizing *admissible* values for fields. Each expression in **TS**, on the other hand, entails more complicated semantic relationships with other types and objects in the database, among them are $\in^d$, $\in^i$, $\leq^d$ and $\leq^i$.

4.3. Syntax

The syntax for defining types is presented in three steps. First, we define the syntax for value specifications, which is an expression-like language with type names as primary operands. Second, the syntax for type specification is described, the result is a simple language with structure enumeration and nesting. Third, a simple assignment-like statement is given, which is used to associate type names with type definitions.

Specifying Values

An expression in **VS** defines, for a field in a complex object, a range of values, the set of admissible values (*admissible sets*). Three operators can be used in composing such expressions: the *conjunction* ($\wedge$), the *disjunction* ($\vee$) and the *complement* ($\neg$).

Definition 4: The set of well-formed value specifications, **VS**, is the smallest set satisfying the following:

1) $\mathbf{T} \in \mathbf{VS}$, where $\mathbf{T}$ is a type name,
2) $\mathbf{V}_1 \neg \mathbf{V}_2 \in \mathbf{VS}$, if $\mathbf{V}_1, \mathbf{V}_2 \in \mathbf{VS}$
3) $\mathbf{V}_1 \wedge \mathbf{V}_2 \in \mathbf{VS}$, if $\mathbf{V}_1, \mathbf{V}_2 \in \mathbf{VS}$
4) $\mathbf{V}_1 \vee \mathbf{V}_2 \in \mathbf{VS}$, if $\mathbf{V}_1, \mathbf{V}_2 \in \mathbf{VS}$

□

A type name alone is an expression (Clause 1). The admissible set is the typeset of the type. For example, "Number", "String", "Point" and "Rectangle" are all value specifications. They define the set of numbers, strings, point objects and rectangle objects, respectively.

The complement an expression is an expression (Clause 2). The admissible set is the set difference of the two types' admissible sets. For example, "Employee $\neg$

Manager" defines the set of Employee objects excluding Manager objects. We use "$\neg$ **F**" as a shorthand for "All $\neg$ **F**". Two expressions combined with a conjunction operator form a new expression (Clause 3), with the resulting admissible set being the intersection of those of its two operands. For example, "Employee $\wedge$ Student" defines the set of database objects that possess both type Employee and type Student. Similarly, two expressions combined with a disjunction operator also form a new expression (Clause 4), with the resulting admissible set being the union of those of its two operands. For example, "Number $\vee$ String" defines the set of database objects that are either a number or a string.

Other forms of value specifications are also possible. For example, an abstract object (actually, its syntactical counterpart, the tag variable), can be a value specification. It denotes a set of database objects that it matches. Finite enumerations or structure restrictions can also be value specifications, denoting the finite set of values or a subset satisfying the restrictions, respectively. An informal discussion on these possible extensions will be given in a later chapter.

Specifying Types

The language for type specification contains suitable constructs for enumerating finite lists, the lists of fields in complex object types. Each field consists of a field name and a value specification. In addition to lists of fields, type expressions may also be formed by successive *supertype prefixing*. Each such prefix defines for the new type a supertype (by extending the $\leq^d$ relation).

Definition 5: Given that **T** is a type name, and $\mathbf{f}_i$'s the field names, for i = 1, ..., n, the set of well-formed type specification expressions, **TS**, is the smallest set satisfying the following.

1) $(\mathbf{f}_1 \rightarrow \mathbf{V}_1, \ldots, \mathbf{f}_n \rightarrow \mathbf{V}_n) \in \mathbf{TS}$;
2) $\mathbf{T}: \mathbf{t} \in \mathbf{TS}$, if $\mathbf{t} \in \mathbf{TS}$.

□

A list of fields delimited with a pair of parentheses is a type specification (Clause 1). No duplicate field name may occur, and the type defined in this manner contains precisely the fields in the list. Each field of the form "*fieldName → valueSpecification*" defines a conformity condition *primitive*, which says that for an object to conform with the type, it must contain a field with the name "fieldName" that takes value from the admissible set defined by the "valueSpecification". Clause 2 suggests that the prefix of the form "typeName:" can be prepended to type expressions, which is a convenient way to define supertypes. The resulting type defined this way becomes a subtype of the types listed. The field list of the new type is the field list appearing in the definition, augmented with the field list from each of its supertypes (*multiple inheritance*).

Name conflict in multiple inheritance is handled as follows. When two supertypes contain a common field name, "fieldName", with value specifications "valueSpec1" and "valueSpec2", respectively, the subtype inherits "fieldName" with a value specification "valueSpec1 $\vee$ valueSpec2". That is, the admissible set of the subtype on this field is the union of those of its two supertypes. Generalization into the case with more than two supertypes is immediate.

Example 18: Below are simple type specifications. The first defines a structure needed to represent points in the 2-D Cartesian space. The second defines a structure that can be used to store the width and the height of a geometric figure, such as a rectangle. The third is a structure to contain the information about the filling color of a rectangle. The fourth type specification contains a supertype prefix, which extends the type for 2-D measurements to a type for 3-D measurements.

(x $\rightarrow$ Integer, y $\rightarrow$ Integer)
(width $\rightarrow$ Integer, height $\rightarrow$ Integer)
(rect $\rightarrow$ Rectangle, color $\rightarrow$ String)
Measurement2D:(depth $\rightarrow$ Integer)

□

In the language for defining objects, multiple values in a multiple occurrence field are simply given as an ampersand-separated list. It is not possible to distinguish a single occurrence field from a multiple occurrence field that happens to have a single value. We will need such a distinction in type specifications, as processing on the two kinds of fields are different in most cases (for example, in conformity checking and in pattern matching). We use a double arrow "$\rightarrow\!\!\rightarrow$" to indicate multiple occurrence fields.

Example 19: A synchronized version of a simple flip-flop is formed by adding a couple of new gates and a few more connections.

RSFlipFlop:(extraGate $\rightarrow\!\!\rightarrow$ Gate, extraConn $\rightarrow\!\!\rightarrow$ Connection)

□

The sufficient condition for an object to conform to a type is: 1) for each single occurrence field "$\mathbf{f} \rightarrow \mathbf{V}$", the object has the field **f** with one and only one value drawn from the admissible set defined by **V**, 2) for each multiple occurrence field "$\mathbf{f} \rightarrow\!\!\rightarrow \mathbf{V}$", the object has the field **f** with a subset of values drawn from the admissible set defined by **V**. Notice that the sufficient condition does not prohibit the objects from having extra fields in addition to what is required by their types, which is the *prescriptive typing* idea in [Maier85]. The type systems of Cardelli [Cardelli84] and of Ohori [Ohori87] are also prescriptive.

Type Names

Types have unique names. Type names are introduced by one of three ways: *equating* with (==, two consecutive equal signs), *subtyping* from (<) and *assigning* to (=), existing types or type specifications.

Definition 6: Each of the following three forms introduces a type name:

1) $\mathbf{T}_1 == \mathbf{T}_2$, where $\mathbf{T}_1$ and $\mathbf{T}_2$ are type names
2) $\mathbf{T}_1 < \mathbf{T}_2$,
3) $\mathbf{T} = \mathbf{t}$, where **T** is a type name and **t** a type specification

□

Clause 1 defines a new type that is *similar* to the definiens. Clause 2 defines a new type that is $\leq^d$-related to the definiens. In the third clause, the type specification is assigned a name.

Example 20: Assigning a type specification to the type name "Rectangle", as is permitted by Clause 3:

$$\text{Rectangle} = (\text{width} \rightarrow \text{Integer}, \text{height} \rightarrow \text{Integer})$$

A square can be simply defined as a rectangle (possibly with additional constraint that the width and the height must be the same).

Square $==$ Rectangle

□

Introducing a type name by equating it with an existing type name does not make them aliases. Instead, a type with an exact copy of the structure (including the position in the type hierarchy) with the definiens is created and is assigned the name. In the example above, the type "Rectangle" and the type "Square" are the same in every respect except that each of them has its own typeset and extension. They are independent types; and any future evolution by one of them would not affect the other. In other words, they are not $\in^{d}$-related.

Example 21: Using Clause 2, we define Student as a subtype of Person, as all students are persons.

Student $<$ Person

This definition introduces a new type with the name "Student" and relates it with the type "Person" using the $\leq^{d}$ relation. □

A new type introduced using Clause 2 also makes a copy of the structure from the definiens, like one defined using Clause 1. In example 21, the type "Student" would have an exact copy of the structure of "Person". There are differences between the two cases. The two types in Clause 1 are only casually related, in that one is a snapshot of the other. They are independent in every other aspect. The two types in Clause 2 are $\leq^{d}$-related. Consequently, future changes in one will affect the other. For example, addition of fields to the supertype is propagated downwards, and addition of instances to the typeset of the subtype is propagated upwards.

Example 22: Using Clause 3, new types are defined by explicitly giving a type specification. Here are a couple of examples for defining types by Clause 3, defining a "Point" and a "LayoutUnit" type, respectively.

Point = (x $\rightarrow$ Integer, y $\rightarrow$ Integer)
LayoutUnit = (rect $\rightarrow$ Rectangle, color $\rightarrow$ String)

□

Example 23: Clause 3 provides a way for defining subtypes with structure extensions. In this example, the type "Employee" is defined as a subtype of the type "Person". The new type ("Employee") contains two fields, "worksIn" and "salary", in addition to what it inherits from the supertype "Person".

Employee = Person:(worksIn $\rightarrow$ Department, salary $\rightarrow$ Integer)

□

4.4. Semantics

The semantics of the language for defining types are discussed from two perspectives: How a type definition relates the type to its instances, and how a type definition relates the type to other types.

Admissible Value Sets

Each value specification defines a range of values that a field of a complex object can take. These values form the *admissible value set* on the field.

Definition 7: A 1-1 function τ is assumed to exist that maps a type (name) to its typeset. The function ν assigns meanings to value specifications. For each $\mathbf{V} \in \mathbf{VS}$, $\nu[\![\mathbf{V}]\!]$ is the admissible value set defined by $\mathbf{V}$.

1) $\nu[\![\mathbf{T}]\!] = \tau(\mathbf{T})$
2) $\nu[\![\mathbf{V}_1 \neg \mathbf{V}_2]\!] = \nu[\![\mathbf{V}_1]\!] - \nu[\![\mathbf{V}_2]\!]$, where "$-$" denotes set difference
3) $\nu[\![\mathbf{V}_1 \wedge \mathbf{V}_2]\!] = \nu[\![\mathbf{V}_1]\!] \cap \nu[\![\mathbf{V}_2]\!]$
4) $\nu[\![\mathbf{V}_1 \vee \mathbf{V}_2]\!] = \nu[\![\mathbf{V}_1]\!] \cup \nu[\![\mathbf{V}_2]\!]$

□

The definition uses the typesets to characterize admissible value sets. We point out that

τ(Integer) = $\{0, \pm1, \pm2, ...\}$
τ(String) = the set of all finite strings over a fixed alphabet.
...
τ(All) = τ(Integer) $\cup$ τ(String) $\cup$ $\tau(\mathbf{T}_1) \cup \tau(\mathbf{T}_2) \cup ...$

Object Conformity

The formal notion of objects conforming to types is defined in a bottom-up manner. Each field **f** in a type **T** defines a primitive and necessary condition for an object to conform to **T**, $\kappa_{\mathbf{f}}$. In the following definition, we use the dot notation to indicate selection of object fields. For example, "aPerson.age" would retrieve the "age" value from a person object "aPerson", and "aPerson.children*" would retrieve all children of the person (as a set).

Definition 8: The construct *struct(*$\mathbf{T}$) is used to denote the set of fields defined in $\mathbf{T}$. Assume that $\mathbf{o}$ is an object.

1) $\mathbf{f} \rightarrow \mathbf{V} \in struct(\mathbf{T})$, then $\mathbf{o}\ \kappa_{\mathbf{f}}\ \mathbf{T}$, if $|\ \mathbf{o}.\mathbf{f}^*\ | = 1 \wedge \mathbf{o}.\mathbf{f} \in \nu[\![\ \mathbf{V}\ ]\!]$
2) $\mathbf{f} \twoheadrightarrow \mathbf{V} \in \mathbf{T}$, then $\mathbf{o}\ \kappa_{\mathbf{f}}\mathbf{T}$, if $\mathbf{o}.\mathbf{f}^* \subseteq \nu[\![\ \mathbf{f}\ ]\!]$

□

If a type contains a single occurrence field, an object must, for it to conform with the type, have the same field with precisely one value drawn from the right domain (Clause 1).

Example 9: As 100, 200 $\in \tau$(Integer), an object defined by the following expression: (x $\rightarrow$ 100, y $\rightarrow$ 200) is both κ_x- and κ_y-related to the type

Point = (x $\rightarrow$ Integer, y $\rightarrow$ Integer)

□

For a multiple occurrence field, say $\mathbf{f}$, the condition becomes that an object must have the field $\mathbf{f}$ with all the values drawn from the right domain. By using set notation to describe values for multiple occurrence fields we allow an empty set to be used as a legitimate value.

Example 24: Suppose that the type "Connection" is defined to contain a "connection" field that has multiple occurrences, as follows:

Connection = (connection $\twoheadrightarrow$ PortRef)

Then the object $\mathbf{o}\ \kappa_{\text{connection}}$ Port, where $\mathbf{o}$ is the object defined by:

(connection $\rightarrow$ PortRef:(portName $\rightarrow$ 'source-of-pullup',
component $\rightarrow$ aPullupTransistor)
& PortRef:(portName $\rightarrow$ 'drain-of-pulldown',

component $\rightarrow$ aPullDownTransistor))

where "aPullupTransistor" and "aPullDownTransistor" represent object identities. □

The sufficient conformity condition of a type is derived from its individual field conformity primitives using logical conjunctions.

Definition 10: We say that an object $\mathbf{o}$ *conforms* to a type $\mathbf{T}$, denoted by $\mathbf{o}\ \kappa\ \mathbf{T}$, if one of the following conformity conditions (corresponding to the different forms in which a type can be defined) is satisfied.

Case 1: $\mathbf{T} = (\mathbf{f}_1 \rightarrow \mathbf{T}_1, \ldots, \mathbf{f}_n \rightarrow \mathbf{T}_n)$:
if $\mathbf{o}\ \kappa_{\mathbf{f}_1}\ \mathbf{T} \wedge \ldots \wedge \mathbf{o}\ \kappa_{\mathbf{f}_n}\ \mathbf{T}$, then $\mathbf{o}\ \kappa\ \mathbf{T}$
Case 2: $\mathbf{T} = \mathbf{T}_1{:}\mathbf{t}_2$, where $\mathbf{t}_2$ is a type specification:
if $\mathbf{o}\ \kappa\ \mathbf{T}_1 \wedge \mathbf{o}\ \kappa\ \mathbf{T}_2$, where $\mathbf{T}_2 = \mathbf{t}_2$, then $\mathbf{o}\ \kappa\ \mathbf{T}$

□

From Case 1 and the definition of "<", we have the following lemma.

Lemma 1: If $\mathbf{T} < \mathbf{T}_1$ and if $\mathbf{o}\ \kappa\ \mathbf{T}$, then, $\mathbf{o}\ \kappa\ \mathbf{T}_1$ □

Example 25: Given the following two type definitions:

Rectangle = (width $\rightarrow$ Number, height $\rightarrow$ Number)
LayoutUnit = (rect $\rightarrow$ Rectangle, color $\rightarrow$ String)

and an object $\mathbf{o}$ described using the following expression

LayoutUnit: (rect $\rightarrow$ Rectangle: (width $\rightarrow$ 2, height $\rightarrow$ 6), color $\rightarrow$ 'RED')

then $\mathbf{o}\ \kappa$ LayoutUnit □

Type Specialization

Similarly, we break down the condition for specializing types into primitives on individual fields.

Definition 11: Suppose $\mathbf{S}$ contains a field $\mathbf{f} \rightarrow \mathbf{V}_1$, and $\mathbf{T}$ contains a field $\mathbf{f} \rightarrow \mathbf{V}_2$. We say $\mathbf{S}$ specializes $\mathbf{T}$ on $\mathbf{f}$, written as $\mathbf{S}\ \xi_{\mathbf{f}}\ \mathbf{T}$, if $\nu [\![\mathbf{V}_1]\!] \subseteq \nu [\![\mathbf{V}_2]\!]$.

□

The definition for type specialization is remarkably similar to that for object conformity.

Definition 12: Type $\mathbf{S}$ is a specialization of type $\mathbf{T}$, if one of the following conditions is satisfied.

Case 1: $\mathbf{T} = (\mathbf{f}_1 \rightarrow \mathbf{T}_1, \ldots, \mathbf{f}_n \rightarrow \mathbf{T}_n)$:
if $\mathbf{S}\ \xi_{\mathbf{f}_1}\ \mathbf{T} \wedge \ldots \wedge \mathbf{S}\ \xi_{\mathbf{f}_n}\ \mathbf{T}$, then $\mathbf{S}\ \kappa\ \mathbf{T}$
Case 2: $\mathbf{T} = \mathbf{T}_1 : \mathbf{t}_2$, where $\mathbf{t}_2$ is a type specification:
if $\mathbf{S}\ \xi\ \mathbf{T}_1 \wedge \mathbf{S}\ \xi\ \mathbf{T}_2$, where $\mathbf{T}_2 = \mathbf{t}_2$, then $\mathbf{S}\ \xi\ \mathbf{T}$

□

Lemma If $\mathbf{T} < \mathbf{T}_1$ and if $\mathbf{S}\ \xi\ \mathbf{T}$, then, $\mathbf{S}\ \xi\ \mathbf{T}_1$ □

A simpler version can be stated as: $\mathbf{S}$ is a specialization of $\mathbf{T}$ if the set of fields defined in $\mathbf{S}$ is a superset of those in $\mathbf{T}$, and for each field $\mathbf{f}$ of $\mathbf{T}$, $\mathbf{S}\ \xi_{\mathbf{f}}\ \mathbf{T}$.

Example 26: The following two types, one defining points in 3-D Cartesian coordinate space, the other defining points in 2-D Cartesian coordinate space, exhibit the type specialization relationship: ThreeDPoint ξ TwoDPoint.

TwoDPoint = (x → Integer, y → Integer)
ThreeDPoint = (x → Integer, y → Integer, z → Integer)

□

Note that any type **S** defined using one of the two subtyping forms is always a specialization of **T**:

$\mathbf{S} < \mathbf{T}$
$\mathbf{S} = \mathbf{T}$:TypeSpec

Example 27: We define "ThreeDPoint" as an extension of "TwoDPoint". In this case the assertion "ThreeDPoint ξ TwoDPoint" still holds.

ThreeDPoint = TwoDPoint:(z → Integer)

□

4.5. Chapter Summary

We presented a type system for TEDM in this chapter. The central concepts in this type system include the conformity conditions, defined by types and applied to objects, and the notion of type generalization. A language for defining types is proposed. Formal syntax and semantics of the type language were discussed.

CHAPTER 5

QUERYING AND MANIPULATING OBJECTS

Query and manipulation of database objects are supported by a command language. The statements of the language can be interactively executed, stored as database objects, or grouped to form *compound commands*, which can in turn be executed or stored as database objects. Normally, we would need two totally different interpretation schemes to handle the semantics of the language, since the interpretation of the command statements depends on how they are used. We need operational semantics for the commands that are executed immediately, and translational semantics for the commands that are to be stored as objects. It turns out there is no such complication in TEDM, as commands are uniformly interpreted as database objects. The pattern-matching semantics of the commands are kept intact by abstract objects. The operational semantics of the commands are preserved using a special class of database objects — command-defining objects (or CDOs).

This chapter describes a language for defining database commands. The study on the meanings of the commands in this chapter explores only the partial semantics, focusing on the pattern-matching aspect and on the operation aspect. The next chapter will supply the translational aspect, to complete the study on command semantics.

The language for defining commands is also a dialect (of a subset) of the canonical object language. Every command defines a CDO; but there are CDOs that cannot be defined in the command language. As CDOs are database objects, they can also be defined by an object term in the canonical language. Furthermore, all CDOs can be defined as object terms. However, the command language usually gives more succinct descriptions of CDOs than the object language.

This chapter contains five sections. Section 1 gives an informal description of the command language. Section 2 discusses two important notions: *command patterns* and *pattern variables* in commands, Section 3 describes *command heads*, which denote imperative operations on class of objects defined by command patterns. Section 4 discusses compound commands as a grouping construct on multiple commands. Section 5 describes the semantics of command execution.

5.1. Commands and Their Informal Semantics

A database command consists of a *pattern* and an *action*. The pattern and the action of a command are separated using the symbol "<==". Patterns as well as actions in turn are built from terms or term-like constructs. Either the pattern or the action can be null. A null action indicates a null operation. A null pattern, on the other hand, indicates unconditional execution of the action.

Although a command resembles a production rule in syntax, their interpretations differ. For example, in a production system, the pattern part of a rule is a predicate with only two possible interpretations, "true" or "false". The pattern part of a command is a *constructive* predicate, not only possessing a truth value but also denoting a set of database objects that establish the truth value. Also, the action part of a rule usually only changes the state of a single entity. The action part of a command, on the other hand, operates on a set of objects (the set defined by the pattern).

Before the syntax of the command language is introduced, we use the following abstract form to describe a command.

$$\text{Action}[X_1, \ldots, X_n] \Longleftarrow \text{Pattern}[Y_1, \ldots, Y_m],$$

where X_i's are distinct variables, so are Y_j's; X_i's and Y_j's may overlap. We assume that the set of variables appearing in the pattern always contains those appearing in the action as a subset. That is, $\{X_i \mid i = 1, \ldots, n\} \subset \{Y_j \mid j = 1, \ldots, m\}$. With this assumption, we can informally state the semantics of such a command as follows. First, the pattern part is used to match the database to get variables bound to database objects. A binding is a list of tuples of the form $[b_1, b_2, \ldots, b_k]$, where $b_i = [Y_{i1}{:}o_{i1}, \ldots, Y_{im}{:}o_{im}]$, for database objects o_{ij} and some finite value k. The list is called a *binding matrix*, and each row a *binding vector*. Second, the action part is used to operate on the database objects in the binding matrix. The action is applied to all binding vectors. The mode of the operation, namely, parallel or sequential, is unspecified. Actions in a command can be predefined simple system procedures, for example, for viewing or updating objects. They can also be user-defined compound commands.

Example 28: Suppose rectangle objects are defined by the origin and the corner points. This command browses the corner points of the rectangles (whose origin) situated at a fixed point (2, 3)

$$\begin{aligned}\text{View}[C] \Longleftarrow \text{rects} \rightarrow \text{Rectangle:R}(&\text{origin} \rightarrow \text{Point:P}(x \rightarrow 2,\ y \rightarrow 3),\\ &\text{corner} \rightarrow \text{Point:C})\end{aligned}$$

□

In this example, "rects" is the name of an extension, a set of objects of a given type. "C", "P" and "R" are command or pattern variables, which are bound during pattern-matching. Although they resemble placeholders in the object language, their treatment will be sightly different. "View" is a predefined operation that displays complex objects in a nested text form; and "rects" is a collection of rectangle objects. To execute the command, the processor looks for "Rectangle" objects with the given description (namely, situated at the point (2,3)) from the collection "rects". For each such rectangle object matching the description, bind its object identifier to the variable "R", its origin point to "P" and its corner point to "C". The "View" operation then display the corner point objects. Assume that there are two rectangle objects in the collection, whose definitions are as follows (we use lower case variables to represent object identifiers):

```
Rectangle:r1(origin → Point:p1(x → 2, y → 3),
             corner → Point:q1(x → 5, y → 10))
Rectangle:r2(origin → Point:p2(x → 2, y → 3),
             corner → Point:q2(x → 6, y → 15))
```

Then the binding matrix produced by pattern matching would be the following:

```
[R:r1, P:p1, C:q1]
[R:r2, P:p2, C:q2]
```

And the result of the "View" operation is the following display:

```
Point:q1(x → 5, y → 10)
Point:q2(x → 6, y → 15)
```

5.2. Patterns and Variables

Patterns and variables in the patterns play important roles in commands. Object terms (in the object language) and patterns are similar in syntax. In conventional approach they would differ in semantics: The former denote database objects; the latter denote operations (matching) on database objects. But this discrepancy in semantics no longer exists in TEDM. As we discussed in Chapter 3, we extended the databases to accommodate abstract objects with pattern-matching semantics. The implication, then, is that patterns are object terms, and that a TEDM database is capable of storing both data and pattern-matching operations. In other words, the syntax and the semantics of the patterns are subsumed by those of object terms and the pattern-matching semantics of the abstract objects.

For the discussion in the remainder of this chapter, we define a special syntax for patterns. In general, patterns contain simple constants as well as arbitrary literals (or structured constants).

Definition 13: The set of literals, **Lit**, is the following.

1) $d \in$ **Lit**, where d is a constant
2) $(g_1 \rightarrow l_1, \ldots, g_m \rightarrow l_m) \in$ **Lit**,
   where g_i's are field labels and $l_i \in$ **Lit**
3) $T{:}l \in$ **Lit**, where T is a type name and l is a literal define by 1) or 2)

□

We assume that there is a domain of variables.

Definition 14: The set of simple patterns, **SimPat**, satisfy the following.

1) $T{:}V \in$ **SimPat**, where T is a type name and V a variable
2) $T{:}V(g_1 \rightarrow p_1, \ldots, g_m \rightarrow p_n) \in$ **SimPat**, where $p_i \in$ **SimPat** or $p_i \in$ **Lit**

□

A pattern can be as simple as a typed variable (Clause 1). In this case, the pattern matches all concrete objects of the given type. Complex patterns contain references to simpler patterns or arbitrary literals (Clause 2). The simpler patterns participate in pattern matching on subobjects and may produce additional bindings. The literals are used merely as selection criteria during pattern matching.

The next definition extends the previous one by allowing multiple simple patterns to be grouped to form compound patterns.

Definition 15: The set of well formed patterns, **Pat**, is the set satisfying the following.

1) $p \in$ **Pat**, where $p \in$ **SimPat**
2) $p_1, p_2 \in$ **Pat**, where $p_1 \in$ **Pat** and $p_2 \in$ **Pat**

□

Example 29: The following pattern matches Student objects with a "major" field and the value of the field is a constant 'CS', and additionally, the students must also be TAs.

Student:S(major → 'CS'), TeachingAssistant:S

□

5.3. Command Actions

Command heads denote imperative operations on the database objects obtained through pattern matching. There may be multiple heads in the action part of a command. In this case, operations denoted by the heads are applied to the objects in a sequential order. Basic operations, such as *adding types* and *adding fields*, are all described using object terms. We describe these basic operations.

Adding Objects to Types

As a database evolves, an object may acquire new types, lose or change existing types. To assign a type to an object (add the object to the typeset of the type), we use an object term of the following form.

T:V

The meaning of such a command head is that each object bound to the variable **V** during pattern-matching phase gets a new type **T**, provided the object conforms to **T**.

Example 30: The command below adds the type "TeachingAssistant" to students who also teach courses. (Presumably, each student that matches the pattern would also conform to the type "TeachingAssistant".)

$$\text{TeachingAssistant:S} \Longleftarrow \text{student} \rightarrow \text{Student:S(teaches} \rightarrow \text{C), Course:C}$$

□

Deleting Objects from Types

Types can be removed from objects as well. For example, after one graduates from school, ones "Student" type is no longer needed and should be changed accordingly (say, becoming an "Alumnus").

T:~ V

The meaning of such a command head is that objects bound to the variable **V** lose the type **T**. In general, removing types from objects does not cause semantic problems.

Example 31: When an employee sells all of his shares of the company stock, the employee is no longer a stockholder of the company.

CompanyStockHolder:~ E $\Longleftarrow$ employee $\rightarrow$ Empolyee:E(share $\rightarrow$ 0)

□

Adding Fields to Objects

Although facilities for objects to change types are provided, study shows that objects changing types is not a high frequency event in most application databases; many objects never change types once created [Zdonik87]. On the other hand, the next two operations on objects, adding and deleting fields, happen more frequently.

Adding fields to objects is denoted using constructs similar to an object description with a list of fields.

$$\mathbf{V}(\mathbf{f}_1 \rightarrow \mathbf{w}_1, \ldots, \mathbf{f}_n \rightarrow \mathbf{w}_n)$$

Example 32: At the end of a term, add a 'passed' field with a Course object as its value to all students who were enrolled in the course and got a pass grade.

```
S(passed → C) <== enrollment →
        Enrollment:(course → C,
                    student → Student:S,
                    grade → 'Pass')
```

□

There are several semantical issues to consider in an "add field" operation. If an object does not have the field to be added, the operation adds the field to the object. If the object has the field as is declared by its type, then if the field is single occurrence, the field is updated; if the field is multiple occurrence, the field value is added to the object as an additional value. Updating the value of a multiple occurrence field is made possible by the following convention. We assume if, in an adding field operation, a field to be added also appears in the pattern, with value bound to concrete objects, then it means updating the value of the field. In this case, a value bound in the pattern is replaced by a value in the action.

Example 33: New offices on the second floor are assigned phone numbers.

```
O(phoneNo → P) <==
        office → Office:O(floor → 'second'),
        phone → PhoneAssignment:A(phoneNo → P, office → O)
```

□

Example 34: Phone numbers for offices on the second floor changed their prefixes.

```
P(prefix → 777) <==
        office → Office:O(floor → 'second'),
        phone → PhoneAssignment:
            A(phoneNo → PhoneNo:P(prefix → 666), office → O)
```

□

Another obvious way to do update is to use a delete operation followed by an add operation.

Deleting Fields from Objects

Deleting fields from objects is denoted using the following general constructs.

$$\mathbf{V} \sim (\mathbf{f}_1 \rightarrow \mathbf{w}_1, \ldots, \mathbf{f}_n \rightarrow \mathbf{w}_n)$$

An object may no longer conform to a type if a field defined in the type is deleted from the object. In this case, the database contains invalid information. We can simply reject operations that would create inconsistencies. We can also deal with this problem using compound commands, where a number of commands are grouped together and are treated as a unit of operation. For example, after deleting fields from an object, a logical operation is to adjust the types for the object.

Example 35: Delete users' accounts on systems where the users have been given a cputime quota of 0.

```
Student:U ~ (account → A) <==
        quote → Quota:Q(system → S, user → ID, cputime → 0),
        student → Student:U(userID → ID),
        system → System:S(chargeNo → A)
```

□

There is a confusion between deleting only the value or the entire field. Our convention is: For a multiple occurrence field, it always means deleting values. For a single occurrence field, it always means deleting the entire field.

Creating New Objects

The object creation operation introduces new objects into databases, using the following general form:

$$\mathbf{T}{:}* \,(\mathbf{f}_1 \rightarrow \mathbf{w}_1, \ldots, \mathbf{f}_n \rightarrow \mathbf{w}_n)$$

The asterisk "*" indicates a new objects identifier. It can be embedded in one of $\mathbf{w}_1$, ..., $\mathbf{w}_n$. In that case, it denotes creating a subobject. New objects are created with given type(s). The conformity condition is checked to make sure the given types are correct.

Example 36: This command creates a "ContactCut" object for each "SimpleLayout" object that is placed 30 units away vertically from the reference point.

ContactCut:∗(rect → Rectangle:*(width → 2, height → 2)) <==
simpleLayout → SimpleLayout:(posn → (y → 30))

□

Temporary types (not yet defined types) can be used for the purpose of collecting newly created objects. Their definitions can be automatically generated upon commit.

New objects can also be placed into an extension upon its creation, by supplying the name of the extension.

$\langle\text{extension}\rangle \rightarrow \mathbf{T}:* (\mathbf{f}_1 \rightarrow \mathbf{w}_1, \ldots, \mathbf{f}_n \rightarrow \mathbf{w}_n)$

Example 37: This command is the same as the one in the previous example, except that the new "ContactCut" objects are placed in the extension "cut".

cut → ContactCut:∗(rect → Rectangle:*(width → 2, height → 2))
<== simpleLayout → SimpleLayout:(posn → (y → 30))

□

The extensions are created if these name is not present already. The update semantics of an extension are the same as for a multiple occurrence field. To remove an object from an extension, one uses an expression of the form

$\langle\text{extension}\rangle \sim\!\rightarrow \mathbf{w}$

To indicate sharing of a newly created object by two or more other objects, we can place a variable after the asterisk in the construct denoting object creation, and use the same variable elsewhere to refer to the new object.

Example 38: This command creates "DoubleBox" objects that contain two boxes with a shared reference point.

```
DoubleBox:*(box1 → Box:*(side → W,
                          reference → Point:*P(x → X, y → Y)),
            box2 → Box:*(side → H,
                          reference → Point:P)) <==
        simpleLayout → SimpleLayout:(rect → Rectangle:
                                        (width → W, height → H),
                                   posn → Point:(x → X, y → Y))
```

□

Compound Commands

Compound commands allow the user to group semantically related simple database manipulation functions together and execute them as a single transaction. The general syntax for using a compound command in command head is (V_i's are arguments):

CompoundCommandName[V_1, ..., V_k]

Example 39: Suppose "HireEmployee" is a compound command that takes one argument, then the command head

```
HireEmployee[P] <== candidate →
        Person:P(education → "PhD", specialty → "CS",
                 proficiency → "High")
```

would mean hiring qualified candidates. The compound command "HireEmployee" would perform what is necessary for hiring new employees. □

Some compound commands are predefined. Typically, "View" is defined for examining objects and "Install" is defined for committing updates.

5.4. Defining Compound Commands

Compound commands are groups of individual command statements. They are a control abstraction mechanism like procedures in programming languages. Individual commands in a compound command are executed sequentially. However, we require that either all of the individual commands are executed or none of them is executed. In other words, the execution of a compound command is atomic.

Atomicity is needed to guarantee the consistency of databases. A database is in a consistent state if its objects and types satisfy all the constraints. Constraints can be application-independent or application-dependent. Application-independent constraints are imposed by the system. In our case, the conformity condition and specialization condition are application-independent. Application-dependent constraints are user-defined and embed specific application domain knowledge. For example, in a VLSI CAD/CAM design database, a possible application-dependent constraint is that the "width" of "ContactCut" object can be no smaller than 2 λ. Currently, we are only concerned with constraints of the first kind: application-independent constraints. A database can potentially be in a transient inconsistent state. For example, after executing the statement

```
Manager:E <== employee →
        Employee:E(name → PersonName:(first → 'Joe', last → 'Dow'))
```

the object conformity condition may be violated, assuming that type "Manager" is defined as a subtype of type "Employee" with an additional field "manages". The statement adds some "Employee" object(s) to type "Manager". Since object(s) resulting from pattern matching may not contain a "manages" field, these objects do not conform to type "Manager". In this situation, several related commands collectively may ensure consistency; and they can be grouped together to form a compound command. As the execution of a compound command is atomic, transient inconsistent database states are invisible from outside the command.

Example 40: We can define a compound command, call it "PromoteToManager" and then invoke it using

```
PromoteToManager[E, D] <== employee →
      Employee:E(name → PersonName:(first → 'Joe', last → 'Dow')),
      department → Department:D(name → 'Sales')
```

to promote "Joe Dow" and still maintain a consistent view on the database. The compound command definition is as follows.

```
PromoteToManager[Employee:E, Department:D]
{
    Manager:E <== Employee:E;
    E(manages → D)  <== Employee:E, Department:D
}
```

□

The syntax for compound commands is much like a procedure in other languages. Each compound command consists of a number of database commands. In addition, a compound command may also contain arguments and local variables. The generic form for compound commands is the following:

```
CompoundCommandName[ T1:V1, ..., Tn:Vn ]
{
    S_1:U_1, ..., S_m:U_m;
    Action_1 <== Pattern_1;
    ...
    Action_k <== Pattern_k;
}
```

where Ti's and Si's are type names, Vi's are arguments, Ui's are local variables.

5.5. Command Semantics

As was mentioned, the semantics of the commands have two perspectives: the role in manipulating database objects and the role of being part of the database itself. Although the two roles are unified in TEDM as database objects, understanding them separately may improve the clarity of the presentation. We concentrate on the first role of the database commands (or their *active semantics*) in this section.

Semantics of Patterns

Binding matrices play an important role in defining the semantics for patterns. Each pattern defines a set of bindings when it is matched against database objects. The bindings of complex patterns are obtained from those of simpler ones by combining their binding matrices in two ways, *join extension* and *literal selection*.

We discuss join extension first. Given two binding matrices

$$\begin{aligned} M_1 = {} & [X_1{:}x_{11}, \ldots, X_n{:}x_{n1}, Z_1{:}z_{11}, \ldots, Z_k{:}z_{k1}] \\ & \ldots \\ & [X_1{:}x_{1m}, \ldots, X_n{:}x_{nm}, Z_1{:}z_{1m}, \ldots, Z_k{:}z_{km}] \\ \text{and } M_2 = {} & [Y_1{:}y_{11}, \ldots, Y_s{:}y_{s1}, Z_1{:}w_{11}, \ldots, Z_k{:}w_{k1}] \\ & \ldots \\ & [Y_1{:}y_{1t}, \ldots, Y_s{:}y_{st}, Z_1{:}w_{11}, \ldots, Z_k{:}w_{kt}] \end{aligned}$$

where X_i's and Y_j's are distinct pattern variables, the join extension of the two binding matrices is defined as follows.

Definition 16: The join extension of M_1 and M_2, denoted by $M_1 \bowtie M_2$, is a binding matrix consisting of rows: $[X_1{:}x_1, \ldots, X_n{:}x_n, Z_1{:}z_1, \ldots, Z_k{:}z_k, Y_1{:}y_1, \ldots,$

Y_s:y_s], such that [X_1:x_1, ..., X_n:x_n, Z_1:z_1, ..., Z_k:z_k] is a row of M_1 and [Y_1:y_1, ..., Y_s:y_s, Z_1:z_1, ..., Z_k:z_k] is a row of M_2. □

Note that neither the order of rows nor the order of columns are significant in the binding matrices. Therefore, the above join operation on binding matrices can easily be shown associative and commutative [Maier83], which allows us to omit parentheses for two or more join operations or to shuffle the operands of a join.

The literals provide another way for manipulating binding matrices — they eliminate the rows with components that do not match literals. A row is eliminated from a binding matrix if the row contains a variable bound to a database object, but the variable has a literal that the database object does not match. Literals match database objects in a straightforward way: exact match.

Definition 17: The notion of *matching literally*, **lit-match** ($\subseteq$ **Lit** $\times$ **G**), is defined as follows.

1) d **lit-match** **v**,
   if the simple object **v** is the interpretation of constant d
2) ($g_1 \rightarrow l_1$, ..., $g_m \rightarrow l_m$) **lit-match** Σ **[** ($\mathbf{f}_1 \rightarrow \mathbf{w}_1$, ..., $\mathbf{f}_n \rightarrow \mathbf{w}_n$) **]**,
   if $\forall g_i \exists \mathbf{f}_j$ ($g_i = \mathbf{f}_j \wedge l_i$ **lit-match** Σ **[** $\mathbf{w}_j$ **]**)
3) T:l **lit-match** Σ **[** $\mathbf{T}_1$ ... $\mathbf{T}_n$:**w** **]**,
   if T = $\mathbf{T}_i$ for some i and l **lit-match** Σ **[** **w** **]**

□

A literal selection trims a binding matrix M by eliminating rows from M, using **lit-match** as the criterion.

Definition 18: Given a binding matrix M = [V_1, ..., V_n], the *literal selection* of M by literals l_{I1}, ..., l_{Ik}, σ(M; l_{I1}, ..., l_{Ik}), is M′ consisting of the rows of M satisfying l_{I1} **lit-match** $V_{I1} \wedge \cdots \wedge l_{Ik}$ **lit-match** V_{Ik}. □

Definition 19: The notion of *simple match*, **sim-match**, is defined as follows.

1) $T{:}V$ **sim-match** $\Sigma[\![\ \mathbf{T_1} \ldots \mathbf{T_k}{:}\mathbf{w}\ ]\!]$, if $T = \mathbf{T_i}$ for some i
2) $T{:}V(g_1 \rightarrow p_1, \ldots, g_m \rightarrow p_m)$ **sim-match**
 $\Sigma[\![\ \mathbf{T_1} \ldots \mathbf{T_k}{:}(\mathbf{f_1} \rightarrow \mathbf{w_1}, \ldots, \mathbf{f_n} \rightarrow \mathbf{w_n})\ ]\!]$,
 if $\forall g_i \exists \mathbf{f_j}\ (g_i = \mathbf{f_j}$
 $\wedge\ (p_i$ **lit-match** $\Sigma[\![\ \mathbf{w_j}\ ]\!] \vee p_i$ **sim-match** $\Sigma[\![\ \mathbf{w_j}\ ]\!]\ ))$

□

Definition 20: We use a semantic function, π, to assign meanings to patterns. The interpretation of well-formed patterns is defineds as follows. (Recall τ maps types to typesets.)

1) $\pi[\![\ T{:}V$ **sim-match** $\Sigma[\![\ \mathbf{w}\ ]\!]\ ]\!] = [V{:}v], \forall v \in \tau(T)$
 where **w** is an arbitrary object term,
2) $\pi[\![\ T{:}V(f_1 \rightarrow p_1, \ldots, f_n \rightarrow p_n, g_1 \rightarrow l_1, \ldots, g_m \rightarrow l_m)$ **sim-match**
 $\Sigma[\![\ \mathbf{w}\ ]\!]\ ]\!] = \sigma([V, \mathit{var}(p_1) \ldots \mathit{var}(p_n)]; l_1, \ldots, l_m)$,
 where $p_i \in$ **Pat**, $l_i \in$ **Lit**, $\mathit{var}(p)$ is the top-level variable in p
3) $\pi[\![\ p_1, p_2\ ]\!] = M_1 \bowtie M_2$, where $p_1, p_2 \in$ **Pat**,
 and M_1 and M_2 their binding matrices respectively.

□

Note that the binding matrix M obtained using Clause 1 is a single column matrix. M may contain multiple rows, as the pattern matches all objects of a given type.

Semantics of Command Heads

The discussion on the semantics for the command heads is simplified by only considering single occurrence fields. The result can be generalized to include multiple occurrence fields. Also, for the basic operations (without compound commands), we require that each of them preserve consistency.

Some notational convention. Formal meanings of command heads are defined by δ, an update function on databases. A k-row binding matrix is denoted by M^k or simply M when k is irrelevant. The i-th row (the i-th binding vector) in a binding matrix M is denoted by M_i, for some $i \leq k$. The notation $M|_{V1, ..., Vn}$ is used to denote the restriction of M on variables V1, ..., Vn: a submatrix obtained by removing column of bindings other than the ones for V1, ..., Vn. We use x' to denote the value for x after an operation. The symbol ":=" is used to make an assignment (for example, $x' := x$). A semicolon is used to separate two sequential semantical actions. The minus sign "–" is used to denote set difference. Also recall that $\in^d$ denotes a declared membership, $\in^i$ an inferred membership, $\leq^i$ an inferred subtype and $\leq^d$ a declared subtype. ($\leq^{d*}$ is the reflexive and transitive closure of $\leq^d$.)

Adding Types

Adding a type to an object means that the object becomes a member of the typeset of the type, and a member of the supertypes, provided that object conforms to the type.

Definition 21: $\delta[\![\ T{:}V\ ]\!]\ M = \tau'(t) := (\tau(t) \cup \{ o \mid o \in M|_V \wedge o \in^i T \}), \forall t\ (T \leq^{d*} t)$ □

Deleting Types

When deleting a type from an object, we make sure the object is also deleted from the subtypes.

Definition 22: $\delta [\![\text{ T:}\sim \text{V }]\!] \text{ M} = \tau'(\text{t}) := (\tau(\text{t}) - \text{M}|_{\text{V}}, \forall \text{ t } (\text{t} \leq^{d*} \text{T}))$ □

Deleting Fields

The function *struct* is used to denote the fields of an object, and the function *state* is used to denote the state (fields and values) of an object. Deleting field operations delete one or more field from an object (not just values). A field is not deleted from an object if the resulting object would violate a type conformity condition.

Definition 23: The v_j's are ignored when interpreting delete-field operation.

$$\begin{aligned}&\delta [\![\text{ V}\sim (\text{f}_1 \rightarrow \text{v}_1, \ldots, \text{f}_n \rightarrow \text{v}_n)]\!] \text{ M} =\\ &\quad \text{for each } \text{o}_i \in \text{M}_i|_{\text{V}} \text{ and each j, j} = 1, \ldots, \text{n:}\\ &\qquad \textit{struct}'(\text{o}_i) := \textit{struct}(\text{o}_i) - \{ \text{ f}_j \text{ } \}, \text{ if } \text{o}_i' \in^d \text{ t}, \forall \text{ t } (\text{o}_i \in^d \text{ t})\end{aligned}$$

Note that $\text{M}_i|_{\text{V}}$ is a one-element matrix. □

Adding Fields

The semantic function, Σ, is used to interpret constants. A field is added to an object if the object does not contain the field already. Otherwise, it replaces the value of the existing field, if the value is admissible (according to type definitions).

Definition 24: Let $\phi [\![\text{ v}_j]\!]$ be i) $\Sigma [\![\text{ v}_j]\!]$, if v_j is a constant or ii) $\text{M}_i|_{\text{v}_j}$ if v_j is a variable. Let *state*(o) $\oplus$(f: v) be i) *state*(o) $\cup$ {(f: v)}, if f $\notin$ *struct*(o), ii) *state*(o) $-$ {(f: o.f)} $\cup$ {(f: v)}, if f $\in$ *struct*(o) and v is an admissible value of the field f, or iii) *state*(o), otherwise.

Then,

$$\begin{aligned}&\delta [\![\, V(f_1 \rightarrow v_1, \ldots, f_n \rightarrow v_n) \,]\!] \, M = \\ &\quad \text{for each } o_i \in M_i|_V \text{ and each } j, j = 1, \ldots, n: \\ &\qquad \mathit{state'}(o_i) := \mathit{state}(o_i) \oplus (f_j\!: \phi [\![\, v_j \,]\!])\end{aligned}$$

□

Creating Objects

New objects are created by obtaining a new object identifier (different from any of the existing ones), and assigning fields and values to the object.

Definition 25: Let $\phi [\![\, v_j \,]\!]$ be i) $\Sigma [\![\, v_j \,]\!]$, if v_j is a constant, ii) $M_i|_{v_j}$ if v_j is a variable, or iii) $\delta [\![\, v_j \,]\!]$, if v_j is request for creating subobject. Then,

$$\begin{aligned}&\delta [\![\, T{:}{*}(f_1 \rightarrow v_1, \ldots, f_n \rightarrow v_n) \,]\!] \, M = \\ &\quad \mathit{state}(o) := \{(f_1\!: \phi [\![\, v_1 \,]\!]), \ldots, (f_n\!: \phi [\![\, v_n \,]\!])\}) \\ &\quad \text{where o is a new object identifier.}\end{aligned}$$

□

Since collections (or extensions) can be viewed as a field in the unique top-level database object, the root object, their update semantics are the same as for multiple occurrence fields. Generalizing the update function δ to handle multiple command actions amounts to applying δ on each of the actions in turn.

Semantics Summary

To summarize the active semantics for the commands, we use **dbState** to denote the state of the database. We also use Σ (the semantic function for object terms) to interpret database commands:

$$\Sigma [\![\text{<Action>} \Leftarrow \text{<Pattern>}]\!] =$$
$$\mathbf{dbState'} := \delta [\![\text{<Action>}]\!] \, \pi [\![\text{<Pattern>}]\!] (\mathbf{dbState})$$

Note that the new state **dbState'** is not a new copy of the old state with changes. **dbState'** is obtained by updating the old state **dbState** directly. (To make this point more explicit, we can think of **dbState** as a database object, the *root object* from which all other objects are reachable.)

5.6. Chapter Summary

In this chapter, we discussed mechanisms for querying and manipulating database objects in TEDM. A rule-like language was presented. The pattern matching style of query processing was described. We also studied the active semantics of the command language.

CHAPTER 6

MATCHING WITH ABSTRACT OBJECTS

Commands are stored as CDOs. The key notion that makes storing commands possible is that of *abstract objects*, structural templates with flexible components. Abstract objects are created as a result of interpreting object terms containing object tags, a notion explored in Chapter 3.

Other than containing variable components, the structure of abstract objects is not unlike that of concrete objects. Abstract objects also contain fields with values. The value of a field can be a simple object or a complex object. The value can be an abstract object or a concrete object. Abstract objects are also typed or multiply typed. Nevertheless, there are considerable differences between the semantics of abstract objects and concrete objects. The centrol notion in abstract objects is that of *pattern matching*. The pattern-matching semantics dictate how to treat the other aspects of abstract objects. For example, although abstract objects are typed, the type conformity condition does not apply to them. Saying that an abstract object is of type **T** means that the abstract object can match at most the objects in the typeset of **T**. (Note that subtypes are implied.)

As abstract objects are interpretations of the object tags, which in turn are introduced as a variant of the variable concept in logic, the pattern-matching semantics are comparable with the binding of variables to semantical entities. But the abstract objects are more powerful than simple variables, as matching with abstract objects is based on given patterns; and even the simplest pattern has the power of a variable.

This chapter studies the properties of abstract objects. We discuss the structure and the semantics of this new class of objects. Most of all, we explore the pattern-matching semantics of the abstract objects. We also describe two uses of abstract objects, in type-defining objects and in command-defining objects. The chapter is organized as follows. Section 1 reviews the basic concepts in pattern matching. Section 2 defines pattern matching for TEDM using *match functions*. It also describes two classs of restricted match functions: decomposition maps and generalized decomposition maps. The former reflect the pattern-matching semantics of abstract objects on concrete objects; and the latter reflect the pattern-matching semantics of abstract object on abstract objects. Section 3 discusses an application of abstract objects, where type structures are stored as abstract objects in type-defining objects. With this representation, type conformity checking in the system can be treated as an ordinary pattern-matching problem. Section 4 is another application of abstract objects, where we discuss representing commands as objects. The chapter summary is given in Section 5.

6.1. Pattern Matching: The Basics

Pattern matching is the foundation of query processing and object manipulation in TEDM. The resulting database language supports set-oriented querying as well as update for an object-oriented data model. Compared with other similar efforts (such as Zaniolo's extension to the relational algebra [Zaniolo85], Kuper and Vardi's logic data model[Kuper84], Bancilhon and Khoshafian's object calculus [Bancilhon86] and Beeri's object logic [Beeri87]), TEDM's approach is natural and flexible. For example, patterns can be freely composed, resulting in queries with arbitrary complexity and precision.

Furthermore, patterns are part of the database itself, complex pattern objects can be dynamically created by connecting existing simple pattern objects.

Pattern matching can be thought of as a special case of unification, a computation mechanism used by Prolog for resolution. In pattern matching, two kinds of entities are distinguished: the patterns (or templates) for matching and the objects to be matched. The roles of the two are different and asymmetric. In matching a pattern with an object, the constants in the pattern are compared with their counterparts in the object, and the variables in the pattern are substituted by their counterparts in the object. If the match is successful (the definition for a successful match would vary from system to system), the substitution (or binding) contains useful information about the structure of the matched object and is said to be the answer of the match.

The binding of a successful match contains handles to the matched objects, which TEDM uses for associative retrieval. Updates on the objects are also made possible by the set of handles. Pattern matching in TEDM is set-oriented: An answer substitution is a set of bindings, each being the result of a successful match. The set of objects in the answer substitution are stored using binding matrices, which also keep the information about the correspondence of an variable and its bindings.

We take one more significant step in adopting the pattern-matching technique for query processing: We store the patterns and even commands as objects. We motivate this approach by contrasting it with more traditional ones. It is customary to treat database objects and the operations on them separately as two distinct entities: The databases objects are passive entities stored in databases; the operations are active and are processed interactively (or embedded in a host language). For example, the traditional dichotomy of a conceptual data model is a data definition language and a data manipulation language. They are clearly separated with non-overlapping roles. The DDL is used to define, constrain and possibly populate a database; whereas the DML is used to describe operations on the database. It is difficult to define database operations in terms of database objects (using a DML), as operations are foreign to the data model. Consequently, it is difficult to manipulate database commands in terms of themselves. Views are usually supported as an add-on feature of a DML and are processed differently.

TEDM addresses this shortcoming by storing commands as objects. Compared with other solutions (such as INGRES and POSTGRES [Stonebraker84, Stonebraker86], in which QUEL commands are storable attributes as text strings), TEDM's approach is abstract and powerful. For example, INGRES treats QUEL commands as textual strings, or possibly cached in a compiled form or byte codes. In either case, they are nondecomposable atomic values and there is no way to look inside them. TEDM on the other hand, associates commands with a semantical structure from the underlying interpretation space and stores them as objects using CDOs. The storage format of CDOs preserves the structural semantics of the commands. The significance of storing commands as database objects is the following: Firstly, commands can be manipulated as ordinary objects. They can be queried in a form similar to what they mean, especially for command patterns and their pattern-matching semantics. Secondly, as objects, they can be freely composed with other objects, promoting reuse and the dynamic construction of commands. Thirdly, pre-tested queries can be packaged with application databases.

Abstract objects are introduced primarily for storing commands. They are a new class of objects with identifiers distinguishable from those of concrete objects. In the definition of the object space **G** in Chapter 3, the set of abstract object identifiers was denoted by $\mathbf{ID}_A$. We point out that there is a big difference in our treatment of complex objects from the others: We view complex objects as semantical entities with both identities and structures (or as graphs), not just identities (or nodes in graphs). We use a function, ι: $\mathbf{G} \rightarrow \mathbf{ID}_{ACD}$, to extract object identifiers from complex objects as follows.

1) $\iota(\mathbf{v}) = \mathbf{v}$, if $\mathbf{v} \in \mathbf{ID}_{ACD}$
2) $\iota(\mathbf{v}) = \mathbf{i}$, if $\mathbf{v} = (\mathbf{i}, \{(\mathbf{f}_1 : \mathbf{w}_1), ..., (\mathbf{f}_n : \mathbf{w}_n)\})$

We can use ι to characterize abstract objects succinctly — an object **o** is an abstract object if and only if $\iota(\mathbf{o}) \in \mathbf{ID}_A$. Furthermore, we extend ι to ι^* to operate on objects and return all their components.

1) $\iota^* (\mathbf{v}) = \{ \iota(\mathbf{v}) \}$, if $\mathbf{v} \in \mathbf{ID}_D$
2) $\iota^* (\mathbf{v}) = \{ \iota(\mathbf{v}) \} \cup \iota^*(\mathbf{w}_1) \cup ... \cup \iota^*(\mathbf{w}_n)$, if $\mathbf{v} = (\text{id}, \{(\mathbf{f}_1 : \mathbf{w}_1), ..., (\mathbf{f}_n : \mathbf{w}_n)\})$

6.2. Matching on Objects

Pattern matching relies on structural "similarity" to establish a relationship between a template and an object: The template matches the object. The notion of *similarity* adopted by TEDM is made precise by *match functions* (denoted by μ), 1-1 mappings from amongst objects.

Definition 26: Given two objects:

$$\mathbf{v}_1 = (\mathbf{i}_1, \{(\mathbf{f}_{11} : \mathbf{w}_{11}, ..., (\mathbf{f}_{1n} : \mathbf{w}_{1n})\})$$
$$\mathbf{v}_2 = (\mathbf{i}_2, \{(\mathbf{f}_{21} : \mathbf{w}_{21}, ..., (\mathbf{f}_{2n} : \mathbf{w}_{2n})\})$$

a 1-1 mapping from $\iota^*(\mathbf{v}_1)$ to $\iota^*(\mathbf{v}_2)$, μ, is a match function if the following conditions hold true simultaneously:

1) $\mu(\mathbf{i}_1) = \mathbf{i}_2$
2) $\forall\, \mathbf{i}\ (\mathbf{i} \in \iota^*(\mathbf{v}_1)),\ \forall\, \mathrm{t}\ (\mathbf{i} \in^d \mathrm{t}),\ \mu(\mathbf{i}) \in^d \mathrm{t}$
3) $\forall\, \mathbf{i}\ (\mathbf{i} \in \iota^*(\mathbf{v}_1))$, if $\mathbf{i}.\mathbf{f}$ is defined, then $\mu(\mathbf{i}).\mathbf{f}$ is defined and $\mu(\mathbf{i}.\mathbf{f}) = \mu(\mathbf{i}).\mathbf{f}$

□

The first condition requires that a match function preserve the root. The second condition says that a match function must also respect the type(s). The third condition requires that a match function preserve the object structure.

Using a 1-1 function, we limit the scope of matching to those that are "truly structurally compatible" with the matching template. For example, in Figure 6.1, h is a match function, but f and g are not. The 1-1 condition can be overly restrictive, as is evident from the case for f: Without the 1-1 condition, f would be a match function in which both a and b are matched with c.

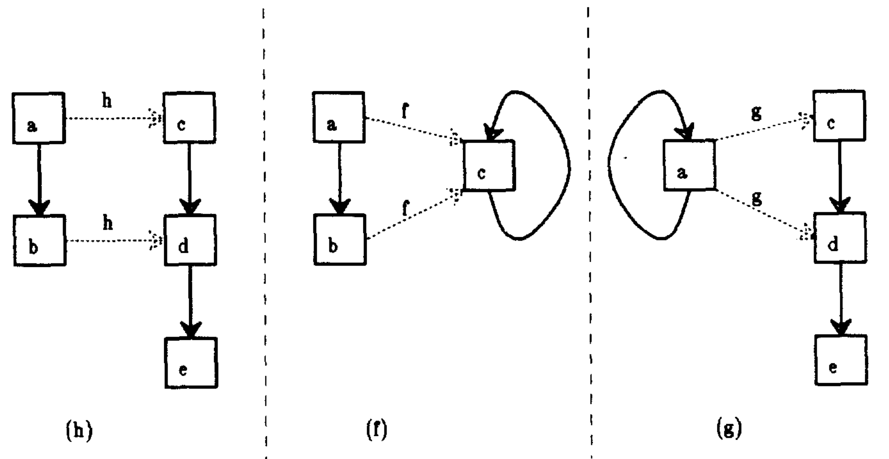

Figure 6.1: Match Functions and Non-match Functions

We say that an object o_1 matches an object o_2, if there is a match function from o_1 to o_2. In the case with only single occurrence fields, it can be shown that if o_1 matches o_2, then there is a unique match function from o_1 to o_2.

We consider computing match functions for objects with only single occurrence fields. The extension to including multiple occurrence field is not difficult but is not discussed. Let $o_1 = (id_1, S_1)$ and $o_2 = (id_2, S_2)$, where S_i is of the form of $\{..., (f_j: o_j), ...\}$, if o_1 matches o_2, then the match function μ can be computed as follows.

The computation starts from the root of o_1:

Case 1). When $S_1 = \emptyset$, the assertion is obviously true with $\mu(id_1) = id_2$.

Case 2). Assume μ is defined for mapping nodes of o_1 with depth n or less to nodes of o_2. Consider in o_1 a field label f leading to a node of depth n+1, a_1, from a node of depth n, a_0, and $\mu(a_0) = b_0$, we can extend h toward a match function and assign b_1 to $\mu(a_1)$, such that $b_0.f = b_1$.

The result above gives us a way to compute match functions: Traverse the structure using breadth-first search, starting from the root. It breaks up matching on two large structures into matching on small pieces. Furthermore, the order of matching on small pieces is immaterial.

When there is a match from object o_1 to object o_2, the match function establishes a 1-1 correspondence from o_1 to portions of o_2. The 1-1 correspondence makes matching in the semantic space useful. We are particularly interested in the pairs of a match function where the source is an abstract object and the target is a concrete object, since they precisely correspond to the kind of pattern matching needed for command processing. To that end, the interesting pairs from each match function are extracted to form a restricted version of the match function, called a *decomposition map*, since it has the ability to break up complex structures. Given a match function μ and a pair $(o_1, o_2) \in \mu$, the pair is an ***abstract-abstract match pair*** if both object o_1 and object o_2 are abstract objects; the pair is an ***abstract-concrete match pair*** if object o_1 is an abstract object and o_2 is a concrete object. Thus, a decomposition map is a match function restricted to contain only abstract-concrete pairs. Or equivalently, a decomposition map is a match function with domain restricted to abstract objects and with range restricted to concrete objects.

Suppose that we partition the components of a complex object into two collections, one consisting of abstract objects and the other concrete objects, then Figure 6.2 illustrates a decomposition map.

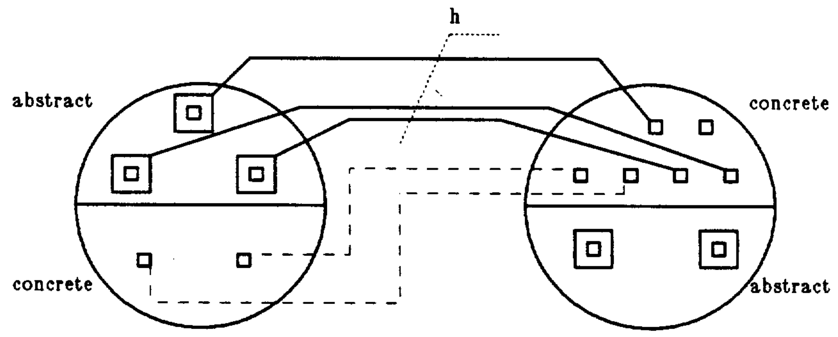

Figure 6.2 A Decomposition Map

In writing, a decomposition map is usually arranged into a row of map pairs, similar to a binding vector in command processing, as in "a_1:c_1, a_2:c_2, ..., a_n:c_n", where a_i's are abstract objects, and c_i's are concrete objects.

Example 41: Consider two objects, described by

1) Point:P?(x $\rightarrow$ Integer:X?, y $\rightarrow$ 10), and
2) Point:(x $\rightarrow$ 20, y $\rightarrow$ 10),

respectively. Suppose the object identifier for the abstract Point object (defined by 1) is a100, the object identifier for the abstract Integer object is a200, and the object identifier for the concrete Point object (defined by 2) is c300, then, the abstract Point object matches the concrete Point object, with the following decomposition map:

a100:c300, a200:20

□

Extending the decomposition maps to include abstract-abstract match pairs results in *generalized decomposition maps.* In other words, generalized decomposition maps remove the concrete-object-only restriction on the ranges of the decomposition maps. Note that the abstract-object-only restriction on the domains still stands. Figure 6.3 illustrates a generalized decomposition map.

The motivation for the extension to generalized decomposition maps is twofold. Firstly, with abstract-abstract matching, abstract objects can be retrieved as a result of pattern matching, the same mechanism that is used to retrieve concrete objects. They can also be manipulated using database commands. Secondly, abstract-abstract match pairs complement abstract-concrete match pairs, in that they together form a unified view for pattern matching: a process in which a template decomposes complex structure, resulting in the binding of the flexible components of the template (abstract objects) to the components of the complex structure.

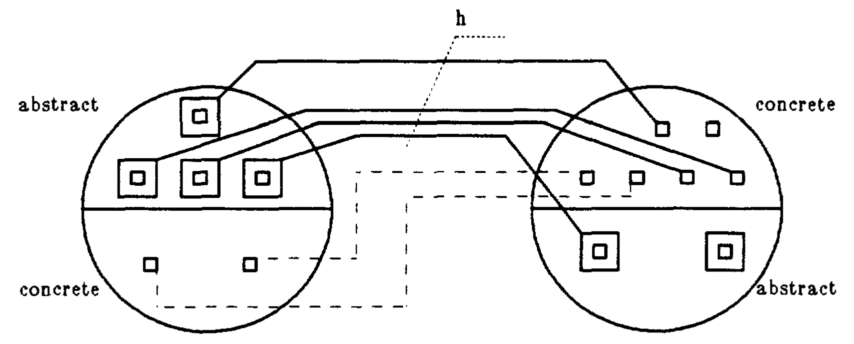

Figure 6.3 A Generalized Decomposition Map

Consider the following update command for setting the x-coordinate value of the Point objects whose y-coordinate value is 10,

P(x $\rightarrow$ 0) $\Longleftarrow$ points $\rightarrow$ Point:P(y $\rightarrow$ 10)

If the match is not restricted to concrete object only, then the abstract object described by the following term is also updated.

Point:P?(x $\rightarrow$ Integer:X?, y $\rightarrow$ 10)

Considering that concrete objects are perhaps manipulated more often than abstract objects, we give the user or application the control over the kind of objects to match against in pattern matching with a slight extension to the syntax for patterns. (Corresponding CDOs also need adjustment to reflect this change in syntax, and to be able to capture the difference in matching semantics.)

To indicate abstract-concrete matching (a decomposition map), we use the same notation that we have been using. For example, the command

$P(x \rightarrow 0) \Longleftarrow$ points $\rightarrow$ Point:P($y \rightarrow 10$)

says setting the x-coordinate value of concrete Point objects only, as the pattern indicates an abstract-concrete matching.

To indicate a generalized decomposition map, we use a question mark following a variable of a pattern. For example, the command

$P(x \rightarrow 0) \Longleftarrow$ points $\rightarrow$ Point:P?($y \rightarrow 10$)

would update the x-coordinate value of both abstract and concrete Point objects that match the pattern.

Finally, to indicate an abstract-abstract match (the range of the match function restricted to abstract objects), we use two consecutive question marks following a variable of a pattern. For example, the command

$P(x \rightarrow 0) \Longleftarrow$ points $\rightarrow$ Point:P??($y \rightarrow 10$)

only picks up abstract Point objects for update

6.3. Using Abstract Objects in TDOs

This section discusses one application of abstract objects, the use of them in type-defining objects, or TDOs. TDOs are the database objects for type definitions. Their function is similar to the system catalogue of a relational database system for maintaining information about tables.

We use the type "TypeDef" to represent TDOs. The basic considerations in the design of "TypeDef" are that it is general, so that it can capture the essential information about types, and that it is efficient, so that it can perform frequently-needed operations on types quickly and easily. The type definition for "TypeDef" is given below:

Typedef = (typename → String,
supertypes →→ Typedef,
template → All?)

where we use the notation "aTypeName" followed by a question mark to indicate an admissible value set consisting of only abstract objects of the given type "aTypeName".

We discuss how this schema meets the two requirements above. The essential information about a type is its name, its supertypes and its structure. It is clear that the name and the supertypes are captured by "TypeDef". Where do we maintain the information about the structure? The answer is: in the "template" field. When defining a type, an abstract object of the new type is created and is assigned to the "template" field as its value. We call this object a *template object*, or TO. The TO contains all the structure defined by its type, with values set to the TOs of the corresponding field specifications. For example, consider the type definition

RectSelect = (rect → Rectangle, cursor → Point)

Its TDO can be described as:

RectSelect:R?(rect → TO_for_Rectangle, cursor → TO_for_Point)

Thus, the first requirement is met. It turns out the second requirement is also satisfied. Two of the most frequent operations are conformity checking and subtype checking, which can be easily formulated as pattern-matching problems with the TOs. Detailed discussion along this line is given in Chapter 8.

The constraining type for the "template" field is "All", the reason is that user-defined types can be arbitrarily close to the top of a type hierarchy (namely, the type "All").

We describe our mapping from several type definition forms to their corresponding TDOs. First, consider the general type definition with a list of fields.

$$T = (f_1 \rightarrow T_1, \ldots, f_n \rightarrow T_n)$$

Its TDO is is illustrated in Figure 6.4.

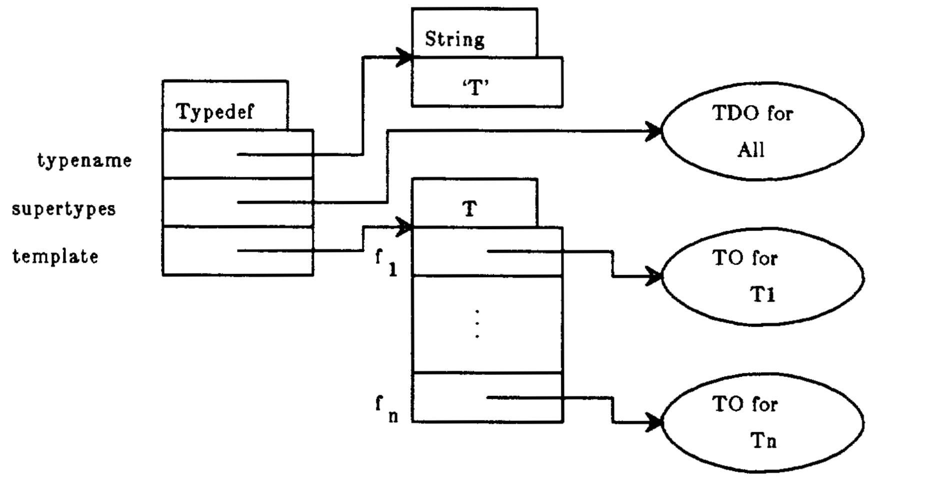

Figure 6.4 TDO for Type T's Definition

If the type S is defined as

$$S = T{:}(g_1 \rightarrow S_1, \ldots, g_m \rightarrow S_m)$$

then, its TDO is illustrated in Figure 6.5. Note that the inherited fields are explicitly duplicated.

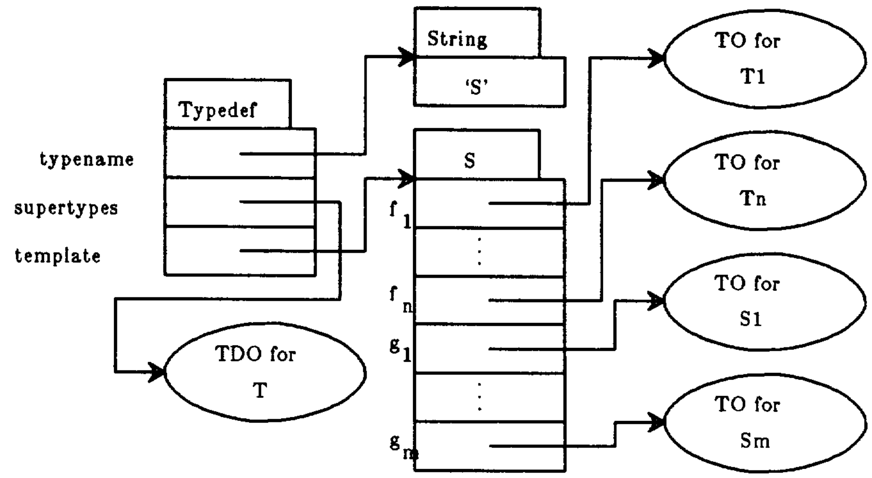

Figure 6.5 TDO for Type S's Definition

Given a type definition

U = S,

its TDO is shown in Figure 6.6. (Note that there is no linkage to the type S.)

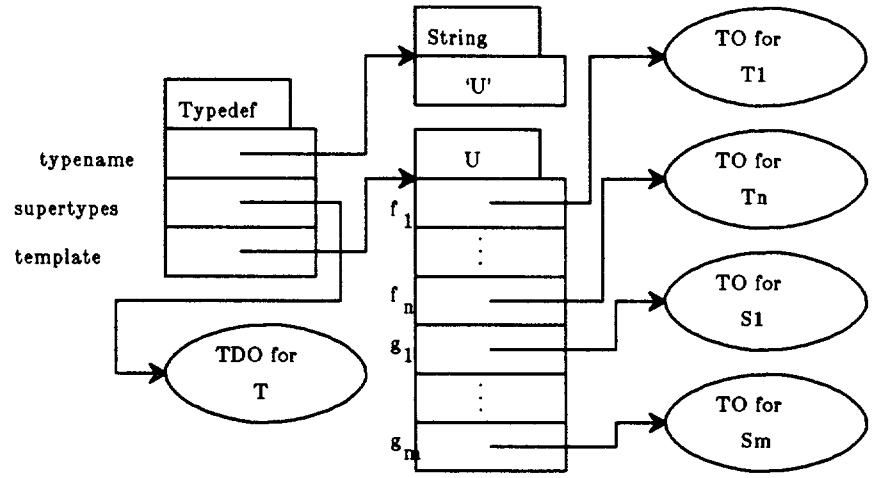

Figure 6.6 TDO for Type U's Definition

In contrast to the last case, a type, W, defined using the form

$$W < S$$

is mapped to the TDO shown in Figure 6.7.

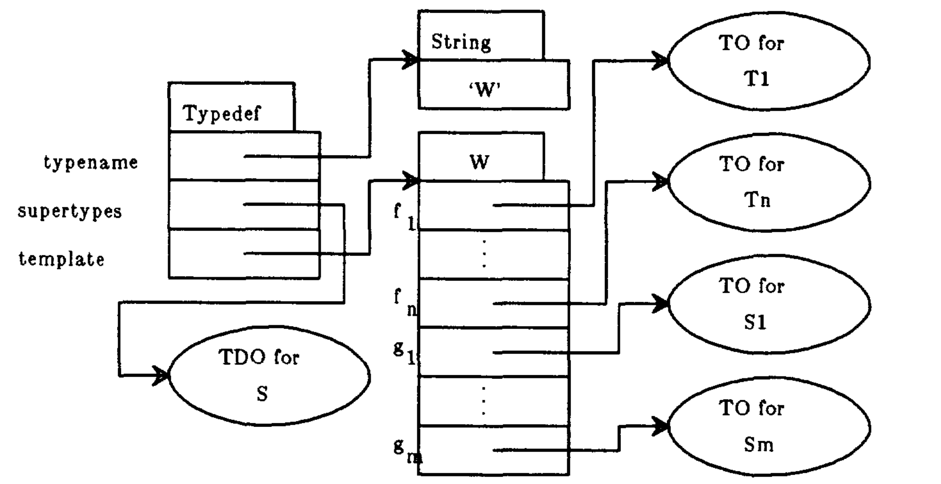

Figure 6.7 TDO for Type W's Definition

6.4. Using Abstract Objects in CDOs

Another interesting the use of abstract objects is representing command-defining objects, or CDOs. We use the type "Command" to store CDOs. The definition of "Command" is as follows.

Command = (head → Head, @collection →→ All?)
Head = (@operation →→ All?)

The notation "@aFieldName" is used to denote an abstract field, a field whose name is a variable. The field "@collection" reflects the fact that pattern matching in command processing always has a scope of a collection of objects. The field "@operation" can be substituted by one of the seven concrete operations: addType, delType, addField, delField, changeField, newObject and compoundCommand. ("View" and "Install" are treated as predefined compound commands.)

The "head" field is defined here as a single occurrence field. Nevertheless, multiple heads are allowed in a command. A better scheme is to define "head" as an ordered multiple occurrence field. That treatment is discussed in Chapter 7.

Consider again the command for setting the x-coordinate value of a Point object whose y-coordinate value is 10:

$$\text{P}(\text{x} \rightarrow 0) \Leftarrow \text{points} \rightarrow \text{Point:P}(\text{y} \rightarrow 10)$$

Its CDO is the following (Recall that object tags are non-repeating):

$$\begin{aligned}\text{Command:C(head} \rightarrow \text{Head:H(}&\text{changeField@head} \rightarrow \text{P?(x} \rightarrow 0\text{)},\\ &\text{points@collection} \rightarrow \text{Point:P?(y} \rightarrow 10\text{))}\end{aligned}$$

6.5. Chapter Summary

In this chapter, the concept of pattern-matching is studied. The semantics of pattern-matching with abstract objects are described. We also discussed two interesting applications of abstract objects in the TEDM data model itself: in type-defining objects and in command-defining objects.

CHAPTER 7

GROUPING COMMANDS INTO COMPOUND OPERATIONS

Database commands can be grouped together to form *compound commands.* The commands in a compound command are executed sequentially, with the condition that either all of them are executed or none of them are executed. In other words, a compound command is a unit of atomicity. Hence, it is most sensible to group semantically related operations as compound commands. The atomicity condition guarantees that the semantics are enforced consistently. For example, removing a field followed by removing a type from an object can be a good combination for maintaining the conformity condition.

Compound commands are also a control abstraction mechanism, like the procedural abstraction mechanism in programming languages. They provide high-level semantics based on low-level operations. They resemble parameterized functions or subroutines, only need to be defined once, and can be used many times from different contexts and with different arguments.

This chapter describes mechanisms for grouping commands into compound commands in TEDM. The discussion centers more on the implementational aspects than on semantical issues. Particular attention is paid to the problem of passing parameters in and out of compound commands. Section 1 defines the syntax for compound commands and describe how they are executed. The generic syntax form is a slightly modified version of the one given in Chapter 5. The variation reflects some of the implementation considerations. Section 2 describes the issues in representing commands as objects. Section 3 describes a parameter-passing scheme suitable for the pattern-matching style of command execution in TEDM. Section 4 discusses the creation and the manipulation of the parameter objects. Section 5 uses an example to illustrate how to execute a compound command from its database representation. Section 6 contains remarks on extension and a chapter summary.

7.1. Syntax and Semantics of Compound Commands

We gave a generic form for defining compound commands in Chapter 5. The form below is a slightly modified version. The modification is needed to incorporate some implementation strategies. For example, argument terms are introduced to implement parameter-passing. (The parser of the prototype recognizes this modified version.) The formal syntax of compound commands is described in the BNF-style as follows.

```
<CompoundCommand> ::= <CommandName> '[' <FormalArgument> ']'
                      '{' <CommandList> '}'
<FormalArgument> ::= [ <InputArgumentList> ] [ ';' <OutputArgumentList> ]
<InputArgumentList> ::= [ <ArgumentList> ]
<OutputArgumentList> ::= [ <ArgumentList> ]
<ArgumentList> ::= <Argument> { ',' <Argument> }
<Argument> ::= <ArgumentName> '→' <TypeName> ':' <VariableName>
<CommandList> ::= <Command> { <Command> }
<Command> ::= <HeadList> '<==' [ <Pattern> ] [ ',' <ArgumentTerm> ]
<HeadList ::= <Head> { ',' <Head> }
```

According to this definition, a generic form for writing compound commands is as follows:

aCommandName[anArg$_1$ → T$_1$:V$_1$, ..., anArg$_n$ → T$_n$:V$_n$]
{
anAction$_{11}$, ..., anAction$_{1m}$ <== aPattern$_1$, anArgumentTerm$_1$;
...
anAction$_{k1}$, ..., anAction$_{km}$ <== aPattern$_k$, anArgumentTerm$_k$; }

Note that there are two new concepts in the commands of this generic form: multiple actions to the left-hand side of "<==", and an additional argument term "anArgumentTerm" to the right-hand side of "<==". The multiple actions in a command head form a *compound head*. The pattern and the argument term form a *parameterized pattern*.

The example below is based on the following type definitions:

```
StateTrivia = (bird → String, flower → String, tree → String,
               song → String, motto → String)
GeoState    = (stateName → String, area → Integer, population → Integer,
               postalAbbr → Integer, fipsCode → Integer,
               capital → String, trivia → StateTrivia)
```

Example 42: This command displays GeoState objects, with names given by the calling environment. Note that the term "ViewStates(nm → N)" is an argument term.

```
ViewStates[nm → String:N]
{
  View[S] <== states → GeoState:S(nm → N), ViewStates(nm → N);
}
```

□

We discuss the semantics of compound commands informally. Individual commands are executed one at a time in a sequential order. However, to maintain database integrity, we require that either all commands are executed or none of them are executed. A simple command is executed in essentially the same way as before: pattern matching followed by the execution of actions. Nevertheless, new concepts are introduced to handle parameter-passing from the calling environment into individual commands and from one action into another. The execution of a compound head is coordinated using a simple strategy: the actions are executed one at a time sequentially from left to right. The outcome of a pattern matching is still a binding matrix, which supplies handles to objects. The bindings from the pattern matching phase are directly used during the execution of the first action. Each action may extend the binding matrix with its output parameters. The binding matrix is passed to the second action for its execution, which in turn may modify the binding matrix and pass it to the next one, and so on. The binding matrix from the last action of a compound head is in the final form of executing a single command, which also reflects the update to the database implicitly.

Initially, a set of objects are passed from the outside into a compound command through parameter-passing. Accessing incoming parameters inside the compound command is accomplished by *argument terms,* which, together with pattern terms, form a parameterized pattern. An argument term consists of a compound command name and a list of arguments, similar to the header portion (that is, the signature portion) of a compound command definition. The purpose of arguments terms is to match the temporary workspace objects for the argument relation, hence the actual arguments are effectively used by the compound command.

Like a procedure, a compound command is invoked by name and a list of actual arguments. Upon entering a compound command, actual parameters are bound to formal parameters. A compound command can be used as an action in the command head, which is also a simple and convenient way for obtaining actual arguments for the command, via pattern matching, as is shown below.

$$\text{aCommandName}[\text{anArg}_1 \rightarrow T_1{:}V_1, \ldots, \text{anArg}_n \rightarrow T_n{:}V_n] \Longleftarrow \text{aPattern.}$$

A compound command expects, for its execution, a set of bindings for each parameter, V_1, ... V_n. Note that it is incorrect to pass the binding set for each individual argument separately. Consider a command that needs two arguments for execution, as in "TwoArgumentCommand[a $\rightarrow$ A, b $\rightarrow$ B]", and the binding matrix obtained in pattern matching is

[A:1, B:1]
[A:1, B:2]
[A:2, B:2]

If the parameters are passed as two separate sets, {1, 2} and {1, 2}, for A and B respectively, it is impossible to reconstruct the binding matrix, and leading to incorrect execution. A correct but inefficient way to pass a set of bindings is to pass them one at a time. In this approach, the command above would need three iterations before it gets the complete information from the binding matrix.

We propose a set-oriented parameter-passing mechanism that is correct, efficient as well as elegant. The basic idea of the method is the following. Create a temporary object for each binding vector, and define argument terms in the command bodies in such a way that they match these temporary objects; then, when individual commands are executed, the pattern-matching process will pick up the temporary object into the binding matrix for the head of the command. This approach is correct since the temporary objects preserve the binding matrix structure. It is efficient because the whole binding matrix is passed all at once. It is elegant because it is based on pattern-matching, a concept used everywhere in the data model.

7.2. Representing Compound Commands

Compound command definitions are stored as objects of the type "Compound". Each "Compound" object maintains the information about the name, the arguments and a list of commands. The type definition of "Compound" is the following: (The notation "[]→" indicates an ordered multiple occurrence field.)

```
Compound = (name → String, @argument →→ All?, command  []→ Command)
Command = (@operation []→ All? @collection →→ All?)
```

We have slightly modified the definition for type "Command" from the last chapter, to accommodate the notion of compound heads. We also used the notation for abstract field in the definition.

Example 43: The compound command "ViewStates" can be represented as follows.

```
Compound:(name → 'ViewStates',
          nm@argument → String:N?,
          command → Command:(View@operation → S?,
                                      states@collection → GeoState:S?
                                                          (nm → N?),
                                      viewStates@collection → N?
```

□

We consider a more complex example. This example is based the following type definitions:

```
Point = (x → Integer, y → Integer)
Rectangle = (width → Integer, height → Integer)
LayoutUnit = (rect → Rectangle, color → String)
Layout = (unit → LayoutUnit, position → Point)
```

These types define a simple representation for a VLSI layout application, where a point on the layout plane is determined by its x-coordinate and y-coordinate, both of type integers. A rectangle is geometric shape defined by its width and height, assuming orientation is always parallel to the x and y axes. A layout unit is a rectangle filled with color, which indicates the layer type of the unit, polysilicon or diffusion, for example. Finally, a layout object is simply such a layout unit with its position fixed, by associating a point in the layout plane with the upper-left corner of the rectangle.

Example 44: NMOS design uses the following color code to indicate different layers: BLUE for metal, RED for polysilicon, GREEN for diffusion, BLACK for contact cut and YELLOW for implant. A unit used to measure distance in design is called λ, whose absolute values range from 3 microns down to 1 micron. There is a set of design rules governing a layout design, which dictates the minimum size of (separation between) layout objects to ensure certain electrical and electronic behavior of the resulting VLSI chips. For example, a polysilicon layout objects must be larger than 2λ, two polysilicon layout objects must be separated by at least 2λ and a polysilicon and a diffusion by at least 1λ, etc.

The following syntax defines a compound command that takes as its input parameter a collection of Layout objects, shifts all diffusion layout object, with a vertical position of 10, from the input collection, by 10λ, and changes all layout objects with a vertical position of 5, again from the input collection, to polysilicon objects.

ShiftDiff[layout $\rightarrow$ Layout:LO]

```
{
 P(y → 0) <== layoutObjects → Layout:LO
                    (position → Point:P(y → 10),
                     unit → LayoutUnit:U(color → 'GREEN')),
                     ShiftDiff(layout → LO);

 U(color → 'RED') <== layoutObjects → Layout:LO
                    (position → Point:P(y → 5),
                     unit → LayoutUnit:U),
                     ShiftDiff(layout → LO);
}
```

One way to invoke this compound command is shown below, which selects from all layout objects those with a width of 2 λ, and apply the "ShiftDiff" command to the results of the selection.

```
ShiftDiff[layout → X] <== layoutObjects → LayoutObject:X
                    (unit → LayoutUnit:(rect → Rectangle:(width → 2)))
```

□

Example 45: The object translation of the compound command "ShiftDiff" is the following.

```
Compound:(name → 'ShiftDiff',
   layout@argument → Layout:LO?,
   command [1]→ Command:(changeField@operation → P?(y → 0),
       layout@collection → Layout:LO?
           position → Point:P?(y → 10),
           unit → LayoutUnit:U?(color → 'GREEN'),
       Shift@collection → LO?)
   command [2]→ Command: ...
)
```

□

7.3. Argument Terms and Parameter Passing

Argument terms match temporary parameter objects, which are created by the system based on the results of pattern-matching connected with the call of the command. One temporary object is created for each row in the binding matrix. In its simplest form, an argument term inside a compound command is merely a repetition of the header of the command, which enables an individual command to access parameter objects by pattern matching. The examples we have seen so far are of this kind (which is the only type the current implementation accepts). However, more complex argument terms can be introduced, for example, to provide simple transformations on parameter objects before they are used.

There are four issues to consider for the parameter-passing problem. The first one is how to represent compound command headers, that is, command names and formal arguments. The second one is how to translate and store argument terms. The third is the inverse of the second, namely, how to generate patterns out of stored argument terms. Finally, the fourth issue is how to generate parameter objects, in such a format that they will match argument terms. We discuss the first two problems in this section, and leave the last two to the next section.

The header of a command definition serves the purpose of associating a name with the definition. On invocation, the name is used as the unique identification for locating the definition of the compound command. In addition, the header also contains a template for formal arguments, which is used to establish a correspondence between the formal arguments and the actual arguments, or parameter-passing. We used the type "Signature" to represent the header portion in command definitions.

Signature = (name $\rightarrow$ String, @argument $\rightarrow$ All?)

Using "Signature", the type "Compound" is rewritten as

Compound = Signature:(command []$\rightarrow$ Command)

As mentioned above, the mechanism that makes arguments passed to a compound command accessible to individual commands is *argument terms*, a special kind of pattern appearing in the body of a command. An argument term may selectively use the parameters of a compound command, by including all or some of the argument names. In syntax, an argument term is similar to a command header in a compound command definition. It is formed with a command name followed by a list of arguments. Furthermore, the command name must be identical with the one appearing in command header, and the arguments must be a subset of those in the command header. Also, the field names in the argument term must also be a subset of the parameter names.

7.4. Parameter Objects

When a compound command is invoked, its definition is retrieved. The object representation of the definition is assembled into an executable form and executed. Part of this conversion process is to identify argument terms in the definition and convert them into a form that can effectively use parameters from the calling environment. This conversion depends on the format in which parameter objects are created. Another important piece of processing during command invocation is augmenting the code with instructions for generating parameter objects dynamically.

Parameters for compound commands are sets of bindings (binding matrix). Each row in this matrix corresponds to a parameter object. Parameter objects are created in a temporary workspace.

Example 46: Given a binding matrix with four binding vectors

[[LO:o1, P:11], [LO:o2, P:12], [LO:o3, P:13], [LO:o4, P:14]]

and a command with one argument "ShiftDiff[layout → Layout:LO]", then, four parameter objects are created:

```
Parameter:(layout → o1)
Parameter:(layout → o2)
Parameter:(layout → o3)
Parameter:(layout → o4)
```

Moreover, the pattern "Parameter:P(layout → LO)" would match all of them.

□

The type definition for "Parameter" is as follows:

```
Parameter = (@argname →→ All?)
```

7.5. Compound Command Execution

We use an example to illustrate the the process of compound command execution. Consider the following command definition of an earlier example:

```
ShiftDiff[layout → Layout:LO]
{
 P(y → 0) <== layoutObjects → Layout:LO
                    (position → Point:P(y → 10),
                     unit → LayoutUnit:U(color → 'GREEN')),
                     ShiftDiff(layout → LO);

 U(color → 'RED') <== layoutObjects → Layout:LO
                    (position → Point:P(y → 5),
                     unit → LayoutUnit:U),
                     ShiftDiff(layout → LO);
}
```

Suppose we invoke the command as follows:

ShiftDiff[layout → L] <== Layout:L

and assume the database contain the following objects (assuming they are labeled by object identifiers):

```
o1) Layout:(unit → LayoutUnit:
          (rect → Rectangle:(width → 2, height → 13), color → 'GREEN'),
          position → Point:(x → 2, y → 1))
o2) Layout:(unit → LayoutUnit:
          (rect → Rectangle:(width → 4, height → 2), color → 'GREEN'),
          position → Point:(x → 1, y → 2))
o3) Layout:(unit → LayoutUnit:
          (rect → Rectangle:(width → 4, height → 2), color → 'GREEN'),
          position → Point:(x → 1, y → 6))
o4) Layout:(unit → LayoutUnit:
          (rect → Rectangle:(width → 4, height → 2), color → 'GREEN'),
          position → Point:(x → 1, y → 10))
o5) Layout:(unit → LayoutUnit:
          (rect → Rectangle:(width → 10, height → 2), color → 'BLACK'),
          position → Point:(x → 0, y → 13))
```

Step 1, pattern matching on "Layout:L" produces the following binding matrix, as the pattern matches all layout objects:

[L:o1], [L:o2], [L:o3], [L:o4], [L:o5]

Step 2, the command head is recognized as an invocation to compound command "ShiftDiff". The corresponding CDO is retrieved and reassembled for execution. At the same time, the following parameter objects are created before the control is transferred to the body of the compound command:

```
Parameter:(layout → o1)
Parameter:(layout → o2)
Parameter:(layout → o3)
Parameter:(layout → o4)
Parameter:(layout → o5)
```

Step 3, the first simple command is executed. The binding matrix of its pattern matching is

[LO:o4, P:p], where p is an identifier for a Point object

Therefore, the action part of this command changes the y-coordinate value of the point object in the binding matrix to 0.

The last step executes the second simple command, which updates the layout unit situated at points whose y-coordinate value is 5, and make them a "RED" unit.

7.6. Remarks and Chapter Summary

Our discussion in this chapter reflects the implementation of the prototype. This closing section suggests some future directions: restrictions on how the binding matrices may change during execution to simplify runtime storage management and to improve the performance and some extensions to the parameter-passing technique.

We do not allow a command to update the value of its binding matrix, nor do we permit the number of bindings in a binding matrix to grow. However, we allow a command to reject bindings, decreasing the number of bindings in its binding matrix. The reason for that is very simple. Conceptually, bindings are always the results of pattern matching. Although command execution can potentially change the outcome of the pattern matching prior to its execution, this change will not be seen until another pattern matching outside the current command is performed. The decision to allow rejection of bindings is based on the consideration that, unlike introducing new bindings, shrinking in a binding matrix would not involve allocation of storage space.

One consequence of these restrictions is that, using this strategy for set-oriented parameter passing, the storage space requirement will never grow as new invocations are made. Therefore, it is possible to use a stack based procedure invocation mechanism for a more realistic implementation in the future.

We consider a number of extensions related to compound command invocation and parameter passing. First, execution of a command may augment its binding with new columns, namely, the binding matrix may grow horizontally. Within such a command, such an intention is indicated by assigning object values to corresponding formal arguments, perhaps with the assistance of a new kind of header term, *assignment terms.*

If the storage management function of the system strictly enforces the *non-increasing* policy, we may invent commands that have ability to preallocate space for intended binding matrix augmentation.

To summarize, this chapter discussed ideas for grouping database commands to form a high-level control abstraction. The notion of atomicity was important to guarantee the database integrity. The discussion was mostly based on the features and techniques of the current prototype, except the content of this last section. The concept of argument terms was instrumental in solving the set-oriented parameter-passing problem for command invocation.

CHAPTER 8

PROTOTYPING

We have implemented a prototype of the TEDM model described in this thesis. The prototype is memory-based: It creates and manipulates objects only in (virtual) memory. The prototype is written using a combination of three different programming languages. Quintus Prolog is used to implement a *model manager* (MM), which handles the majority of the functionalities of the prototype, ranging from language translation to query answering. Tektronix Smalltalk is used to build a small user interaction facility (UIF), which adds to the prototype a simple window-based user interface. The MM module and the UIF module are each a concurrent process. The communications between the two processes are facilitated by a set of independent C++ library tasks (CLIB).

We discuss the prototype in this chapter, how it is organized and how it is implemented. The emphasis of the discussion is on the MM module. We pay particular attention to how complex objects are mapped into simple forms (decomposition), how they are stored (storage model), and how type-defining objects and command-defining objects are represented. Also of interest is the extensibility of the MM with respect to languages. Several language interfaces are implemented in the prototype. Others can be easily incorporated if needed. Addition of a new language follows a straightforward procedure and is done in a modular fashion.

The organization of this chapter is as follows. Section 1 contains an overview of the overall architecture and a brief description for each of the three major functional blocks. Section 2 considers representation issues, such as how to partition object space and how to decompose complex objects. Section 3 describes a translator for objects from their definition to their representation. Section 4 and Section 5 discuss similar translators for types and commands, respectively. Command execution is also described there. Section 5 examines the prototype's adaptability to interface languages, one of many advantages resulting from an expressive object language and generalized object spaces.

8.1. Architecture

The three functional blocks are connected (conceptually) using a strict linear hierarchy, as shown in Figure 8.1.

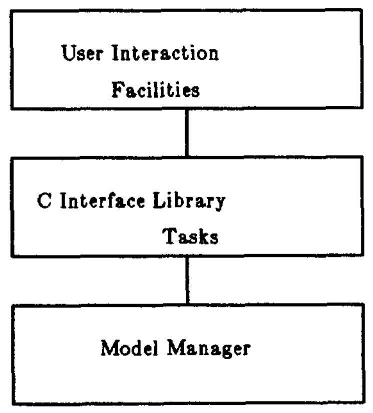

Figure 8.1 System Organization

Notice that each block in the diagram is a separate runtime task. The parent/child relationship among the concurrent tasks coincides with the hierarchical structure of the block diagram. The process relationship is set up during initialization. A CLIB task is forked from Smalltalk during the creation of a UIF object. The CLIB task in turn starts a Prolog servant as its child process. There are two pipes between the Smalltalk process and a CLIB task: a write-pipe for sending data to the CLIB process and a read-pipe for receiving data in the opposite direction. The communications between a CLIB task and its Prolog servant use the external language library that comes with the Quintus Prolog system.

The UIF Process

The UIF is constructed using the MVC paradigm of Smalltalk. The user starts a TEDM session by opening a UIF window, enters and edits definitions for objects, types or commands in the UIF window, selects expressions and invokes appropriate menu actions.

In addition to common editing features such as "cut" and "paste", the interface supports the following functions:

1). *Translate Object*: This function sends an object term to the model manager, via the CLIB task. The model manager translates the term and checks if the object described by the term is valid, according to type conformity requirements.

2). *Translate Type*: This function sends a type definition to the model manager. The model manager translates the definition into a "Typedef" object, and stores the result in the database.

3). *Translate Command*: This function sends a command procedure definition to the model manager. The model manager translates the procedure definition into a "Command" or "CompoundCommand" object, and stores the result in the database.

4). *Execute Command*: This function sends an interactive command to the model manager. The model manager compiles the interactive command and executes it. The result of execution is sent back to the Smalltalk process displayed in a popup window.

The CLIB Tasks

A group of C++ programs make up the CLIB. Their primary function is to create Prolog servant processes. They also handle the communications between the UIF and the MM. Although the amount of data from the UIF to the MM is usually small, data transfer in the opposite direction may require large bandwidth. For example, the result of a viewing operation may contain all objects in the database. Consequently, a CLIB task uses pre-determined files in which a servant process deposits execution results. (It turns out the external language interface of Quintus Prolog only supports low-level and low-bandwidth communications.) The successful completion of a servant process is communicated to a CLIB task via a return Boolean value.

The Servant Processes

Each servant process is an incarnation of the model manager. Database objects are represented as Prolog facts and stored on a heap. Commands are translated into and executed as Prolog goals. Several features of Prolog have proven valuable in the implementation. For example, the definite clause grammar (DCG) formalism facilitates parser construction, and the unification mechanism fits well with the pattern-matching style of command processing.

The MM is made up of three translators and a command processor. Each translator converts external syntax into internal representation (that is, Prolog facts and rules). The command processor compiles commands into Prolog queries, executes them using the Prolog interpreter, and maintains the result in a *binding matrix*. The block labeled by "Object Memory" is simulated using part of Prolog's heap space. A separate access interface is defined to hide the implementation details of the object memory from how it is used by the MM. This way, the amount of change is minimized when we switch to a disk-based object store. Figure 8.2 shows the organization of the model manager.

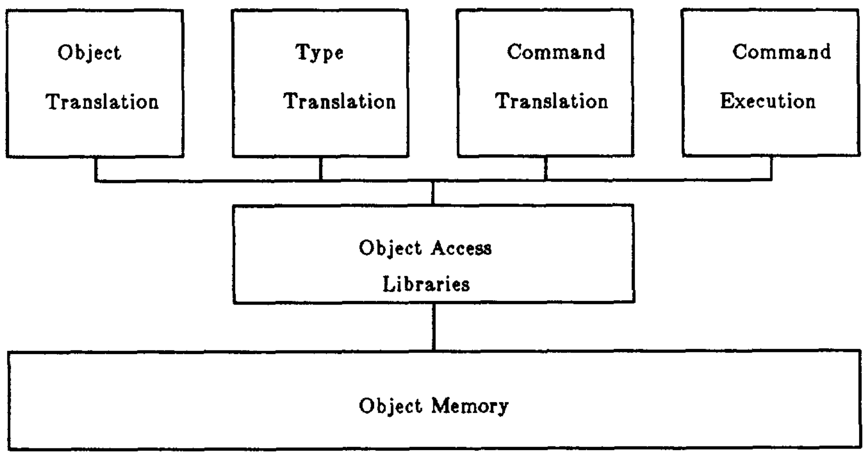

Figure 8.2 Model Manager Organization

8.2. Representing Objects

Two issues in object representation are the management of object identifiers and the mapping of complex structure into the storage model.

Managing Object Identifiers

The most fundamental requirement on object identifiers, that they must be unique, can be met using a natural number generator. In addition, we partition the space of object identifiers into disjoint subspaces, and use object identifiers from different subspaces to represent different categories of objects (concrete, abstract, etc.) Also, only a portion of the natural numbers should be used as object identifiers, for we still need to support the primitive type "Number". Similarly, to support the primitive type "String", another chunk of the natural numbers is reserved as indexes into the table for string values (*stringTable*). Finally, based on the consideration of uniformity in the storage model, a table (*symbolTable*) is used to maintain symbols.

Storing Objects in Tuples

The storage space for database objects is a quadruary relation (the *quadruary storage model* or QSM). Complex objects on a field occurrence basis are fully decomposed into tuples with a uniform storage structure. Advantages and disadvantages of the fully decomposed storage model versus the direct storage model have been studied in detail by Copeland and Khoshafian [Copeland85]. The applicability of each approach is discussed in an earlier working paper [Zhu85]. The quadruary relation is essentially a fully decomposed model, with minor variations and extensions.

The four columns in the (QSM) are *OBJECT_ID, FIELD_LABEL, FIELD_INDEX* and *FIELD_VALUE.* The following rules are observed by the QSM:

1). OBJECT_ID and FIELD_VALUE may contain object identifiers, integer values and string indices.

2). OBJECT_ID and FIELD_VALUE, FIELD_LABEL and FIELD_INDEX have pair-wise disjoint domains.

Values in FIELD_INDEX are encoded as follows. A negative one (-1) denotes a single occurrence field. A zero (0) denotes a multiple occurrence field. A positive integer denotes an indexed field. In this case, the value in the FIELD_VALUE column is the field value at the index position.

Decomposing Objects

In order to store complex objects using the QSM, they must be decomposed into quadruary tuples. We illustrate the idea of decomposition using a simple example. Given an object of the following form

State:(stateName → 'Oregon', postalCode → 'OR'),

the decomposition process would generate the following QSM tuples.

aStateID, anIndexToSymbolType, 0, theStateTypeID
aStateID, anIndexToSymbolstateName, -1, anIndexToStringOregon
aStateID, anIndexTopostalCode, -1, anIndexToStringOR

The first tuple represents the type membership ("anIndexToSymbolType") of the object ("aStateID"). In the example, the object is a member of the typeset for type "State" ("theStateTypeID"). Since objects can be multiply typed, the index field contains a 0. The remaining two tuples represent the two fields in the object. Note that field labels are represented by indexes to the symbol table, and string values are represented by indexes to the string table. The general decomposition procedure is the following.

1). For an atomic integer value, no decomposition is possible. The identity of the integer value is itself.

2). For an atomic string value, no decomposition is possible. Insert the string into the string table and use the index to the string table as the identity of the string value.

3). For a complex object, $T:(f_1 \rightarrow v_1, ..., f_n \rightarrow v_n)$. Obtain a new object identity, say, id, for the object, and its decomposed form is

$$\{(id, f_1, fi_1, id_1), ..., (id, f_n, fi_n, id_n), (id, type, 0, id_T), ...\} \cup D_1 \cup ... \cup D_n$$

where fi_i is the field occurrence indicator for f_i, id_i is the identity for v_i, and D_i is the result of decomposing v_i.

8.3. Translating Objects

The object translator (OT) converts external object descriptions (in the object definition language) into internal representation (in QSM). The structure of OT follows that of standard language processors, consisting of a scanner, a parser, a semantic checker and a QSM representation generator.

The scanner is a low level utility that is shared by other translators in the MM. It handles text input and token recognition. The output of the scanner is a list of token type and token value pairs. Syntactical analysis is done using a definite clause grammar parser. The parser generates abstract syntax trees (in the form of Prolog terms).

The semantic checker traverses the abstract syntax trees, produces an intermediate form with temporary object identifiers, and checks the satisfaction of the conformity conditions. If the semantic checking is successful, a QSM representation of the object(s) is created and added to the database. An additional activity during generating QSM representation is eliminating redundancy in the intermediate form.

Parsing Objects

Language parsing in Prolog is an easy task. We sketch *definite clause grammars* (DCG) to help describe the translators. A DCG is a set of *definite clauses* (Horn clauses with exactly one positive literal). A definite clause,

$$p \longrightarrow p_1, \ldots, p_n.$$

with the assumption that each literal is of arity 2 (one input and one output), can be interpreted as a production with a nonterminal p as its left-hand-side, and grammar symbol sequence $p_1, \ldots, p_n$ as its right-hand-side. A DCG for a context-free grammar without left recursion is easily obtained by a natural isomorphism. To illustrate, consider the following example.

Example 47: Given a grammar, with two nonterminal symbols: "object_term" and "field_list", and five productions:

```
object_term ::= '(' field_list ')' | string | number
field_list  ::= field_name '→' object_term ',' field_list
              | field_name '→' object_term
```

the corresponding DCG is:

```
object_term ⟶ consume_lparenthesis, field_list, consume_rparenthesis.
object_term ⟶ consume_string.
object_term ⟶ consume_number.
```

```
field_list --> consume_fieldname, consume_arrow, object_term,
    consume_comma, field_list.
field_list --> consume_fieldname, consume_arrow, object_term.
```

where literals of the form "consume_X" are rules for token consumptions (usually supported by lexical analysis routines). □

The resulting DCG is an executable parser. A query of the form "object_term(TokenList, [])", with "TokenList" bound to a sentence represented as a list, would determine whether the sentence is in the language. The DCG formalism is convenient as it requires a minimal amount of programming. The DCG mechanism is general as it is applicable to any context-free grammar without left recursion, which is known to be equivalent to the class of context-free grammars itself. The disadvantage of using DCG for parsing is that certain DCG parsers may be inefficient to execute.

Generating parse trees or abstract syntax trees is just as easy: one simply adds additional arguments to literals. We can also select what to include in the abstract syntax tree. For example, punctuations or delimiters are typically excluded.

Example 48: In this example, we add an argument to the literal "object_term" to obtain a tree representation.

```
object_term(oterm(FL)) --> consume_lparenthesis,
    field_list(FL), consume_rparenthesis.
object_term(oterm(ST)) --> consume_string(ST).
object_term(oterm(NM)) --> consume_number(NM).

field_list(flist(FN, FV, FL)) --> consume_fieldname(FN),
    consume_arrow, object_term(FV), consume_comma, field_list(FL).
field_list(flist(FN, FV)) --> consume_fieldname(FN),
    consume_arrow, object_term(FV).
```

□

Example 49: Given the following object description (with appropriate lexical processing) as input:

$$(\text{name} \rightarrow (\text{first} \rightarrow \text{'Joe'}, \text{last} \rightarrow \text{'Dow'}), \text{age} \rightarrow 2)$$

The parser in the previous example would produce the following abstract syntax tree (as a Prolog term):

```
oterm(flist(name, oterm(flist(first, 'Joe', flist(last, 'Dow'))), flist(age, 2))
```

□

Traversing the resulting tree is also a simple task. The synthesis and the analysis of the abstract tree bear a surprising symmetry in Prolog.

Example 50: The following program analyzes a tree structure, assuming the result of the analysis is saved as global facts and communications are through the global states. It traverses the tree in a depth-first fashion.

```
analyze_object_term(oterm(FL)) :- analyze_field_list(FL),
analyze_object_term(oterm(ST)) :- analyze_string(ST).
analyze_object_term(oterm(NM)) :- analyze_number(NM).

analyze_field_list(flist(FN, FV, FL)) :- analyze_fieldname(FN),
    analyze_object_term(FV), analyze_field_list(FL).
field_list(flist(FN, FV)) :- analyze_fieldname(FN), analyze_object_term(FV).
```

□

Generating Intermediate Forms

The intermediate representation is a direct implementation of the object decomposition algorithm of the previous section. As the trees are traversed depth-first, and new temporary object identifiers are acquired along the way, a stack is used to store the intermediate identifiers.

Example 51: Adding a stack to the traversal routine of the last example, we obtain a program for generating intermediate forms. (Assuming the stack is implemented elsewhere.)

```
analyze_object_term(oterm(FL)) :- acquire_tempid(ID),
    push_stack(ID), analyze_field_list(FL).
analyze_object_term(oterm(ST)) :-
    analyze_string(ST, ST_ID), push_stack(ST_ID).
analyze_object_term(oterm(NM)) :-
    analyze_number(NM, NM_ID), push_stack(NM_ID).

analyze_field_list(flist(FN, FV, FL)) :- analyze_fieldname(FN, FN_ID),
    analyze_object_term(FV), pop_stack(FV_ID), top_stack(OB_ID),
    emit_tuple(OB_ID, FN_ID, FV_ID), analyze_field_list(FL).
field_list(flist(FN, FV)) :- analyze_fieldname(FN, FN_ID),
    analyze_object_term(FV), pop_stack(FV_ID),
    top_stack(OB_ID), emit_tuple(OB_ID, FN_ID, FV_ID).
```

□

Example 52: Consider the following object description again:

(name → (first → 'Joe', last → 'Dow'), age → 2)

Its intermediate representation contains the following tuples.

```
TEMPID_1, name, -1, TEMPID_2,
TEMPID_1, age, -1, TEMPID_3,
TEMPID_2, first, -1, TEMPID_4,
TEMPID_2, last, -1, TEMPID_5,
TEMPID_3, integer, -1, 2,
TEMPID_4, string, -1, 'Joe',
TEMPID_5, string, -1, 'Dow'
```

□

We should point out that this series of illustrative examples reflect the general principle of the object translator. Nevertheless, the code shown here is a much simplified version of the actual implementation.

Generating Internal Representation

Generating the final internal representation involves checking the intermediate forms against possible semantic violations (such as conformity), removing redundant tuples in the intermediate form, processing placeholders and tags and replacing temporary object identifiers with permanent object identifiers.

Redundancy is introduced for several reasons. One is to maintain additional information for delayed processing. Another is to make it easier to structure the code (a recursion pattern, for example).

In the intermediate form, atomic data values (integers and strings) are represented as tuples of the following form:

```
TEMP_ID, integer, –1, <integer_value>
TEMP_ID, string, –1, <string_value>
```

In the final representation, atomic values are directly used instead. The following procedure is used to remove these redundant tuples:

1). Find all tuples that references the temporary identifiers for atomic values

2). Substitute corresponding atomic values for the references.

Example 53: Continuing with the previous example. The following is the result after removing tuples for atomic data values,

```
TEMPID_1, name,  -1, TEMPID_2
TEMPID_1, age,   -1, 2
TEMPID_2, first, -1, 'Joe'
TEMPID_2, last,  -1, 'Dow'
```

□

Translating object descriptions containing object tags requires additional semantic processing. The following two questions need to be addressed: How are they translated? How do they match concrete objects? The assumption that object tags are non-repeating also simplifies the translation process. Each object tag is always translated into a distinct abstract object. However, object tags may be related to each other by placeholders. There are two cases to consider when translating an object tag, depending on whether or not the tag is related to another tag by co-referencing placeholders. (Recall that no placeholder means distinct a distinct placeholder.) Assume placeholder H_1 is associated with tag T_1 and placeholder H_2 with tag T_2. (H_1 and H_2 are always different.)

Case 1 ($H_1 \equiv H_2$):
the two abstract objects denoted by T_1 and T_2 have the same pattern-matching power (any object matching with one also matches with the other.)

Case 2 ($H_1 \neq H_2$):
the two abstract objects denoted by T_1 and T_2 perform independent matching.

While placeholders disappear after the translation, object tags remain. This information is used to determine what object to create in the final stage of translation. The final step in this process is to replace temporary object identifiers by permanent object identifiers.

Example 54: The QSM representation for the object description in this series of examples contains the following tuples (with the FIELD_INDEX column omitted).

```
PERMID_1, name,  PERMID_2
PERMID_1, age,   2
PERMID_2, first, 'Joe'
PERMID_2, last,  'Dow'
```

□

8.4. Translating Types

The translator for types consists of a parser, a semantic checker and a QSM representation generator, much like the object translator. Our discussion focuses on the representation of type definitions and semantic checking issues.

Type-Defining Objects

Type definitions are stored as type-defining objects (TDOs). A TDO is a concrete object with, among others, a component abstract object. Recall that the type definition for TDOs is the following:

```
TypeDef = (typename → String,
           supertypes →→ TypeDef,
           template → All?)
```

The field "typename" contains the external name of a type ("Employee", for example). The field "supertypes" contains a type-defining object for each of the supertypes. The field "template" contains an abstract object defining a structure template (the template object, or TO) for the instances of the type.

We have to use the most general type ("All") on "template" in a TDO, because this field contains a template for the TDO itself, and "All" is the only type that bounds from above every other type.

Example 55: We write the following type definition, "Implementation", for VLSI CAD design application as a TDO.

```
Implementation = Cell:(cellImplementation → CellImplementation)
Cell = (cellName → String,
        cellPorts →→ Port)
```

The TDO for type "Implementation" is (with placeholder or tags omitted):

```
TypeDef:(typename → 'Implementation',
         supertypes →→ TDO_for_Cell,
         template → Implementation:?
                      (cellName → TO_for_String,
                       cellPorts →→ TO_for_Port,
                       cellImplementation TO_for_CellImplementation))
```

□

Using TO for Conformity Checking

When translating an object from its external description into the QSM form, the object is also added to one or more typeset. Each type defines a conformity condition, which the object being added may violate, for example, when the object does not contain all the fields defined by the type (call the object defective). We have three alternatives for dealing with this problem at the time of object creation: ignoring the defects (and restricting the use of such object until the defects are removed), repairing the defects (using null values) and rejecting objects with defects. The prototype uses the third approach: defective objects are not created.

The template objects in TDOs provide a number of convenient ways for conformity checking. We can formulate the problem as *query-execution* or *structure-matching*. To check if an object conforms to a type in the first formulation, we proceed as follows.

1) Retrieve the type-defining for the type.

2) Extract the template object from the TDO.

3) Execute the TO as a query

4) The object conforms to the type if and only if it is included in the answer.

The conformity-checking-as-query-execution approach demonstrates the versatility of the abstract objects and an interesting application of command objects. This approach to conformity checking can be expensive, as the potential search space is the entire database. Checking conformity condition in the second formulation directly makes use of the matching semantics of the abstract objects. The template object in the TDO and the object in question are directly handed to the matcher of the query processor, which decides whether the object matches the TO, hence whether the object conforms to the type.

Checking Type Specialization

When new types are defined, they are related to existing types by the subtype relation. If no explicit supertype is given, a new type by default has the supertype "All". To relate two existing types using the subtype relation (via $\mathbf{T}_1 < \mathbf{T}_2$, for example), we need to check if the intended subtype is indeed a specialization of the supertype. Similarly, the template objects in the TDOs can be used for this purpose. Since one type is a specialization of another if the former contains all fields defined in the latter, we use the TO of the latter to match the TO of the former. The condition for subtyping is not violated if and only if the match succeeds.

8.5. Processing Commands

There are two kinds of processing on commands: Translating their external description into internal representation (as command-defining objects, or CDOs) and storing them, or translating their external description into executable forms and executing them.

Translating Commands

Consider the following command:

```
ViewStates[nm → String:N]
{
 View[S] <== states → GeoState:S(stateName → N), ViewStates(nm → N)
}
```

Its QSM representation consists of the following tuples.

```
aCommandID, type, TDO_for_Compound
aCommandID, name, 'ViewStates'
aCommandID, nm@argument, anAbstractStringObject
aCommandID, command, aSimpleCommandID
aSimpleCommandID, type, TDO_for_Command
aSimpleCommand, View@operation, anAbstractGeoStateObject
aSimpleCommand, states, anAbstractGeoStateObject
aSimpleCommand, name, 'ViewStates'
aSimpleCommand, nm@argument, anAbstractStringObject
anAbstractGeoStateObject, stateName, anAbstractStringObject,
```

Executing Commands

Executing a command is a two step procedure: Translating the command into an executable form and then the actual execution. The executable form of a command consists of an executable pattern and an executable action. The execution of commands, as discussed previously, consists of pattern-matching followed by an imperative operation. An executable pattern (pattern-matching goal) is a conjunction of simple literals in the QSM, and is executed by the Prolog interpreter. As patterns always take a collection as the search scope for matching, a root object ("$TEDMRootObject") is defined to hold top-level collections. Thus, given the following example pattern

states $\rightarrow$ GeoState:S(stateName $\rightarrow$ N)

Its executable form is

```
$TEDMRootObject, states, VariableS
VariableS, type, TDO_for_GeoState
VariableS, stateName, VariableN
```

An executable action (operation goal) is also a conjunction of Prolog literals, either user defined operations (such as "PromoteToManager") or built-in predicates (such as "assert" and "retract"). In addition, there are a number of user-defined predicates that function as an access interface to the built-in update predicates ("assert" and "retract") simulating the object memory.

The communications between the pattern-matching phase and the action phase make the results of binding available to the operations. There are several ways to satisfy this communications needs. A simple and elegant technique is to append the operation goal to the pattern-matching goal, and rely on the unification mechanism of Prolog to naturally establish the communication. A second approach is to generalize the parameter-passing strategy for compound-command invocation, namely, creating temporary binding objects. The current implementation uses a third approach. In this method, binding objects are retained as parameters to the Prolog procedure invocation, rather than being kept as temporary objects in the workspace. An additional procedure is used to explicitly bind the objects from pattern-matching to the placeholders in the operation.

8.6. Tailoring for Special Languages and Chapter Summary

To conclude this chapter, we offer a few comments on the tailorability of the prototype architecture with respect to syntax variations of the different languages. We point out that all the languages discussed in this chapter, except the object definition language, can be implemented with an alternative elegant architecture. As the object definition language is a canonical language, it is logical to construct preprocessors for the other languages, as they are special-purpose dialects of the canonical language. The same idea can be extended to general cases. A presentation format or concrete syntax can always be easily supported by adding a preprocessor front-end to the canonical language processor.

In this chapter, we described a prototype for the TEDM model. The overall architecture and the functions of its major components were discussed. The process for translating objects into internal representations was given in detail. The issues and strategies in semantic checking during translations were also presented.

CHAPTER 9

SUPPORTING COMPUTATION

We propose a novel technique for incorporating computation into databases. We extend a database with computational objects, or objects with functional interpretations. Such an object, with given arguments in an environment, can be evaluated to produce a *value*, the value of the computational object.

We use the *graph reduction* technique from functional programming as a basis for supporting computation. We discuss semantic issues for such an extension, including ordering of pattern-matching and computation and maintaining intermediate results. As we know, commands are executed in two phases in TEDM, a *matching* phase followed by an *action* phase. During the first phase, objects matching the *pattern* are bound to pattern variables. The second phase performs imperative operations on the matched objects (again, through variable binding). A strategy for extending the query processor, then, is to add a *reduction* phase to the two phase-execution scheme, as is proposed in [Maier87a]. The graph-reduction technique (and the G-machine) fit well with the proposal, since the data entities in graph-reduction are directed graphs, or complex objects. There is not much difficulty in mapping the nodes in a graph to an object in the database. For example, we can proceed as follows. Define a database type for each node type in the graph, whose instances represent nodes in an expression graph. Primitive data types in the G-machine, such as integers, strings, cons and nils, are all mapped straightforwardly. For a "function" node, define a type to contain the name of the function and the compiled code for the function. *Computational* objects would be used to represent a "apply" node.

This chapter provides an overview of the graph reduction technique and an abstract reduction machine (the G-machine) in Section 1. Section 2 discusses issues in extending the query processor with an evaluation engine (the G-machine). We discuss a three-phase query evaluation strategy, *matching-reduction-execution*, and its operational semantics. In Section 3, we look at the possibilities of parallel computation for certain aspects in query processing. In particular, pattern-matching produces a set of results, so parallelizing the reduction process at a large granularity is feasible. The chapter summary is given in Section 4.

9.1. Reduction in the G-Machine

A reduction architecture evaluates an expression by transforming it through a series of intermediate forms until it cannot be further transformed, under a set of rewrite rules. The expression is then said to be in normal form, which is the value attributed to the original expression. In a *pure reduction* system (such as beta-reduction, combinator reduction or string reduction), the control—the selection of the next reduction step—is derived dynamically from the form of the current expression. An alternative is *programmed reduction*, in which the steps of a computation are still applications of reduction rules, but where control is derived from a static analysis of the original expression.

By using a representation for an expression that is a graph, rather than a simple string, multiple occurrences of a common subexpression are captured as multiple references to a common subgraph. *Graph reduction* refers to a reduction process in which expressions are represented as such graphs, which avoids redundant re-evaluations of a common expression. In a graph-reduction architecture, a reducer traverses the graph of an applicative expression to locate a redex (a subexpression for rewriting), chosen according to the computation rule. A graph-rewrite rule is then applied to replace the redex with a subgraph representation of its value. Other parts of the graph that referenced the redex now have access to this value. In a programmed graph-reduction architecture, the reducer executes a program derived from the definition of the function used at the redex to perform the replacement.

The G-machine is an abstract architecture for programmed graph reduction. It was defined by Thomas Johnsson and Lennart Augustsson [Johnsson84] as the evaluation model for a compiler for lazy ML (LML). In it, the reduction step is quite efficient, as the program computes in terms of a stack of pointers (the P-stack) into the expression graph.

The G-machine consists of a stack and a graph memory. The graph memory holds nodes interconnected as a directed graph. The stack contains pointers to nodes in the graph to be manipulated by instructions. Graph manipulation involves creation of new nodes, change of connection patterns and reduction using predefined operations. Evaluation of an expression starts by constructing a graph in the graph memory to represent the expression, and invoking precompiled code to transform the graph into a new graph representing the normal form of the expression, if the expression has a normal form.

As an example of graph reduction with the G-machine, consider the following program.

letrec fact n = if n=0 then 1 else n × fact (n-1) in fact 3

The definition of the factorial function is compiled into the following G-machine instructions. (Read left to right.)

```
push 1;          eval;        get;
pushint 0;       get;         eq;
jfalse label_1;  pushint 1;   jmp label_2;
label label_1;   push 1;      eval;
get;             push 2;      eval;
get;             pushint 1;   get;
sub;             mkint;       pushfun fact;
mkap;            eval;        get;
mul;             mkint;       label label_2;
update 3;        ret 2;
```

The evaluation of the expression "fact 3" after building an initial graph for the expression, starts by executing an "eval" instruction. Each recursive invocation of the "fact" function does the following:

1) Push the argument node and the function node onto the stack (unwinding).

2) Evaluate the argument and if its value is 0, return with a value 1.

3) Or else if another recursive call is needed, save a copy of the argument value in the dump stack (a temporary storage for primitive data values and saving current context upon entry of a function).

4) Decrement the current argument and build an expression graph for the recursive call.

5) Make a new stack to hold the pointer to the new expression graph, save the current stack and instruction register using the dump and then enter a new evaluation cycle.

The graphs during the execution of the first few instructions are illustrated in Figure 9.1.

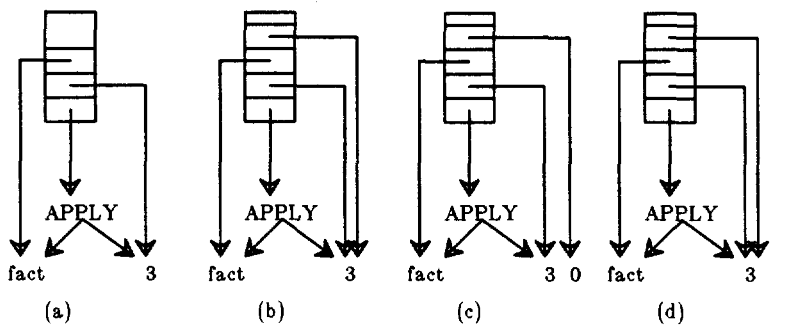

Figure 9.1 Effects of the First Few Instructions

In the diagram, (a) is the configuration after a so-called *unwinding* process, in which pointers to the argument node (INT 3) and the function node (FUNC fact) are pushed onto the stack (which grows upwards), and compiled code for the function (fact) is looked up from the environment and loaded into the instruction memory. State (b) is the result of executing the first instruction (push 1), which pushes another pointer to the argument node (INT 1). The next instruction in the compiled code (eval) does not change the configuration since the expression to be evaluated (INT 1) is already reduced. The next instruction (get) moves the integer value 1 from the top of the stack to the dump (not shown in the diagram). State (c) is the result of executing the instruction "pushint 0", which creates a new graph node (INT 0) and pushes the pointer to it onto the stack. The next few instructions primarily operate on the dump stack. First, the "get" instruction pushes an integer value 0 onto the dump. Next, the "eq" instruction compares the top-most two items (3 and 0) in the dump stack and replaces them with a Boolean value "false". Then the "jfalse label_1" instruction consumes the

Boolean value and makes a jump to "label_1". The configuration after the instruction "push 1" is shown in (d).

Figure 9.2 shows the configuration of the machine upon entry of the next recursive call. Configuration (a) illustrates the state before the call. Configuration (b) is the state at the time after the initial unwinding of the function call is done.

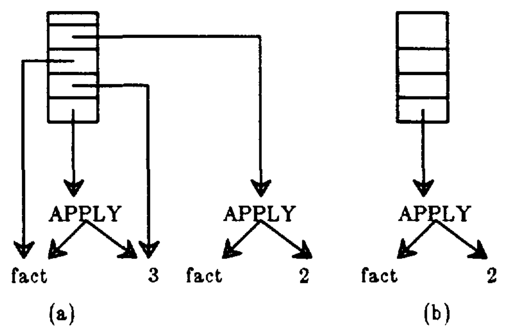

Figure 9.2 Configuration After One Recursion

For more detailed introduction to the G-machine, the graph reduction techniques and their hardware realizations, the reader is referred to [Johnsson84, Kieburtz85, 87a, 87b, 87c].

The graph memory supports seven kinds of nodes. Two of them, *CONS* and *NIL*, support the *list* built-in data structure; another three, *INT*, *BOOL* and *FUNC*, represent primitive data types. The *APPLY* node denotes an application of a function to its arguments. The *HOLE* node is a temporary placeholder for constructing graphs containing recursions.

9.2. The Semantics of Computational Objects

For the discussion in this section, we call the result of pattern-matching during command execution a *binding matrix*, For example, suppose that a student named "John Smith" is stored in the database, and he has taken "Calculus", "Algebra" and "Physics", and got grade "A" for all of them. Then the binding matrix for the command

View[C] $\Longleftarrow$
students $\rightarrow$ Student:S(name $\rightarrow$ (first $\rightarrow$ 'John', last $\rightarrow$ 'Smith')),
enrollments $\rightarrow$ Enrollment:E(course $\rightarrow$ C, student $\rightarrow$ S, grade $\rightarrow$ 'A')

after pattern-matching may look like the following.

[S:s_1, E:e_1, C:c_1]
[S:s_1, E:e_2, C:c_2]
[S:s_1, E:e_3, C:c_3]

where s_1, e_1, e_2, e_3, c_1, c_2 and c_3 are all object identifiers.

We explore two ways to extend query processing based on the interactions between the pattern-matching phase and the reduction phase. The first mixes the two and second separates them.

Mixing Reduction with Pattern-Matching

In the mixed strategy, pattern objects may contain computational objects and vice versa. We view pattern-matching phase as a reduction process. The outcome of the process is a normalized graph (containing no more reduction patterns) and a binding matrix. The binding matrix is used by the action phase in the usual way. We write a database command using the following variant format.

Action[X_1, ..., X_n] $\Longleftarrow$ Reduction[Y_1, ..., Y_m]

This strategy may be *unsafe*: It is possible to generate infinite binding matrices. A computational object may evaluate to infinite values, if it contains unbound arguments. However, we can always precompile the reduction process to obtain a safe plan, if it exists.

To illustrate the mixed strategy, consider the following example command, which finds in a person profile database all people whose last name is the concatenation of their first names and middle names, and displays them.

```
View[P] <== persons → Person:P
 (name → PersName:N(first → F,
                    middle → M,
                    last → Concat:L(arg1 → F, arg2 → M)))
```

where the pattern object of type "PersName" and tagged by "N" has three fields, "first", "middle" and "last". Field values for the "first" field and the "second" field are pattern variables "F" and "M" respectively. The field value for the "last" field is a computational object that concatenates its two arguments. Also assume the database contains the following three Person objects.

```
1). Person:(name → PersName:(first → 'BOW',
                             middle → 'LONG',
                             last → 'BOWLONG'))
2). Person:(name → PersName:(first → 'FRANK',
                             middle → 'FELIX',
                             last → 'BROWN'))
3). Person:(name → PersName:(first → 'JOY',
                             middle → 'SMITH',
                             last → 'JOYSMITH'))
```

The previous command execution can proceed as follows. In the first step, it starts from the top of the pattern object, retrieving all three "Person" objects. The second step grabs the name field values of the "Person" objects. At this point, the binding matrix is as shown below. (We make up object identifiers as needed.)

[P:p_1, N:n_1]
[P:p_2, N:n_2]
[P:p_3, N:n_3]

Next, the pattern matching continues, fields "first" and "middle" and their values are retrieved, resulting in the following binding matrix.

[P:p_1, N:n_1, F:'BOW', M:'LONG']
[P:p_2, N:n_2, F:'FRANK', M:'FELIX']
[P:p_3, N:n_3, F:'JOY', M:'SMITH']

There are two ways to go on from here. One is to go ahead and fully instantiate the binding matrix and get:

[P:p_1, N:n_1, F:'BOW', M:'LONG', L:'BOWLONG',]
[P:p_2, N:n_2, F:'FRANK', M:'FELIX' L:'BROWN',]
[P:p_3, N:n_3, F:'JOY', M:'SMITH', L:'JOYSMITH']

Then the computational object is activated to perform the reduction. The result of each reduction is compared with corresponding "L" binding, invalidating the second binding vector. The final binding matrix is then

[P:p_1, N:n_1, F:'BOW', M:'LONG', L:'BOWLONG']
[P:p_3, N:n_3, F:'JOY', M:'SMITH', L:'JAYSMITH']

Or alternatively, we can interleave pattern matching with reduction. With this strategy, reduction is performed as soon as "F" and "M" are bound. The result of reduction then replaces "L" as an added selection criterion (indicate using L′). Since reduction does not change binding matrices. the binding matrix remains the same after the reduction:

[P:p_1, N:n_1, F:'BOW', M:'LONG', L':'BOWLONG']
[P:p_2, N:n_2, F:'FRANK', M:'FELIX', L':'FRANKFLIX']
[P:p_3, N:n_3, F:'JOY', M:'SMITH', L':'JOYSMITH']

When the pattern matching process resumes, a dynamic test condition is activated to validate the current binding vectors, which is to test if the object n_1 has a "last" field with value 'BOWLONG' , and so on, producing the same result:

[P:p_1, N:n_1, F:'BOW', M:'LONG', L:'BOWLONG']
[P:p_3, N:n_3, F:'JOY', M:'SMITH', L:'JAYSMITH']

Either execution plan produce the same final binding matrix as the result of the reduction phase. The second plan will likely perform better, since there is no wasted variable bindings to database objects; but it requires a precompiled safe execution plan or an intelligent runtime scheduler. The first plan, on the other hand, has the advantage of being conceptually simpler, since there is no interleaving of pattern matching with reduction. Note there is a difference in the treatment of the variable "L" in the two different approaches. In the first approach, it is just another pattern variable; while in the second, it is used as a temporary storage for the results of a computational object. We also point out that the command itself can be rewritten into the following form, which makes the first execution plan clearer. The collection "compobjects" is used to hold computational objects.

```
View[P] <==
  persons → Person:P(name → PersName:N(first → F,
                                          middle → M,
                                          last → L)),
  compobjects → Concat:L(arg1 → F, arg2 → M)
```

Precompilation of command execution plan with matching and reduction interleaving is based on the notion of *safe* patterns and *safety dependencies*, a concept similar to *functional dependencies.*

Separating Reduction from Pattern-Matching

In the separate strategy, graphs nodes of pattern objects and graph nodes of computational objects do not interact, although they may share variables. To emphasize this non-interactiveness, we write a command as follows:

$$\mathrm{Action}[X_1, \ldots, X_n] \Longleftarrow \mathrm{Reduction}[Y_1, \ldots, Y_m] \mid \mathrm{Pattern}[Z_1, \ldots, Z_p],$$

In this case, the binding matrix is initially created during pattern-matching. It is modified by the reduction phase. The data retrieved during pattern matching are fed to a reduction phase, which can change the binding matrix in two ways. First, it can augment the binding matrix in width by adding certain reduction results to the matrix, those temporarily stored in variables not appearing in the pattern. Second, it can also shrink the binding matrix in height by invalidating certain binding vectors produced by pattern matching, when results of the reduction are in conflict with the results of the matching.

Consider another example. Suppose we define a company employee database as follows. A "Person" object has a "name" field of type "PersonName", and a "dob" (date of birth) field of type "Date". An "Employee" object is also a "Person" object and in addition has a "dept" field, a "salary" field and a "yos" (year of service) field. Suppose also the company's pay raise policy is stated using a "PayScale" object, which has a "base" field defining the base pay, a "rate" field specifying the rate of increase, and a "period" field stating how often a raise occurs. The database schema is summarized below.

```
PersonName = (first → String, middle → String, last → String)
Date = (day → Number, month   → Number, year → Number)
Person = (name → PersonName, dob → Date)
Department = (dname → String, manager → Employee)
Employee = Person: (dept → Department, salary → Number, yos → Number)
PayScale = (base → Number, rate → Number, period  → Number)
```

The command below updates the salaries of employees according to the company's pay raise plan.

```
E(salary → NS)
 <== Add:NS(arg1 → BS, arg2 →
        Mul:(arg1 → BS, arg2 →
        Mul:(arg1 → RT, arg2 →
        Div:(arg1 → YH, arg2 → PD))))
  |  Employee:E(yos → YH),
     PayScale:(base → BS, rate → RT, period → PD)
```

Assume the company has three employees, who have been with the company for 5, 10 and 15 years respectively. Also assume the "PayScale" object is as shown below:

```
PayScale:(base → 25000, rate → 10, period → 5)
```

where the rate means 10 percent.

The command execution starts up by entering the pattern matching phase and producing the following binding matrix, say:

[E:e_1, YH: 5, BS:25000, RT:10, PD:5]
[E:e_2, YH:10, BS:25000, RT:10, PD:5]
[E:e_3, YH:15, BS:25000, RT:10, PD:5]

Next the reduction phase takes over, performing the desired arithmetic and the results are reflected in this final binding matrix:

[E:e_1, YH: 5, BS:25000, RT:10, PD:5, NS:27500]
[E:e_2, YH:10, BS:25000, RT:10, PD:5, NS:30000]
[E:e_3, YH:15, BS:25000, RT:10, PD:5, NS:32500]

which can then be used to update the employee objects.

Semantics Independent of Evaluation Order

We investigate the semantics of a computational object that do not depend on specific evaluation orders in this section. We call it *orderless* semantics. We will see that if a computational object is unsafe under this abstract orderless semantics, then there is no safe evaluation order for it. On the other hand, a computational object that is safe under the orderless semantics may not necessarily have a safe evaluation order.

We will collectively refer to both computational objects and pattern objects as patterns, viewing computation as a special kind of pattern matching, one that can potentially produce infinitely many bindings. A dual view is that a pattern object is a computational object that can be reduced nondeterministically. Notationally, we may always rewrite a computational object Func:X(X_1, ..., X_n) into a pattern form FuncPat(X_1, ..., X_n, X), such that if X = Func(X_1, ..., X_n) then there is an object FuncPat(X_1, ..., X_n, X) matching the pattern.

We introduce pattern graphs for visualizing the discussion. Consider a pattern of the form $P_1(X_{1,1}, ..., X_{1,m1}), ..., P_n(X_{n,1}, ..., X_{n,mn})$, where variables are distinct within a single pattern term, but they can overlap across the terms. Then the pattern graph for the pattern is constructed as follows. The node set is $\{P_1, ..., P_n\}$, that is, a one node for each pattern term P_1 through P_n. The edges are undirected and the edge set includes an edge, with label X, connecting two nodes P_i and P_j, if and only if pattern P_i

and pattern P_j share variable X.

After constructing a pattern graph, we can associate with each node a binding matrix, as we did for entire patterns. Constructing a binding matrix for a node not involving computations relies on matching the pattern against the database. In a top-down fashion, an initial binding matrix for the root variable of the pattern is first built. As the pattern matching moves down through the substructure of the pattern, this binding matrix is updated in two ways, by join extension or by literal selection.

The binding matrix for a computational node is taken to be the function (in the mathematical sense of an n-nary relation) the node denotes. As such, a binding matrix for a computational node may be infinite, with infinite number of binding vectors. Nevertheless, assuming we are only dealing with primitive recursive functions, the set of row elements is always totally enumerable. Therefore, a binding matrix for a computational node P, denoting $P(X_1, ..., X_n)$, can also be written as

$$\begin{array}{l}
[X_1{:}x_{1,1}, X_2{:}x_{2,1}, ..., X_n{:}x_{n,1}] \\
[X_1{:}x_{1,2}, X_2{:}x_{2,2}, ..., X_n{:}x_{n,2}] \\
[X_1{:}x_{1,3}, X_3{:}x_{3,2}, ..., X_n{:}x_{n,3}] \\
\ldots
\end{array}$$

For commonly seen functions, such as arithmetic, logic and comparison operations, we always use their natural semantics that we are all familiar with.

Example 56: The binding matrix for Add:(arg1 $\rightarrow$ X, arg2 $\rightarrow$ Y, sum $\rightarrow$ Z), an addition function on positive integers, is

[X:0, Y:0, Z:0]
[X:0, Y:1, Z:1]
[X:1, Y:0, Z:1]
[X:0, Y:2, Z:2]
[X:1, Y:1, Z:2]
...

□

Individual binding matrices are combined using join extension to form a binding matrix as the meaning for the whole pattern. Join extension is similar to the natural join operation in relational databases. The join extension of two binding matrices M_1 and M_2, denoted $M_1 \bowtie M_2$, is based on their shared variables, that is, the edges connecting their nodes. Note it is possible that $M_1 \bowtie M_2$ produces a finite binding matrix, even if each of the operands is infinite.

If a pattern graph has nodes P_1, ..., P_n, and M_i is a binding matrix for P_i, then the binding matrix for the whole pattern graph is defined to be

$$M_1 \bowtie M_2 \bowtie \ldots \bowtie M_n$$

A pattern is *safe* if its binding matrix is guaranteed to be finite. A pattern is *computationally safe* if there is an order of evaluation to compute its binding matrix such that each intermediate binding matrix is finite. Note that according to this definition, computational safeness implies safeness, since the final binding matrix computed is a special intermediate result. Hence we have the following observation. If either the separate strategy or the mixed strategy results in a finite binding matrix for a pattern, then the pattern is safe. But a safe pattern need not be computationally safe.

Example 57: The pattern below is safe, but not computationally safe, where "Ident" is an identity function.

Add:S(arg1 → X, arg2 → Y), Mul:S(arg1 → X, arg2 → Y),
Ident:S(arg → 4).

□

On the other hand, if a pattern is not safe under the orderless semantics, we can assert that there exists no order by which the pattern can be safely evaluated. A non-safe pattern is not computationally safe.

There are some connections between pattern graphs and the notion of computational safeness. For each pattern, we construct its pattern graph, and use the following procedure to reduce the graph.

1) remove database pattern nodes and their incident edges
2) repeatedly remove computational nodes that have at most one incident edge

It is easily seen that if the above procedure results in an empty graph, then the pattern is computationally safe. (We make an implicit assumption that there can be no column with infinitely many values that are related to a single row on the remaining columns in its interpretation of any computational object.)

Finally, computationally safe patterns can be compiled into an execution plan based on the notion of safety dependency. At any point in the procedure above, we say that a computational object is ***applicable*** if its node has at most one incident edge. Then a strategy for precompilation is that a computational object is scheduled for execution as soon as it becomes applicable.

The notions of pattern graphs and query safety appeared in [Maier83], in the context of query processing for relational data models. Queries are represented as query graphs and query optimizations are processes in which query graphs are transformed to obtain more efficient executions. Query safety is introduced when computed relations are proposed in [Maier81]. The generalized notion of safeness for non-first normal form databases and to deductive databases can be found, for example, in [Bancilhon86, Zaniolo86].

9.3. Further Extension

We consider two directions for further extension. The first is the opportunity for parallel execution created by the pattern-matching style of query processing and reduction. The second is an extended abstract machine that would allow tighter coupling of graph reduction with pattern-matching.

Parallel Reduction

Each binding independently provides all the data needed for an execution of the reduction engine. Under a sequential model, the whole reduction process can be logically viewed as an iterative computation, in which the binding vectors are fed to the reduction engine one at a time, and the results of the reduction are built up as another binding matrix, which is later merged with the original binding matrix in one of two ways, binding augmentation or binding validation, as discussed earlier. However this computation can be fully parallelized, since there are no data dependencies among binding vectors.

We propose to achieve parallelism at a very high level (or large grain), that is, at independent reduction graph level. Schematically, the idea is illustrated as follows (Figure 9.3).

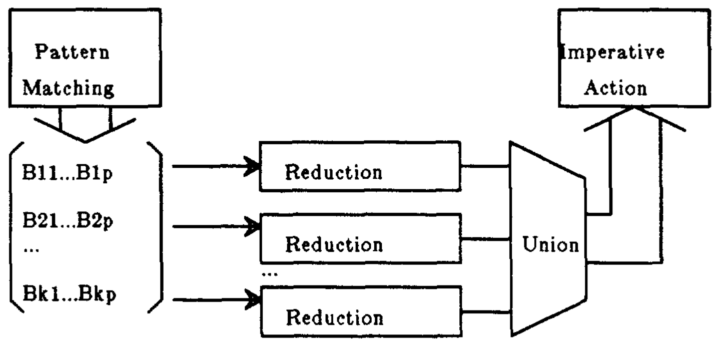

Figure 9.3 A Parallel Reduction Scheme

Each binding vector from the intermediate binding matrix forks out a reduction process, and each reduction process performs reduction independently. Finally, a union process is added to collect results from individual processes. Conceptually, this parallel procedure is extremely simple and clean. Yet we expect there is substantial amount of parallelism to be gained from it, especially when pattern matching always produces tall binding matrices, since there is practically no communication cost overhead involved.

As an example, consider the example from the last section. The fork and union processes are illustrated below (Figure 9.4).

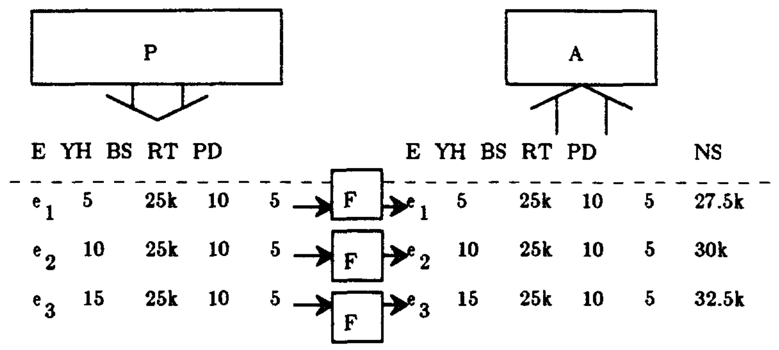

Figure 9.4 An Example Parallel Reduction

In Figure 9.4 above, the box labeled with "P" denotes the pattern

Employee:E(yos → YH),
  PayScale:(base → BS, rate → RT, period → PD)

The box labeled with "A" denotes the action

E(salary → NS)

And the box labeled with "F" denotes the function

```
Add:NS(arg1 → BS, arg2 →
   Mul:(arg1 → BS, arg2 →
   Mul:(arg1 → RT, arg2 →
   Div:(arg1 → YH, arg2 → PD))))
```

Schaefer [Schaefer90] explores fine grain parallel graph reductions, where reduction computation is mapped to a distributed memory architecture. In his approach, a virtual processor is allocated to every reducible expression in the program graph. The advantage there is that reduction of a single expression is very simple, so it is possible to use "flyweight" tasks in which the cost of context switch is extremely low.

Tighter Coupling

Tight coupling between pattern matching and reduction can be done at the level of integrating the two processing engines. The idea is that the G-machine can be extended architecturally with abstract instructions capable of constructing patterns in the graph memory, coordinating the pattern matching process and retaining answers from the pattern matching.

Instructions for constructing patterns are similar to those already present in G-machine constructing expression graphs. We may want to enrich the kinds of nodes in the graph memory, to include more data types, such as strings and variables (or abstract objects [Zhu88]).

To coordinate pattern matching against the database, we may introduce a "match" instruction, whose function is to use the pattern referenced by the top element of the pointer stack to match the database objects, and to save the bindings in an additional memory repertory (see below). All of the work can be done by invoking a system defined pattern matching routine.

To retain the results of executing the "match" instruction, (that is, binding matrices) we propose to add an additional memory resource to the machine's architecture, call it ***pattern memory***. Pattern memories are dedicated to supporting table structured data. It is allocated in preparation for the execution of a "match" instruction and is modified as the "match" instruction proceeds.

9.4. Chapter Summary

In this chapter, we discussed a novel approach to computing in databases. We explored the feasibility through a case study, where we incorporated the graph-reduction technique into the TEDM data model. The result is a database language supporting complex object manipulation as well as computation.

CHAPTER 10

SUPPORTING VIRTUAL DATA

A database contains both stored and implied information, the information not directly represented by physical data but deducible from the known information by inference rules. Virtual data are important as they provide alternative ways for looking at the same set of physical data. An application can choose a logical presentation that best fits its needs. The relational databases support derived data using *views*. Each view is a virtual table defined by a query statement. In deductive databases, there is no clear line between virtual data and physical data (though one could say facts are physical). In this case, the database contains both facts and rules. Query-answering becomes a demonstration for deducibility.

TEDM supports virtual data using *database rules*. Three kinds of virtual information are supported by database rules: virtual membership, virtual fields and virtual objects. A database rule is like a definite clause in Prolog, both in syntax and to some degree in semantics. Nevertheless, database rules may contain complex structure that may not readily be representable as Prolog rules.

Section 1 discusses three kinds of rules in TEDM, for virtual (typeset) membership, virtual fields and virtual objects. They are called type rules, field rules and object rules, respectively. Section 2 describes the handling of type rules and field rules: Direct translation into Prolog rules and the use of them in query processing. Section 3 investigates issues in translating object rules, and describes a meta-interpreter to handle object rules. Section 4 is the chapter summary.

The ideas discussed in this chapter are not implemented in the current version of the prototype. However, they are part of an earlier implementation [Zhu85]; and the translation of rules and the meta-interpreter are drawn from that implementation.

10.1. Database Rules

Like commands, rules are made up of a head and a body. The body of a rule is a template for matching database objects. Both the form and the meaning of a body in a rule are the same as a command pattern. The head of a rule is a restricted form of a command head. The restriction is needed to control the complexity of the definable rules. The general syntax for the rules is written as:

<Virtual Type> | <Virtual Field> | <Virtual Object> ← <Pattern>.

A rule is a deferred command executed on demand, triggered by the execution of other commands. As we do not materialize the rules, the effect of the rules is visible only during the execution of commands. The problem of updating on virtual data is beyond the scope of this thesis.

Type Rules

To define a virtual type of an object, the following type rule is used.

T:**V** ← <Pattern>

where **T** is a type name and **V** is a variable (placeholder) that gets bound to a database object during pattern-matching. The rule defines a virtual type for a class of objects that satisfy the pattern.

Example 58: This rule defines a virtual type "ManagementEmployee" for people with a title "Manager".

$$\text{ManagementEmployee:E} \leftarrow \text{Employee:E(title} \rightarrow \text{'Manager')}$$

If, say, 'Joe' and 'Mary' are managers, then they would be of both type 'Employee' and type 'ManagementEmployee'. Thus, they would be selected for viewing in the following command.

$$\text{View[E]} \Longleftarrow \text{managers} \rightarrow \text{ManagementEmployee:E}$$

□

Field Rules

A field rule is written as

$$\mathbf{V}(\mathbf{f}_1 \rightarrow \mathbf{w}_1, \ldots, \mathbf{f}_n \rightarrow \mathbf{w}_n) \leftarrow \text{<Pattern>}$$

where $\mathbf{w}_i$ is a variable or a constant. The variables are assumed to be bound by the pattern. The rule defines, for each object bound to **V**, n virtual fields, with name $\mathbf{f}_1, \ldots, \mathbf{f}_n$, and value $\mathbf{w}_1, \ldots, \mathbf{w}_n$, respectively.

Example 59: This rule defines a virtual field "grandFather" for each person object.

$$\begin{aligned}&\text{P(grandFather} \rightarrow \text{G)} \leftarrow \\ &\qquad \text{person} \rightarrow \text{Person:P(father} \rightarrow \text{Person:F(father} \rightarrow \text{Person:G))}\end{aligned}$$

□

A database rule is *safe* if the closure of the database with respect to the rule is finite. Type rules and field rules are always safe as they only reference existing objects, and there are only finitely many such objects.

Object Rules

Virtual objects are defined using rules of the following form.

$$\mathbf{T}{:}{*}(\mathbf{f}_1 \rightarrow \mathbf{w}_1, \ldots, \mathbf{f}_n \rightarrow \mathbf{w}_n) \leftarrow \text{<Pattern>}$$

The asterisk (*) assigns a *temporary* object identifier to each virtual object.

Example 60: The rule below defines virtual Couple objects.

$$\text{Couple:*(husband} \rightarrow \text{H, wife} \rightarrow \text{W)} \leftarrow \text{Person:H(wife} \rightarrow \text{Person:W)}$$

□

There exist two difficult problems in deriving new objects. First, object rules with recursion (direct or indirect) may not be safe, as the computation may not terminate, in an attempt to create an infinite number of objects. Thus, we require that object rules contain no recursion. (Note that recursion is still allowed in field rules, and the transitive-closure type of computation is expressible.)

The second problem is that virtual objects may have multiple identifiers. Different activations would assign different identifiers to a virtual object. One way to maintain unique identifiers in different activations is to use *Skolem functions* as temporary object identifiers. Each virtual object has an identity that is functionally dependent on the specific binding of the variables. Consequently, object identifiers remain the same as long as the binding does not change. The idea of using Skolem functions as object identifiers is investigated in Chen and Warren [ChenW89] and Kifer and Wu [Kifer89].

10.2. Translating Simple Rules

Type rules and field rules are simple rules as they are directly translated into Prolog clauses.

Translating Type Rules

A type rule for an object is translated into a single Prolog clause, with the head asserting the virtual type for the object and the body defining the condition that the object must satisfy. The general form of a translation is the following. (We use QSM in description, and omit the column FIELD_INDEX.)

qsm(**V**, type, **T**) :— <TranslationForPattern>

Example 61: The translation of the type rule

ManagementEmployee:E ← Employee:E(title → 'Manager')

is

```
qsm(E, type, 'ManagementEmployee') :—
    qsm(E, type, 'Employee'), qsm(E, title, 'Manager')
```

□

Translating Field Rules

A field rule for an object is translated into multiple Prolog clauses, one for each virtual field, with its head defining the virtual field. All Prolog clauses for a particular field rule have the same body, obtained by translating the pattern:

qsm(**V**, $\mathbf{f}_1$, $\mathbf{w}_1$) :— <TranslationForPattern>
...
qsm(**V**, $\mathbf{f}_n$, $\mathbf{w}_n$) :— <TranslationForPattern>

Example 62: The translation of the field rule

P(grandFather → G) ← Person:P(father → Person:F(father → Person:G))

is

```
qsm(P, grandFather, G) :—
    qsm(P, type, 'Person'), qsm(P, father, F),
    qsm(F, type, 'Person'), qsm(F, father, G).
```

□

10.3. Handling Object Rules

Object rules cannot be directly translated into Prolog clauses, as doing so would cause the database to produce incorrect answers. To illustrate the problem, consider the object rule for virtual couples:

Couple:*(husband → H, wife → W) ← Person:H(wife → Person:W)

Suppose we simply translated this rule into the following Prolog clauses:

```
qsm(C, type, 'Couple') :— newObjectID(C),
    qsm(H, type, 'Person'), qsm(W, type, 'Person'), qsm(H, wife, W).
qsm(C, husband, H) :—
    qsm(H, type, 'Person'), qsm(W, type, 'Person'), qsm(H, wife, W).
qsm(C, wife, W) :—
    qsm(H, type, 'Person'), qsm(W, type, 'Person'), qsm(H, wife, W).
```

Notice that the variable C occurs in the head only in the last two clauses. The intended meaning for C is an object identifier for each virtual object defined by the object rule. Although C changes as it can denote different virtual objects, it should remain a constant for a particular object. Consider the scenario (a is married to b, c is single) described in the QSM as the following facts:

qsm(a, type, 'Person'), qsm(b, type, 'Person'), qsm(a, wife, b)

```
qsm(c, type, 'Person')
```

Intuitively, only one virtual Couple object should be derivable from this database. However, the goal

```
:— qsm(O, type, 'Couple'), qsm(O, husband, X), qsm(O, wife, Y).
```

would succeed for every person in the database, effectively deriving three virtual Couple objects, one each for a, b and c, respectively.

The problem arises because variables in Prolog are local in clauses. In order for the translation of an object rule to work, variable binding should occur simultaneously in all the clauses generated from the object rule. (In the example, the binding needs to occur in all three Prolog clauses.) As Prolog does not have the power to support global binding, a meta-interpreter was developed to handle object rules.

The meta-interpreter is built in Prolog itself. Object rules are stored as Prolog facts: rule(HEAD, BODY), where both HEAD and BODY are lists. For example, the object rule for virtual couples is stored as

```
rule([qsm(C, type, 'Couple'), qsm(C, husband, H), qsm(C, wife, W)],
    [qsm(H, type, 'Person'), qsm(W, type, 'Person'), qsm(H, wife, W)]).
```

Virtual objects are accumulated along the way when they are derived. Both the current goal and the virtual objects are kept in the form of a list, (GO_LIST and VO_LIST), To solve a goal, the interpreter tries the following alternatives in turn:

1). Solve the goal using the database

2). Solve the goal using virtual objects derived so far;

3). Solve the goal using rule clauses: If the goal matches one of the terms in a rule head, add the rule head to the virtual objects, and add the rule body to the current goal.

To see how the meta-interpreter works, consider the example of this section again:

```
DATABASE:
    qsm(a, type, 'Person'), qsm(b, type, 'Person'), qsm(a, wife, b)
    qsm(c, type, 'Person')
RULE:
    rule([qsm(C, type, 'Couple'), qsm(C, husband, H), qsm(C, wife, W)],
        [qsm(H, type, 'Person'), qsm(W, type, 'Person'), qsm(H, wife, W)]).
GOAL:
    :— qsm(O, type, 'Couple'), qsm(O, husband, X), qsm(O, wife, Y).
```

The initial state of the meta-interpreter is

```
GO_LIST = [qsm(O, type, 'Couple'), qsm(O, husband, X), qsm(O, wife, Y)],
VO_LIST = [].
```

The current subgoal is "qsm(O, type, 'Couple')". It is matched against the database and the match fails. The match against VO_LIST also fails. The third trial, matching against the rules, succeeds. Thus the head of the of the rule is added into VO_LIST, and the body of the rule to GO_LIST. The current state becomes:

```
GO_LIST = [qsm(H, type, 'Person'), qsm(W, type, 'Person'), qsm(H, wife, W),
        newObjectID(VO), qsm(VO, husband, X), qsm(VO, wife, Y)]
VO_LIST = [qsm(VO, type, 'Couple'), qsm(VO, husband, H), qsm(VO, wife, W)].
```

The next three subgoals successfully match the database, binding a to H and b to W respectively. At this point, the predicate newObjectID is executed, producing for the potential virtual object a unique object identifier (say, id2000 in this case). The state is:

```
GO_LIST = [qsm(id2000, husband, X), qsm(id2000, wife, Y)],
VO_LIST = [qsm(id2000, type, 'Couple'), qsm(id2000, husband, a),
```

qsm(id2000, wife, b)].

The next two steps successfully resolve the remaining two subgoals against VO_LIST, resulting in an empty GO_LIST. Thus, the following virtual Couple object (with identifier id2000) is derived:

Couple:C(husband → Person:a, wife → Person:b).

On backtracking, the subgoal "qsm(id2000, wife, Y)' is retried and it fails, as id2000 is unique. Similarly, the retry for "qsm(id2000, husband, X) also fails. The predicate "newObjectID" is not backtrackable: it also fails. The system gets to the state:

```
GO_LIST = [newObjectID(VO), qsm(VO, husband, X), qsm(VO, wife, Y)],
VO_LIST = [qsm(VO, type, 'Couple'), qsm(VO, husband, a), qsm(VO, wife, b)].
```

On further backtracking, the goal "qsm(a, wife, W)" still fails, leaving the system at the state:

```
GO_LIST = [qsm(H, type, 'Person'), qsm(H, wife, W), newObjectID(VO),
        qsm(VO, husband, X), qsm(VO, wife, Y)],
VO_LIST = [qsm(VO, type, 'Couple'), qsm(VO, husband, H), qsm(VO, wife, W)].
```

Resatisfying the goal "qsm(H, type, 'Person')" would succeed in binding H to b and c, leaving the system state at, respectively:

```
GO_LIST = [qsm(b, wife, W), newObjectID(VO),
        qsm(VO, husband, X), qsm(VO, wife, Y)],
VO_LIST = [qsm(VO, type, 'Couple'), qsm(VO, husband, b), qsm(VO, wife, W)].
```

and

```
GO_LIST = [qsm(c, wife, W), newObjectID(VO),
        qsm(VO, husband, X), qsm(VO, wife, Y)],
VO_LIST = [qsm(VO, type, 'Couple'), qsm(VO, husband, c), qsm(VO, wife, W)].
```

However, the next subgoal fails in either state, and the resolution fails.

10.4. Chapter Summary

The mechanism for supporting derived information in TEDM is discussed. Three kinds of rules are identified. Type rules and field rules are directly translated into Prolog clauses. The problem with direct translation of object rules is demonstrated. The operation of a meta-interpreter for handling object rules is also discussed.

CHAPTER 11

FUTURE DIRECTIONS AND CONCLUDING REMARKS

We have been exploring other possible extensions to TEDM. In particular, the following areas are considered for future research: more support for application modeling, the use of abstract objects in schema definitions and the role of local variables in compound commands and workspaces, We discuss these directions in this chapter. At the end, a few remarks are given to conclude the thesis.

The chapter is organized as follows. Section 1 discusses a number of new concepts for conceptual modeling, including ***abstract fields*** and ***abstract types***. Their potential uses are also examined. Section 2 discusses the use of abstract objects in schema definitions. Section 3 proposes more extensions to compound commands. Summary and concluding remarks are provided in Section 4.

11.1. More Modeling Concepts

The ideas discussed in this section grew out of a study reported in [Anderson89]. In the study, TEDM is used to represent CSG solids. Their study shows that, in some cases, the conceptual framework of TEDM does not support the application modeling in a direct way. The study suggests that, for complex modeling tasks such as CSG modeling, it is desirable that the data model support a wider range of modeling constructs and allow these constructs to be flexibly combined. In particular, they proposed the concepts of ***abstract types*** and ***abstract fields***, and used these concepts in their design.

Abstract Fields

Complex objects are constructed from simpler ones using the concept of fields. We made the following assumption about the set of names (or labels) for fields: It is a non-structured set. Each element is independent from the rest of the set. However, allowing field names to be organized in meaningful ways adds a lot to the modeling power of the data model. For example, a type describing a corporate office may contain a field with the name "officer". When creating instances, it is helpful to include concrete titles such as "marketingVP", "researchVP", and so on. In this instance, both "marketingVP" and "researchVP" are corporate officers, but they are more concrete and carry more information.

The concept of abstract fields intends to allow this kind of scenario to be modeled elegantly. It defines a hierarchical structure on the set of field names. In the example above, as "marketingVP" $\leq$ "officer" and "researchVP" $\leq$ "officer", we can use the "marketingVP" or "researchVP" for object creation. The field name "officer" is called an *abstract field*. KL-ONE [Brachman85], a system for knowledge representation, allows classification on slot names, which is essentially similar to the notion of abstract fields.

Abstract Types

An abstract type is a type that does not have instances. In object-oriented programming, a good practice is to to define common generic behavior in an abstract superclass, and supplement the superclass with variations and more specific behavior using subtypes. For example, the "Collection" class and its subclasses such as "Set" and "Bag" in Smalltalk-80 belong to this approach. In conceptual modeling, abstract types can provide a common abstraction on generic entities. The difference is that an abstract type can be made instantiable without having to extend the type hierarchy (by defining subtypes). An abstract type is instantiated by concrete types using *substitution*. For example, an abstract type "GenericShape" can define the structure of generic geometric figures (square, circles, polygons, etc.), as a sequence of coordinate points. The abstract type can be instantiated using, say, "Square@GenericShape", "Circle@GenericShape", "Polygon@GenericShape" and so on. Then, the instances of the type "GenericShape" can be created as squares, circles or polygons. Note that "Square" (or "Circle@GenericShape" or "Polygon@GenericShape") does not need to be a subtype of "GenericShape".

Optional Fields

Optional fields are useful in early design phases. Optional fields reflect the anticipation by the designer on the final outcome of a complex object structure, and allows flexibility to exist in the design. The concept is a compromise between prescriptive typing (allowing more fields) and strict typing (allowing no more, no less).

For example, in designing a data path for a VLSI circuit, it may not be clear in the outset whether or not a certain buffer would improve the performance. We can make use of optional fields in this situation:

$$\text{DataPath} = (\ldots, [\text{buffer} \rightarrow \text{Buffer}], \ldots)$$

where the construct "[...]" encloses optional fields. Thus, a DataPath may or may not contain a field named "buffer". More importantly, if the field "buffer" does occur in such an object, its type is constrained to be of "Buffer".

11.2. Abstract Objects as Value Specifications

Abstract objects and types have many things in common. For each type, there is a conformity condition, which classifies objects as those that conform and those that do not. Similarly, for each abstract object, there is a classification of objects based on whether or not each object matches the abstract object. Furthermore, to test the conformity condition of a type on an object, the abstract object of the type-defining object is used to match the object: The object conforms to the type if and only if the match succeeds. These parallelisms suggests that we can use abstract objects the same way we use types.

We explore one such use — the use of abstract objects in type definitions, as value specifications. For example, given an abstract object described by

Rectangle:RECT?(width → 10)

which, incidently, matches "Rectangle" objects with a width of 10, to define a special type "Width10LayoutUnit", we can use the abstract object (description) in the following type definition:

```
Width10LayoutUnit = (rect → Rectangle:RECT?(width → 10),
                     position → Point )
```

In most cases, using an abstract object in a type definition amounts to adding more constraints to the object conformity condition with respect to that type. In the previous example, for an object to conform to the type "Width10LayoutUnit", the object has to have both a "rect" field and a "position" field. In addition, the value of the "rect" field, a "Rectangle" object, has to be of a specific width (10).

The appearance of an abstract object in a type definition does not make the abstract object a type. To trigger such a transition, we introduce a new form of type definition. For example, we can use the following

Width10Rectangle := Rectangle:RECT(width $\rightarrow$ 10)

to promote the abstract object to a type.

11.3. Extensions to Compound Commands

This section suggests two ways for extending compound commands: the use of local variables and local rules in compound commands.

Local Variables

Compound commands are a mechanism for control abstraction: the execution of a compound command has the same effect as its individual commands executed in sequence. (One additional condition is that either all of the commands are executed or none of them is executed.) The concepts of local variables and static scoping from block-structured programming languages can also be incorporated into compound commands.

Local variables in a compound command are visible to its individual commands. The binding of a local variable becomes invalid beyond the execution of the compound command. Local variables are similar to parameters, except the bindings of the former are established and used by the compound command itself, but those of the latter are provided or used by the surrounding environment. The approach we used to handle arguments and parameters can be applied to local variables. For example, an argument term may contain local variables, in addition to arguments. Access to local variables can be provided using argument terms containing local variables.

Local Rules

The scope and the effect of database rules can also be defined by compound commands. This is useful for controlling the activation of rules. For example, for data security reasons, some rules can be used only in certain privileged transactions.

Rules defined in a compound command are local rules. They are visible only to the individual commands of the compound commands. The general form for compound commands can be extended as follows.

$$
\begin{array}{l}
\text{CommandName}[\text{arg}_1 \rightarrow \text{T}_1\text{:V}_1, \ldots, \text{arg}_n \rightarrow \text{T}_n\text{:V}_n] \\
\{ \\
\quad \text{locvar} \rightarrow \text{Type:LocVar}, \ldots \\
\quad \text{VirtualData}_1 \leftarrow \text{RPattern}_1, \text{RArgumentTerm}_1; \\
\quad \ldots \\
\quad \text{VirtualData}_m \leftarrow \text{RPattern}_m, \text{RArgumentTerm}_m; \\
\quad \text{Action}_1 \Leftarrow \text{Pattern}_1, \text{ArgumentTerm}_1; \\
\quad \ldots \\
\quad \text{Action}_k \Leftarrow \text{Pattern}_k, \text{ArgumentTerm}_k; \\
\}
\end{array}
$$

11.4. Summary and Concluding Remarks

We studied a number of interesting and important issues in object-oriented databases. Several new concepts are proposed to extend the database technology for non-traditional applications. The study is based on a specific logic-based object-oriented data model (TEDM), but the concepts discussed are generally applicable. We also reported a prototype implementation of TEDM.

We described in detail several important languages for defining objects, types, commands, as well as for composing interactive queries. With the goal of "total objectification", and in searching for a canonical language for object description, we studied different ways for describing complex data structure. The result of the search is a language (the object definition language) that can describe other model constructs such as types, patterns and commands. The abstract object extension to the object model plays an important role for the goal "total objectification."

Rules are the mechanism for deductive query processing and data derivation. Three kinds of database rules are supported: type rules, field rules and object rules. Two problems concerning object rule translation are: consistent object identifiers and safety in derivations.

The prototype system consists of three major functional blocks: the model manager, the user interface facility and the communication library. The model manager has support for three languages, for defining objects, defining types and defining commands. Addition of new languages or modification of existing languages is a simple task in the prototype.

The notion of abstract objects is the key contribution of this thesis. It is a significant step towards a complete self-descriptive data model: treating types and commands as database objects. From a theoretical perspective, it provides a clean semantics for variables and command objects in such data models. The pattern-matching semantics for abstract objects form a consistent basis for all of their uses in the thesis.

BIBLIOGRAPHY

[Abiteboul87]	Abiteboul, S. and Hull, R., "IFO: A Formal Semantic Database Model," *ACM Transactions On Database Systems*, 12(4), 1987.
[Abrial74]	Abrial, J. R., "Data Semantics," *Data Base Management*, North Holland, Amsterdam, 1974.
[AitKaci84]	Ait-Kaci, H., "Lattice Theoretic Approach to Computation Based on A Calculus of Partially Ordered Type Structures," *Ph.D. Thesis*, University of Pennsyvania, 1984.
[Albano85]	Albano, A., Cardelli, L. and Orsini, R., "Galileo: A Strongly-Typed, Interactive Conceptual Language," *ACM Transactions On Database Systems*, 10(2), 1985.
[Anderson86]	Anderson, T. L., Ecklund, E. F. Jr. and Maier D., "PROTEUS: Objectifying the DBMS User Interface," *Proceedings of the International Workshop on Object-Oriented Database Systems*, 1986.
[Anderson89]	Anderson, T. L., Ohkawa, H., Gjovaag, J., Maier, D. and Shulman, S., "Representing CSG Solids Using a Logic-Based Object Data Model," *Proceedings of the International Workshop on Object-Oriented Database Systems*, 1989.
[Andrews87]	Andrews, T. and Harris, C. "Combining Language and Database Advances in an Object-Oriented Development Environment," *Proceedings of 86 Conference on Object-Oriented Programming Systems, Languages and Applications*, 1987.
[Astrahan76]	Astrahan, M. M., Blasgen, M. W., Chamberlin, D. D., Eswaran, K. P., Gray, J. N., Griffiths, P. P., King, W. F., Lorie, R. A., McJones, P. R., Mehl, J. W., Putzolu, G. R., Traiger, I. L., Wade, N. W. and Waston, R., "System R: Relational Approach to Database Management," *ACM Transactions on Database Systems*, 1(2), 1976.
[Atkinson83]	Atkinson, M. P., Chisholm, K. J., Cockshott, W. P., Bailey, P. J. and Morris, R., "An Approach to Persistent Programming," *The Computer Journal*, 26(4), 1983.

[Atkinson84] Atkinson, M. P., Bailey, P. J., Cockshott, W. P., Chisholm, K. J. and Morris, R., "Progress with Persistent Programming," *Database — Role and Structure, an Advanced Course*, Stocker, P. M., Gray, P. M. D. and Atkinson, M. P. (eds.), Cambridge University Press, New York, 1984.

[Atkinson87] Atkinson, M. P. and Buneman, O. P., "Types and Persistence in Database Programming Languages," *ACM Computing Surveys*, June, 1987.

[Augustsson88] Augustsson, L. and Johnsson, T., *Lazy ML User's Manual*, Preliminary Drafet, 1988.

[Bancilhon85] Bancilhon, F., Kim, W. and Korth, H. F., "A Model of CAD Transactions," *Proceedings of 11th International Conference on Very Large Data Bases*, 1985.

[Bancilhon86] Bancilhon F. and Khoshafian, F. S., "A Calculus for Complex Objects," *Proceedings of ACM Symposium on Principles of Database Systems*, 1986.

[Banerjee87] Banerjee, J., Chou, H. T., Garza, J. F., Kim, W., Woelk, D., Ballou, N. and Kim, H. J., "Data Model Issues for Object-Oriented Applications," *ACM Transaction on Office Information Systems*, 5(1) 1987.

[Banerjee87a] Banerjee, J., Kim, W., Kim, H. J. and Korth, H. F., "Semantics and Implementations of Schema Evolution in Object-Oriented Databases," *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, 1987.

[Batory84] Batory, D. S. and Buchmann, A. P., "Molecular Objects, Abstract Data Types and Data Models: A Framework," *Proceedings of 10th International Conference on Very Large Data Bases*, 1984.

[Batory85] Batory, D. S. and Kim, W., "Modeling Concepts for VLSI Objects," *ACM Transaction on Database Systems*, 10(3) 1985.

[Beeri87] Beeri, C., "On Combining Object Orientation and Logic Programming," *XP8.5i Workshop*, Oregon Graduate Center, 1987.

[Brachman85] Brachman, R. J. and Schmolze, J. G., "An Overview of the KL-ONE Knowledge Representation System," *Cognitive Science*, Vol. 9, 1985.

[Breazu-Tannen89]
Breazu-Tannen, V., Buneman, P. and Ohori, A., "Can Object-Oriented

Databases Be Statically Typed?" *Proceedings of the 2nd Workshop on Database Programming Languages*. 1989.

[Brodie82] Brodie, M. L., "Axiomatic Definitions For Data Model Semantics," *Information Systems*, 7(2) 1982.

[Buneman79] Buneman, P. and Frankel, R. E., "FQL — A Functional Query Language," *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, 1979.

[Buneman86] Buneman, O. P. and Atkinson, M. P., "Inheritance and Persistence in Database Programming Languages," *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, 1986.

[Cardelli84] Cardelli, L., "Amber," *Technical Memorandum*, TM 11271-840924-10, AT&T Bell Laboratories, 1984.

[Carey86] Carey, M. J., DeWitt, D. J., Frank, D., Graefe, G., Richardson, J. E., Shekita, E. J. and Mutalikrishna, M., "The Architecture of the EXODUS Extensible DBMS," *Proceedings of International Workshop on Object-Oriented Databases*, 1986.

[Carey86a] Carey, M. J., DeWitt, D. J., Richardson, J. E. and Shekita, E. J., "TObject and File Management in the EXODUS Extensible Database System," *Proceedings of 12th International Conference on Very Large Data Bases*, 1986.

[Chen76] Chen, P., "The Entity-Relationship Model: Toward a Unified View of Data," *ACM Transactions On Database Systems*, 1(1), 1976.

[ChenW89] Chen, W. D. and Warren, D. S., "C-Logic of Complex Objects," *Proc. of the ACM Symp. on Principles of Database Systems*, 1989.

[ChenW89a] Chen, W. D., Kifer, M. and Warren, D. S., "Hi-Logic as a Platform for Database Languages," *Proc. of the 2nd International Workshop on Database Programming Languages*, 1989.

[Cockshott84] Cockshott, W. P., Atkinson, M. P., Chisholm, K. J., Bailey, P. J. and Morris, R., "Persistent Object Management System," *Software Practice and Experience*, 14(1), 1984.

[Codd70] Codd, E. F., "A Relational Model for Large Shared Data Banks," *Communications of the ACM*, 13(6), 1970.

[Codd79]	Codd, E. F., "Extending the Database Relational Model to Capture More Meanings," *ACM Transactions on Database Systems*, 4(4), 1979.
[Copeland84]	Copeland, G. and Maier, D., "Making Smalltalk a Database System," *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, 1984.
[Copeland85]	Copeland, G. and Khoshafian, S., "A Decomposition Storage Model," *Unpublished Manuscript*, MCC, 1985.
[Cox86]	Cox, B. J., *Object-Oriented Programming — An Evolutionary Approach*, Addition Wesley, 1986.
[Derrett85]	Derrett, N., Kent, W. and Lyngbaek, P., "Some Aspects of Operations in an Object-Oriented Database," *IEEE Database Engineering*, 8(4), 1985.
[Dittrich86]	Dittrich, K. R., Gotthard, W. and Lockemann, P. C., "DAMOKLES — A Database System for Software Engineering Environments," *Proceedings of IFIP Workshop on Advanced Programming Environments*, 1986.
[Ecklund87]	Ecklund, D. J., Ecklund, E. F. Jr., Eifrig, B. O. and Tonge, F. M., "DVSS: A Distributed Version Storage Server for CAD Applications," *Proceedings of International Conference on VLDB*, 1987.
[Ege87]	Ege, A. and Ellis, C. A., "Design and Implementation of GORDION, an Object Base Management System," *Proceedings of 13th International Conference on Very Large Data Bases*, 1987.
[Emden76]	van Emden, M. and Kowalski, R., "The Semantics of Predicate Logic as a Programming Language", *Journal of the Association for Computing Machinery*, 23(4), 1976.
[Fishman87]	Fishman, D. H., Beech, D., Cate, H. P., Chow, E. C., Connors, T., Davis, J. W., Derrett, N., Hoch, C. G., Kent, W., Lyngbaek, P., Mahbod, B., Neimat, M. A., Ryan, T. A. and Shan M. C., "Iris: An Object-Oriented Dababase Management System," *ACM Transaction on Office Information Systems*, 5(1) 1987.
[Flynn88]	Flynn, B. C., "User Interface Management for Database Objects," *Research Paper*, Oregon Graduate Center, 1988.

[Gallaire84] Gallaire, H., Minker, J. and Nicolas, J. M., "Logic and Databases: a Deductive Approach," *ACM Computing Surveys*, 16(2), 1984.

[Goldberg83] Goldberg, A. and Robson, D., *Smalltalk-80, The Language and its Implementation*, Addison-Wesley, 1983.

[Graefe87] Graefe, G., "The EXODUS Optimizer Generator," *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, 1987.

[Hammer81] Hammer, M. and Mcleod D., "Database Description with SDM: A Semantic Database Model," *ACM Transactions On Database Systems*, 6(3), 1981.

[Hammerstrom86]
Hammerstrom, D., "Microsimulator (BitSim) Style," *CSE 529 Course Material*, Oregon Graduate Center, 1986.

[Honeywell80] Honeywell Information Systems, *Series 60 (level 68). Multics Relational Data Store Reference Manual*, Order Number AW53, 200 Smith Street, Waltham, Massachusetts, 1980.

[Hull84] Hull, R. and Yap, C. K., "The Format Model: A Theory of Database Organization," *Journal of the ACM*, 31(3), 1984.

[Hull89] Hull, R., Morrison, R. and Stemple, D. (eds.), *Proceedings of the 2nd Workshop on Database Programming Languages.* 1989.

[Jensen74] Jensen, K. and Wirth, N., *Pascal User's Manual and Report*, Springer Verlag, 1974.

[Johnsson84] Johnsson, T., "Efficient Compilation of Lazy Evaluation," *Proceedings of ACM SIGPLAN*, 1984.

[Katz83] "Managing the Chip Design Database", Katz, R. H., *IEEE Computer*, 16(12), 1983.

[Kowalski78] Kowalski, R., "Logic for Data Description", *Logic and Databases*, Nicolas, J. M., Gallaire, H. and Minker, J. (eds.), Plenum Press, New York, 1978.

[Kemper87] Kemper, A., Lockemann, P. C. and Wallrath, M., "An Object-Oriented Database System for Engineering Applications," *Proceedings of the*

ACM-SIGMOD International Conference on the Management of Data, 1987.

[Kent79] Kent, W., "Limitations of Record-Based Information Models," *ACM Transactions On Database Systems,* 4(1), 1976.

[Kifer89] Kifer, M. and Wu, J., "Maier's O_Logic Revisited," *Proc. of the ACM Symp. on Principles of Database Systems,* 1989.

[Kifer89a] Kifer, M. and Lausen, G., "F-Logic: A Higher-Order Logic for Reasoning about Objects, Inheritance and Scheme," *Proc. of the ACM International Conference on Management of Data,* 1989.

[Kim87] Kim, W., Chou, H. T. and Banerjee, J., "Operations and Implementation of Complex Objects," *Proceedings of 13th International Conference on Very Large Data Bases,* 1987.

[Kuper84] Kuper, G. M. and Vardi, M. Y., "A New Approach to Database Logic," Proceedings of the ACM Symposium on Principles of Database Systems, 1984.

[Lindsay87] Lindsay, B., McPherson, J. and Pirahesh, H., "A Data Management Extension Architecture," *Proceedings of the ACM-SIGMOD International Conference on the Management of Data,* 1987.

[Lloyd84] Lloyd, J. W., *Foundation of Logic Programming,* Springer-Verlag, Berlin, 1984.

[Lorie83] Lorie, R. and Plouffe W., "Complex Objects and Their Use in Design Transactions," *Proceedings of SIGMOD Conference, Database Week,* 1983.

[Maier83] *The Theory of Relational Databases,* Maier, D., Computer Science Press, 1983.

[Maier84] Maier D. and Price D., "Data Model Requirements for Engineering Applications," *Proceedings of IEEE 1st International Workshop on Expert Database Systems,* 1984.

[Maier85] Maier, D., "TEDM Data Model," *Working Paper,* Department of Computer Science and Engineering, Oregon Graduate Center, 1985.

[Maier86] Maier, D., Stein, J., Otis A. and Purdy, A., "Development of an Object-Oriented DBMS," *Proceedings of 86 Conference on Object-Oriented Programming Systems, Languages and Applications*, 1986.

[Maier86a] Maier, D. and Stein J., "Indexing in Object-Oriented DBMS," *Proceedings of International Workshop on Object-Oriented Databases*, 1986.

[Maier87] Maier, D. and Warren D. S., *Computing With Logic*, Benjamin-Cummings, 1987.

[Maier87a] Maier, D., "Why Database Languages are a Bad Idea," *First International Workshop on Database Programming Languages*, Roscoff, France, 1987.

[Maier89] Maier, D., "Why Isn't There An Object-Oriented Data Model," *IFIP-89*, San Francisco, 1989.

[Maier89a] Maier, D., Zhu, J. and Ohkawa, H., "Features of the TEDM Object Model," *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, Tokyo, Japan, 1989.

[Mylopoulos80] Mylopoulos, J., Bernstein, P. A. and Wong H. K. T., "A Language Facility for Designing Database-Intensive Applications," *ACM Transactions on Database Systems*, 5(2), 1980.

[Ohkawa87] Ohkawa, H., "Mapping an Engineering Data Model to a Distributed Storage System," *Research Paper*, Oregon Graduate Center, 1987.

[Ohori88] Ohori, A., "Semantics of Types for Database Objects," *Proceedings of the International Conference on Database Theory*, LNCS 326, 1988.

[Ohori89] Ohori, A., Buneman, P. and Breazu-Tannen, V., "Database Programming in Machivelli — A Polymorphic Language with Static Type Inference," *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, 1989.

[Paul87] Paul, H. B., Scheck, H. J., Weikum, G. and Deppisch, U., "Architecture and Implementation of the Darmstadt Database Kernel System," *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, 1987.

[Porter88] Porter, H., *A Logic-Based Grammar Formalism Incorporating Feature-Structures and Inheritance*, Ph.D. Dissertation, Oregon Graduate

Institute, 1988.

[Rosenberg80] Rosenberg, L. M., "The Evolution of Design Automation to Meet the Challenges of VLSI," *Proceedings of 17th Design Automation Conference*, 1980.

[Schaffer86] Schaffer, C., Cooper, T., Bollis, B., Kilian, M and Wilpolt, C., "An Introduction to Trellis/Owl," *Proceedings of 86 Conference on Object-Oriented Programming Systems, Languages and Applications*, 1986.

[Schaefer90] Schaefer, B., *Massive Asynchronous Concurrency Through Parallel Combinator Reduction*, Ph.D. Dissertation, Oregon Graduate Institute, 1990.

[Schmidt83] Schimdt, J. W. and Brodie, M. L., *Relational Database Systems*, Springer-Verlag, 1983.

[Shipman81] Shipman, D., "The Functional Data Model and the Data Language DAPLEX," *ACM Transactions on Database Systems*, 6(1), 1981.

[Shoens79] Shoens, K. A. and Rowe, L. A., "Data Abstraction, Views and Updates in RIGEL," *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, 1979.

[Sidle80] Sidle, T. W., "Weakness of Commercial Database Management Systems in Engineering Applications," *Proceedings of 17th Design Automation Conference*, 1980.

[Smith77] Smith, J. M. and Smith, D. C. P., "Database Abstractions: Aggregation and Generalization," *ACM Transactions On Database Systems*, 2(2), 1977.

[Stonebraker76] Stonebraker, M., Wong, E., Kreps, P. and Held, G., Wade, N. W. and and Waston, R., "Design and Implementation of INGRES," *ACM Transactions on Database Systems*, 1(3), 1976.

[Stonebraker84] Stonebraker, M., Anderson, E., Hanson, E. and Rubenstein, B., "QUEL as a Data Type," *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, 1984.

[Stonebraker86] Stonebraker, M. and Rowe, L. A., "The Design of POSTGRES," *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, 1986.

[Stonebraker86a]Stonebraker, M., "Inclusion of New Types In Relational Data Base Systems," *Proceedings of IEEE International Conference on Data Engineering*, 1986.

[Stonebraker87] Stonebraker, M., "The Design of POSTGRES Rules System", *Proceedings of IEEE International Conference on Data Engineering*, 1987.

[Stroustrup86] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, New York, 1986.

[Su83] Su, S. Y. W., "SAM*: A Semantic Association Model for Corporate and Scientific Statistical Databases," *Information Sciences*, 29, 1983.

[Todd76] Todd, S., "The Peterlee Relational Test Vehicle — A System Overview," *IBM Systems Journal*, 15(4), 1976.

[Ullman83] *The Principles of Database Systems*, Ullman, J. D., Computer Science Press, 1983.

[Zaniolo85] Zaniolo, C., "The Representation and Deductive Retrieval of Complex Objects," *Proceedings of International Conference on VLDB*, 1985.

[Zdonik85] Zdonik, S., "Object Management Systems for Design Environments," *IEEE Database Engineering*, 8(4), 1985.

[Zdonik88] Zdonik, S., "Can Object Change Type? Can Type Objects Change?" *Manuscript*, Department of Computer Science, Brown University, 1988.

[Zhu85] Zhu, J., "Prototype Implementation of an Engineering Data Model," *Unpublished Manuscript*, Oregon Graduate Center, 1985.

[Zhu88] Zhu, J. and Maier, D., "Abstract Objects In An Object-Oriented Data Model," *Proceedings of 2nd International Conference of Expert Database Systems*, 1988.

[Zhu89] Zhu, J. and Maier, D., "Computational Objects In Object-Oriented Data Models," *Proceedings of 2nd International Workshop on Database Programming Languages*, 1989.

APPENDIX I: A CAD Database Application

This appendix contains a design object instance populated using the schema discussed in Chapter 2. In order to help simplify the presentation and improve its readability, we make several syntactical adaptations to TEDM's object definition language so that the format for the example is more natural to the domain of VLSI design. We point out that these adaptations affect only syntactical appearance. Also, as was discussed throughout the thesis, domain-specific presentation format is easily supported in TEDM.

Syntactic Variations and Their Specifications

Before a listing for the example application database is given, we describe each syntactical variations that we will use in this appendix. We outline a general specification mechanism for syntactic variations, as an extension to schema definitions.

First, instead of repeating field labels "portName" and "instanceName" for "PortRef" objects, we omit the labels and use the following form

```
<portName> of <instanceName>
```

where <portName> and <instanceName> denote values for the respective fields. This variation is described by the following extended type definition.

```
PortRef = (portName → String:, instanceName → String:)
    { <portName> of <instanceName> }
```

Similarly, to present "Port" objects, we use a <portName> followed by either an upward arrow or a downward arrow to indicate the signal direction of the port.

Port = (portName → String:, portDirection →→ String:)
{ <portName> (↑ | ↓) }

An "Instance" object is presented using

<instanceName> **of** <cellName> **from** <libraryName>

Instance = (instanceName → String:, cellName → String:, libraryName → String:)
{ <instanceName> **of** <cellName> **from** <libraryName> }

Values in this multiple occurrence field in "Connect" objects are simply written as a list.

Connection = (portRef →→ PortRef:)
{ '{' <PortRef>, ... '}' }

A "Cell" object is identified by a cell name, followed by a list of "Port" object descriptions.

Cell = (cellName → String:, cellPort →→ Port:)
{ **Cell** <cellName> **Port** '{' <Port>, ... '}' }

A "CellImplementation" object is described as a list of component instances and a list of connections among the components.

CellImplementation = (instance →→ Instance:, connection →→ Connection:)
{ **with Instances** <Instance> ... **and Connections** <Connection>) ... }

An "Implementation" object is represented by a list of ports and a representation of an "Implementation".

Implementation = Cell:(cellImplementation → CellImplementation:)
{ **Interface** '{' <Ports> ... '}' **Implementation** <CellImplementation> }

A "Library" object is abbreviated to a name, a type and a list of "Cell" representations.

```
Library = (libraryName → String:, libraryType → String:, libraryCell →→ Cell:)
    { <name> of <type> with <Cell> ... }
```

Finally, the representation for a "Design" object consists a list of "Library" representations and an "Implementation".

```
Design = (useLibrary →→ Library:, implementation → ImplementedCell:)
    { Design of <name> uses Libraries <Library> ... <ImplementedCell>
    End }
```

Listing of A Design Instance

```
Design of fourBitAdder
  uses Libraries default of global with
    Cell not    Port { not_1↓, not_2↑ }
    Cell and_2  Port { and_2_1↓, and_2_2↓, and_2_3↑ }
    Cell and_3  Port { and_3_1↓, and_3_2↓, and_3_3↓, and_3_4↑ }
    Cell and_5  Port { and_5_1↓, and_5_2↓, and_5_3↓, and_5_4↓, and_5_5↓,
                       and_5_6↑, and_5_7↑, and_5_8↑, and_5_9↑ }
    Cell nand_2 Port { nand_2_1↓, nand_2_2↓, nand_2_3↑ }
    Cell nor_2  Port { nor_2_1↓, nor_2_2↓, nor_2_3↑ }
    Cell adder_4 Port {adder_4_clk↓, adder_4_1↓, adder_4_2↓, adder_4_3↓,
                       adder_4_4↓, adder_4_5↓, adder_4_6↓, adder_4_7↓,
                       adder_4_8↓, adder_4_9↑, adder_4_10↑, adder_4_11↑,
                       adder_4_12↑ }
    Cell register_4 Port { register_4_clk↓, register_4_1↓, register_4_2↓,
                           register_4_3↓, register_4_4↓, register_4_5↑,
                           register_4_6↑, register_4_7↑, register_4_8↑ }
   Interface { inbus_B2_1↓, inbus_B2_2↓, inbus_B2_3↓, inbus_B2_4↓,
               reset↓, lda_2↓, ldb_2↓, restkn_1↓, phi1↓,
               phi2↓, out_1↑, out_2↑, out_3↑, out_4↑ }
   ImplementedCell with Components
    U_1 of and_5 from default
    U_2 of and_2 from default
    U_3 of and_2 from default
    U_4 of and_2 from default
    U_5 of and_2 from default
    U_6 of and_2 from default
    U_7 of not from default
    U_8 of nor_2 from default
    U_9 of not from default
    U_10 of nor_2 from default
    U_11 of not from default
```

```
U_12 of nor_2 from default
U_13 of nor_2 from default
U_14 of not from default
U_15 of nor_2 from default
U_16 of not from default
U_17 of nor_2 from default
U_18 of and_2 from default
U_19 of and_2 from default
U_20 of and_2 from default
U_21 of nor_2 from default
U_22 of nor_2 from default
U_23 of nor_2 from default
U_24 of and_2 from default
U_25 of and_3 from default
U_26 of and_2 from default
U_27 of and_3 from default
U_28 of and_2 from default
U_29 of nor_2 from default
U_30 of and_2 from default
U_31 of and_2 from default
U_32 of nand_2 from default
U_33 of register_4 from default
U_34 of register_4 from default
U_35 of adder_4 from default
U_36 of register_4 from default
and Connections
 { inbus_B2_1 of fourBitAdder, and_5_1 of U_1 }
 { inbus_B2_2 of fourBitAdder, and_5_2 of U_1 }
 { inbus_B2_3 of fourBitAdder, and_5_3 of U_1 }
 { inbus_B2_4 of fourBitAdder, and_5_4 of U_1 }
 { not_2 of U_7, nor_2_1 of U_8 }
 { reset of fourBitAdder, nor_2_2 of U_8, nor_2_2 of U_10, nor_2_2 of U_12 }
 { lda_2 of fourBitAdder, and_2_1 of U_3 }
 { not_2 of U_9, nor_2_1 of U_10 }
 { ldb_2 of fourBitAdder, and_2_1 of U_5 }
 { not_2 of U_11, nor_2_1 of U_12 }
 { restkn_1 of fourBitAdder, nor_2_2 of U_21,
 { phi1 of fourBitAdder, and_2_2 of U_24, and_3_1 of U_25, and_2_2 of U_26,
   and_3_1 of U_27, and_2_1 of U_28, and_2_1 of U_30 }
 { phi2 of fourBitAdder, and_2_5 of U_1, and_2_2 of U_2, and_2_2 of U_3,
   and_2_2 of U_4, and_2_2 of U_5, and_2_2 of U_6, and_2_1 of U_19,
   and_2_1 of U_20, and_2_2 of U_31 }
 { nor_2_3 of U_8, and_2_1 of U_2 }
 { nor_2_3 of U_10, and_2_1 of U_4 }
 { nor_2_3 of U_12, and_2_1 of U_6 }
 { and_5_6 of U_1, register_4_1 of U_34 }
```

```
  { and_5_7 of U_1, register_4_2 of U_34 }
  { and_5_8 of U_1, register_4_3 of U_34 }
  { and_5_9 of U_1, register_4_4 of U_34 }
  { and_2_3 of U_2, nor_2_1 of U_13, not_1 of U_14 }
  { and_2_3 of U_3, nor_2_2 of U_13, and_3_3 of U_25 }
  { and_2_3 of U_4, nor_2_1 of U_15, not_1 of U_16 }
  { and_2_3 of U_5, nor_2_2 of U_15, and_3_3 of U_27 }
  { and_2_3 of U_6, nor_2_1 of U_17 }
  { and_2_3 of U_19, nor_2_2 of U_17, and_2_2 of U_30 }
  { nor_2_3 of U_13, nor_2_1 of U_23 }
  { and_2_3 of U_20, and_2_1 of U_18, nor_2_2 of U_22, nor_2_2 of U_23 }
  { nor_2_3 of U_15, nor_2_1 of U_22 }
  { nor_2_3 of U_17, nor_2_1 of U_21 }
  { and_2_3 of U_18, and_2_2 of U_19 }
  { nor_2_3 of U_23, and_2_1 of U_24 }
  { not_2 of U_14, and_3_2 of U_25 }
  { nor_2_3 of U_22, and_2_1 of U_26 }
  { not_2 of U_16, and_3_2 of U_27 }
  { nor_2_3 of U_21, and_2_1 of U_28 }
  { nor_2_3 of U_29, and_2_2 of U_20 }
  { and_2_3 of U_24, nand_2_1 of U_32, not_1 of U_7 }
  { and_3_4 of U_25, register_4_clk of U_34 }
  { and_2_3 of U_26, nand_2_2 of U_32, not_1 of U_9 }
  { and_3_4 of U_27, register_4_clk of U_33 }
  { and_2_3 of U_28, nor_2_2 of U_29, not_1 of U_11 }
  { nand_2_3 of U_32, and_2_1 of U_31, nor_2_1 of U_29 }
  { register_4_5 of U_34, adder_4_1 of U_35 }
  { register_4_6 of U_34, adder_4_2 of U_35 }
  { register_4_7 of U_34, adder_4_3 of U_35 }
  { register_4_8 of U_34, adder_4_4 of U_35 }
  { register_4_5 of U_33, adder_4_5 of U_35 }
  { register_4_6 of U_33, adder_4_6 of U_35 }
  { register_4_7 of U_33, adder_4_7 of U_35 }
  { register_4_8 of U_33, adder_4_8 of U_35 }
  { adder_4_9 of U_35, register_4_1 of U_36 }
  { adder_4_10 of U_35, register_4_2 of U_36 }
  { adder_4_11 of U_35, register_4_3 of U_36 }
  { adder_4_12 of U_35, register_4_4 of U_36 }
  { register_4_5 of U_36, out_1 of fourBitAdder }
  { register_4_6 of U_36, out_2 of fourBitAdder }
  { register_4_7 of U_36, out_3 of fourBitAdder }
  { register_4_8 of U_36, out_4 of fourBitAdder }
End
```

BIOGRAPHICAL NOTE

The author was born 8 October 1957, in Nanchang City, Jiangxi Province, People's Republic of China. He has been married for five years to Xing Liu and they have a child, Yun (Eddie), age 3.

The author graduated from Wenjiang High School of Sichuan Province in 1975. He studied at South-China Institute of Technology where he received in 1982 his Bachelor of Science degree in Computer Engineering.

The author entered the Oregon State University in 1983 as a graduate student and completed in 1984 with the degree Master of Science in Computer Science.

The author started his graduate study at the Oregon Graduate Institute Of Science and Technology in 1985.

The author is currently a member of the technical staff in the Department of Science and Technology, U S WEST Advanced Technologies, 6200 South Quebec Street, Englewood, Colorado 80111.