A FORMAL SYSTEM FOR NETWORK DATABASES AND ITS APPLICATIONS TO INTEGRITY RELATED ISSUES

Dipayan Gangopadhyay

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-84-19 May 1984

A FORMAL SYSTEM FOR NETWORK DATABASES AND ITS APPLICATIONS TO INTEGRITY RELATED ISSUES

APPROVED BY SUPERVISORY COMMITTEE:

J. C. Browne

M. Gouda

A. G. Dale

J. Misra

A FORMAL SYSTEM FOR NETWORK DATABASES AND ITS APPLICATIONS TO INTEGRITY RELATED ISSUES

BY

DIPAYAN GANGOPADHYAY, B.E.TEL.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December, 1983

To My Parents

ACKNOWLEDGEMENTS

This dissertation would not have materialized without the guidance, interest and gentle reminders from my advisor, Professor James C. Browne. He not only helped me to organize and present the research, but provided constant encouragement and support throughout my graduate career at UT, Austin. I am grateful to him.

I wish to thank the members of my dissertation committee, Professors Jay Misra, Don Good, Mohamed Gouda and Alfred Dale, for their encouragement and many fruitful discussions. I am thankful to Dr. Umesh Dayal for helping me to organize the ideas during the earlier stage of this research. Professor Frank Brown also has my sincere thanks for several discussions which clarified the decidability issue presented in this dissertation.

During the period of this research, several people provided me with emotional support. In particular, I am thankful to my brother, Nirmal, my parents and my friends, Bob, Valerie and Sumitra, for their patience and selfless support. I would also like to thank Nancy Eatman and Ann Arnold for their friendship and ever ready assistance irrespective of their pressing responsibilities.

This research was supported in part by grants from NASA (NSG 1446), NSF (MCS-8214613) and Honeywell Inc. (50CP357-CP30).

The task of formatting this dissertation was greatly eased by use of automated document processing. The Scribe document formatter was conceived of and created by Brian Reid. The current version has been maintained and enhanced by Unilogic, Ltd. The Scribe format definitions for proper dissertation format for The University of Texas at Austin were developed by Richard Cohen.

Dipayan Gangopadhyay

The University of Texas at Austin
December, 1983

ABSTRACT

A formal system to reason about databases based on the Network Data Model is presented. The formal system consists of a first order many sorted assertion language, its proof theory, a simple data manipulation language (DML) and the proof-rules for its constructs. Questions about network databases such as whether a given schema is consistent or whether a transaction preserves integrity assertions can be formulated and answered in our formal system by applying theorem proving and program verification techniques.

To implement these proof-techniques effectively, we identify a decidable subset of our assertion language which consists of universally quantified Horn formulas. We present a polynomial time algorithm for deciding whether a given formula in this subset is a theorem or not. A large class of integrity assertions which arise naturally in Network databases is encompassed by the decidable subset and thus the consistency of a schema involving only these integrity assertions can be algorithmically checked. Moreover, for a wide variety of update transactions which are annotated by such integrity assertions, each verification condition turns out to be an implication of two Horn formulas and can therefore be proved or disproved by our algorithm.

From the database point of view, our work provides a formal foundation for Network databases and results in capabilities with respect to schema design and compile-time integrity enforcement which are at present absent for Network databases. From the point of view of programming methodology, our work provides an example of the state-of-the-art in program verification, namely utilizing the specificity of the application domain in deriving efficient theorem provers. Finally, our methodology of developing the formal system is based upon algebraic specification technique of abstract data types. We provide a practical solution to the problem of algebraically specifying shared mutable objects.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

Integrity assertions are statements about the relationships among various entities that exist in the real world. They form a basis for organizing the semantic knowledge about a database. It is required that the stored data in a database always satisfy these integrity assertions. Thus, on one hand database schemas must incorporate the integrity assertions and on the other hand, database transactions have to preserve the assertions as invariant properties of the stored data.

An application database design consists of defining the database schema and the transactions against the schema occurrences. Both these activities have to deal with integrity assertions. In defining the schema, the designers have to contend with the fact that the assertions are collected from different user groups and therefore can be contradictory or contain redundant information. A good schema design has, among other things, to remove contradictions and redundancy from the given set of integrity assertions. The programmers writing the transactions, on the other hand, have to ensure that the updates performed by their transactions will not violate the integrity assertions defined in the schema. Three tasks related to integrity assertions emerge from the preceding discussion:

- detect contradictions in the integrity assertions of a given schema,
- optimize a schema with respect to the number of integrity assertions, and
- ensure correctness of transactions with respect to the preservation of integrity assertions.

In the database literature, the first two tasks fall under the topic of schema design and the last one under integrity management.

In relational databases, methods have been developed to aid the designers with the tasks of schema design and integrity management. The schema design methods are based upon the capability to express integrity assertions as statements in a formal language and to infer assertions from other assertions. The ability to infer assertions is crucial in answering many important questions during relational schema design, such as whether two schemas are equivalent [Beeri 79a, Fagin 79] or whether a schema embodies all the intended integrity assertions [Beeri 79b]. For integrity management, the technique of synthesizing run-time tests [Bernstein 80, Stonebraker 75] has been developed. A run-time test is such that if the test succeeds, the update guarded by the test will preserve the integrity assertions. All these advances in relational database technology have their origin in the elegant mathematical foundation of relational data model.

In contrast, virtually no design aids are available to the practitioners of network databases, despite the fact that many commercially available databases are based on the network data model. We believe that the lack of clean and mathematically amenable specifications for the network data model is the main reason behind this state of affairs. In order to solve the problems of schema design and integrity management for network databases, we therefore lay the groundwork by providing a formal specification of the network data model. Based upon these specifications, we then formulate methods to solve the preceding problems by techniques from logic and program verification. Finally, we concentrate on the problem of implementing the logic techniques effectively and efficiently. Thus, we use a methodology which integrates specification techniques from programming languages, program verification, and logic to solve the problems of databases. Moreover, our work gives an example of the state of the art in program verification.

In the following several sections, we present a brief overview of the issues involved and outline our solutions.

1.1 Overview of the Formal Specifications

A data model can be characterized by its structures, its constraints on the structures and its operations on the occurrences of the structures. A database schema is defined by providing a description of the structures and specifying the additional properties of these structures as constraints. A database transaction uses the schema definition and manipulates the occurrences of these structures. Thus, one can operationally view the process of defining schemas and transactions as one of declaring data types and writing procedures using these data types. In this operational viewpoint, which we take here, formal specification of a data model consists of defining a type-declaration language and a procedural language and specifying the formal semantics of these two languages. We therefore use a methodology consisting of applying specification techniques from programming languages to construct the formal specification of network data model.

Our methodology adapts two specification techniques from the programming language area: namely, the algebraic specification of abstract data types [Guttag 78] and the axiomatic specification of language semantics [Hoare 69]. We choose these techniques because we will be interested in proving properties of these languages. The earlier work Biller et. al. [Biller 76], in contrast, have used denotational method of specifying semantics with an aim of providing a precise guide to the implementors of the languages.

Briefly, we use the algebraic and axiomatic techniques as follows. First, we identify a set of primitive operations on network database structures and axiomatize them. The axiomatization is accomplished by algebraically specifying a few abstract data types. Second, we develop an assertion language to describe the properties of objects in a network database which are instances

of these abstract data types. The assertion language is a first order predicate calculus augmented by the functions and axioms of the data types. Third, we define a data manipulation language (DML). The formal semantics of the statements in this language are given as proof-rules in the axiomatic style of Hoare by using sentences of the assertion language as pre- and post conditions. Thus, the formal specifications for network databases consist of two parts: the assertion language with its proof-theory and the DML with its proof-rules. Of course, the definition of the abstract data types underly both components.

1.2 Methods for Solving Network Database Problems

Given the formal specifications, we can prove facts within the proof-theory of the assertion language. In developing methods to solve the problems of schema design and integrity maintenance, we formulate them in terms of proving facts in the assertion language.

To begin with, integrity assertions are expressed as formulas in the assertion language. To show a given set of integrity assertions in a schema to be inconsistent, one then must prove that the conjunction of these assertions implies "false".

To show that a given integrity assertion is redundant in a schema and therefore can be removed, one must prove that it is logically implied by the rest of the assertions defined in the schema. If one can determine whether or not an arbitrary assertion is redundant, then one can check each of the assertions successively for redundancy and remove them from the schema. The result of this process will be a schema which is optimal with respect to the number of assertions defined in it.

To determine if an update transaction will preserve an integrity assertion, we can apply program verification techniques to prove if the integrity assertion is an invariant across the transaction. The process of

program verification consists of the following three steps. First, one specifies assertions about what the program is intended to do. Second, the verifier applies the proof rules of the programming language statements to generate a finite set of formulas in the assertion language such that the program is consistent with its specifications if the formulas are valid in an appropriate model. Such formulas are called verification conditions (VCs). Finally, the validity of each VC is proved by using the proof-theory of the assertion language. It is important to note that we can easily incorporate the DML proof rules in an existing program verifier which can then generate the VCs. The formal specifications then allow us to use a program verifier as a tool for integrity maintenance.

By adapting and integrating the existing technologies, we thus obtain a formal specification of network data model. The specification in turn leads to the capabilities in solving the problems of schema design and integrity management to an extent comparable to those provided by the relational dependency theory. However, we advocate program verification as a compile time method of integrity management. This is different than the commonly accepted technique for relational databases, namely that of synthesizing run-time tests. This solution is appropriate for network databases because the transactions are commonly written by application programmers, as opposed to by end-users.

1.3 Implementing Mechanical Proof Procedures

Our methods of solving the problems of schema design and integrity management rely upon the ability to prove facts in the proof theory of the assertion language. The process of proving theorems can be implemented by the existing general purpose resolution based theorem provers, which are in fact embedded in traditional program verifiers. However, we may not always need the generality offered by the resolution based theorem provers. We therefore explore the possibility if one can take advantage of the specificity of

our application in building specialized theorem provers. In the following, we first explain two problems due to the generality of a traditional theorem prover: namely semi-decidability and inefficiency. We aim at overcoming these problems by opting for specialized theorem provers.

General purpose theorem provers, being based on first order predicate calculus, are semi-decidable. That is, they can only prove a theorem in finite time but are not guaranteed to terminate if the candidate theorem is not a theorem. The significance of this is that we would not be able to determine if a transaction is incorrect. The only solution to this problem is to restrict the assertion language such that the validity of a formula in the restricted assertion language can be proved or disproved.

The second problem with a general purpose theorem prover is related to its inefficiency. Traditionally, a theorem prover is built as a complete deduction engine which, when fed with the candidate theorem along with a set of axioms defining the intended model, tries to connect the axioms to the candidate theorem by a chain of inferences. This approach has led to only modest success at best. It seems that proofs of even simple formulas are too long and the proof space to be searched is too large for reasonable computing resources. It is thus appropriate to look for specialized proof procedures which can either reduce the length of a proof or size of the proof space or both.

There are two approaches to achieving efficiency, both of which put some restrictions on the applicability of the resulting theorem prover. In the first approach, one identifies some higher level properties which occur often among the commonly occurring theorems of interest in an application domain. These properties are then used as lemmas in order to prove a candidate theorem, rather than trying to derive it from the scratch. In that case, the proof of a candidate theorem may be obtained in a few steps if the lemmas are applicable [Good 82]. In the second approach, instead of directly considering a

problem domain, one can identify some class of logical formulas which makes the process of finding a derivation step simple. One can then examine properties of the domain as to whether they are expressible in the identified class. In our work here, we follow the second approach.

In our effort to take advantage of the specificity of the applications at hand, we identify a decidable subclass of the assertion language. The subclass consists of universally quantified Horn formulas. We also present an efficient decision procedure which can be used to prove the validity of a formula of the decidable subclass. By restricting ourselves to universally quantified formulas, we achieve the first goal - namely the decidability. The choice of limiting the formulas to Horn form is motivated by the second goal - namely the efficiency. In fact, the decision procedure that we develop here for the Horn clauses, has cubic time complexity in comparison to the exponential complexity of general purpose theorem proving.

The restricted class that we identified obviously has less expressive power than the full version of the assertion language. But a large number of integrity assertions of our interest in network databases fall under the restricted class. Also, in many instances of the problems of schema design and integrity management we encounter candidate theorems which belong to this restricted class, signifying their validity problem to be efficiently decidable. In other words, the decision procedure that we develop here can be used to implement the proposed methods for schema design and integrity management.

Notice that the use of the decision procedure complements the general purpose resolution based theorem proving. In fact, we are envisaging a theorem prover which consists of both the specialized decision procedure and the general purpose proof procedure. A candidate theorem fed to this composite theorem prover is directed either to the specialized decision procedure or to the general purpose one. In case the restriction required for the

decidable class do not apply, the general purpose theorem prover would be used. In sections 5.3 and 5.4 of chapter 5, we argue that the latter situation occurs infrequently in the context of the network database problems at hand.

1.4 Decidability Results and Program Verification

Apart from the applicability of the decidability results in implementing the methods of interest in network databases, these results constitute an important example of the state of the art in program verification. From the point of view of program verification, there are two aspects of our work which we wish to emphasize. First, we shall show that a large class of network database transactions, when annotated by assertions in the decidable class, are verification decidable. These transactions produce VCs which are all of the form $A \Rightarrow B$, where A and B are formulas in the decidable class of the assertion language. Therefore, such VCs can be proved or disproved by the decision procedure. Such standard form of VCs imply that the specialized theorem prover can use the same proof structure, thereby gaining efficiency. Moreover, the conditions on the transactions and the assertions under which such standard VCs are possible are readily recognized by simple inspection. Thus, if one were building a composite theorem prover which uses different proof procedure for different classes of theorems, such easy recognition of the applicability conditions is an important asset. In fact, the current trends in program verification systems advocate the use of such composite theorem provers.

Second, we use a reduction technique to identify the decidable class of our assertion language. The technique may be applicable to other cases. In studying verification decidability of annotated programs, one endeavors towards identifying a decidable subtheory of the data types used by the programs. For example, Suzuki and Jefferson [Suzuki 80] showed the verification decidability of sorting programs which include the array of integers as a data type. In our case, we have complex data types and it is not obvious

that we should have a decidable theory involving these data types. Our reduction technique transforms an assertion language involving these data types into a function-free quantificational language. The function-free quantificational language has been extensively studied by the logicians and many decidable subsets of it have been identified [Dreben 79]. The recognition of a decidable subset of the original assertion language becomes simple after transforming it into the function-free form. We believe this technique of reducing assertion languages to function-free quantificational calculus is attractive when a decidable subset of a given assertion language is to be identified.

1.5 Outline of the Chapters

To summarize the issues to be discussed, our goal is to construct a formal specification for network databases so that we can solve the schema design and integrity management problems. Chapters 2 and 3 are devoted to the development of the specifications. Chapters 4 and 5 will focus on the application of the specifications to the database problems. In chapter 5 we shall concentrate on developing the capability to infer formulas from other formulas in an algorithmic way.

The organization of the rest of this dissertation is as follows. In chapter 2, we shall present the formal specification methodology. Here we shall review the algebraic technique for specifying abstract data types and the axiomatic technique for specifying the semantics of programming languages. We shall apply these techniques to formally specify an object oriented language for database programming, which permits sharing of components among several composite objects.

In chapter 3, we apply the specification methodology to build the formal specifications for the Network data model. Here we extend the object oriented programming language from the previous chapter with the data types

specific to the network databases. The two components of the formal specifications, namely the data definition language logic and data manipulation language logic, are precisely defined here.

In chapter 4, we shall introduce the problems of schema design and monitoring integrity assertions for network databases and discuss how the DDL logic and the DML logic developed in chapter 3 can be applied to the solutions to these problems.

In chapter 5, we concentrate on achieving decidability. Here we shall identify the decidable subset of the assertion language of DDL logic and show that most of the important integrity assertions of interest are expressible in this subset. We shall then present the algorithm which can decide in finite time whether or not a formula in this subset is logically implied by a set of formulas from the subset. This algorithm is then used to provide solutions to the schema design problems. Also, we shall identify the class of transactions which can be verified effectively, without the problems associated with general purpose verifiers.

In chapter 6, we shall list the contributions of this dissertation and indicate directions for future research.

Chapter 2

Formal Specification Methodology

The main theme of this dissertation is the construction of an axiomatic basis for database programming languages based upon the Network Data Model. The construction requires a specification of the data objects encountered in Network Databases, a specification of the operations upon these objects and a specification of the semantics of a programming language based on these operations which manipulates these objects. This chapter introduces and illustrates specification techniques for data abstraction and language semantics and the issues in integrating these techniques into a verification methodology for the Network data manipulation language (DML) programs. The particular issue addressed in integrating the techniques is how to characterize precisely the notion of an data object. We shall develop a skeleton object-oriented language for database programming and its formal semantics, to illustrate the use of the specification techniques.

Various methods for specifying the semantics of a programming language have been proposed in the literature. As noted in [Hoare 74], these proposals follow two main directions: the constructive approach and the axiomatic approach. In the constructive approach, an abstract machine or an interpreter is defined and how the machine responds to the programs expressed in a core subset of the programming language is described. The semantics of the full version of the language is given by providing a translation of the language constructs to the those of the core subset. Examples of this approach are the semantics of LISP and Vienna Definition Language. In the axiomatic approach, a language is defined by making statements about the properties of

the programs, from which the user is able to deduce whether a program does what the user expects it to do. Examples of the axiomatic approach are Floyd-Hoare inductive assertion method [Hoare 69] and Dynamic Logic [Harrel 79]. While the constructive approach usually provides a good guide to the implemention of the language, the axiomatic approach enjoys the advantage of resulting in compact and representation-independent specifications. More importantly, the axiomatic approach gives a deductive theory, tailor-made for proving the correctness of programs written in the programming language.

As our major aim is to be able to prove the correctness of database programs, it is only appropriate that we follow the axiomatic method for defining the semantics of programming languages. Moreover, being concerned only with the input/output properties of programs, as opposed to their equivalence properties, we will adopt Hoare's deductive theory for a fragment of Pascal [Hoare 69] as our starting point.

As Hoare has shown in that pioneering paper, the deductive theory of Pascal programs dealing with integer data is based upon an axiomatization of the primitive arithmetic operations on integers. The programming language uses these primitive operations in forming the expressions of the language either to assign new values to the variables or to control the sequence of execution in branching or looping statements. The same primitive operations are used to form the terms of an assertion language in which one can express the properties of the integer variables at any state of computation. Finally, the axiomatization of these primitive operations form a part of the proof-theory in which the verification conditions are proved. Thus, the axiomatization of the primitive operations on integers plays an important role in the deductive theory of programs dealing with integers.

In a similar vein, our first step towards a deductive theory of Network Data Manipulation Language (DML) programs, which deal with data stored in

records and owner-coupled sets, will consist of identifying and axiomatizing the primitive operations on these structured objects. We can achieve such axiomatization via the algebraic specification of abstract data types, which characterize the behavior of the database objects under the primitive operations. With the definitions of the appropriate abstract data types, we can then augment our base programming language (fragment of Pascal with assignment, sequencing, branching and looping) and its deductive theory as follows:

- to obtain a DML by composing the primitive operations of the data types to form the database expressions and then using these expressions to assign new values to database objects (database updates) or to assign them to non-database variables representing data retrievals.
- to obtain a database assertion language for expressing the properties of database objects either in a particular state of a database or in all admissible states of a database.
- to obtain a proof-theory for databases by using the axioms of the data types.

The net result of these additions is then a deductive theory of programs in which database objects are manipulated. In fact, we shall use this strategy of constructing deductive theories of programs several times in this dissertation; the differences among these theories will be only in the specific abstract data types used.

The rest of this chapter is organized as follows. In section 2.1, we review the relevant concepts of the algebraic specification technique [Guttag 78] in the context of an example data type and illustrate how the defined primitive operations and their axiomatization are traditionally incorporated into the logic of our base programming language.

In section 2.2, we examine the notion of an object as used in the programming languages. There we argue that the programmers' idea of an

object is not synonymous with the notion of an instance of a data type as used by the algebraic specification technique. The facts that a composite database object would merely refer to the storage of its component objects, rather than copying their values, and that the components can be updated, lead us to the problem of specifying shared mutable objects. In the literature of data abstraction [Flon 79, Liskov 77], the algebraic specification technique has been criticized for its failure to resolve this problem. We propose a practical solution to this problem by using a combination of axiomatic techniques and algebraic techniques in specifying the semantics of data abstractions.

In section 2.3, we build upon the concepts developed in the section 2.2 to propose a skeleton programming language supporting object creation (database insertion), update and selection of objects based on their values. The semantics of this language is developed by first identifying and axiomatizing some primitive operations in the form of an abstract data type and then giving the semantics of the language statements in the axiomatic style of Hoare.

2.1 Algebraic Specification Technique

The main idea behind a data abstraction as used in the programming methodology is that the concept of a data structure can be defined by identifying a set of primitive operations on the data structure and then by defining the behavior of the data structure as observable under these operations. This behavioral abstraction then defines the commonality among a class of entities which are indistinguishable under the primitive operations. Consider a simple example where we want to define the concept of a LIST of integers. First we would consider what we want to do with such lists viewed as entities. Perhaps we want to create an empty list (NIL), to add an integer element to the list (INSERT), to delete the first element of the list (DELETE), to test whether an integer appears in the list (ISELEMENT), to extract the first integer element from a list (HEAD) or to compute the number of elements of a list (LENGTH). A behavioral abstraction of such list can now be given by defining the precise meanings of these primitive operations.

There are two possible approaches: the abstract model approach and the algebraic approach. In the abstract model approach, a sequence may be taken as an abstract representation of a list and the meaning of each operation on a list is expressed in terms of its effect on the sequence. The algebraic approach defines, instead, the properties of the operations by stating their relationships to one another. For example, we would state that the first element of a list just after an insertion is the element inserted. The advantages of the algebraic specification are the same as those enjoyed by the axiomatic approaches in general, namely ease of using them in proving properties of the entities under consideration, and a compact and representation-free definition. An algebraic specification of the abstract data type LIST is shown in figure 2-1. In the following, we are going to show how these specifications provide a behavioral abstraction of lists.

Before discussing the specifications, let us introduce some of the terminology of the algebraic specification technique to be used later throughout this section. In this technique, a data type is viewed as a heterogeneous algebra which is a pair <V,F>, V being a set of phyla and F being a set of finitary mappings. Each phylum is a set of all possible values of a given type. In particular, a distinguished phylum TOI stands for the set of possible values of the type of interest. Each function in F is a finitary mapping whose domains and range are in V. Specifically, some of the functions, called the constructors of the type being defined, results in a new value in TOI by acting on arguments from TOI and/ or from other phyla. There may be a zero-ary function among the constructors which is called a primitive constructor. The rest of the functions, called the observers of the type being defined, take at least one argument from TOI and results in a value in some phylum other than TOI. As only the properties of the extraneous phyla are assumed to be known, the observer functions provide the means of expressing the properties of the entities in TOI in terms of the known ones.

```
type LIST
operations
   NIL:          → LIST
   INSERT: LIST × INTEGER → LIST
   DELETE: LIST → LIST
   ISELEMENT: LIST × INTEGER → BOOLEAN
   HEAD: LIST → INTEGER
   LENGTH: LIST → INTEGER

axioms
L1. ISELEMENT(NIL,i) = false
L2. ISELEMENT(INSERT(l,i1),i2)
      = if EQ(i1,i2)
           then true
           else ISELEMENT(l,i2)

L3. DELETE(NIL) = NIL
L4. DELETE(INSERT(l,i)) = l

L5. HEAD(NIL) = undefined
L6. HEAD(INSERT(l,i)) = i

L7. LENGTH(NIL) = 0
L8. LENGTH(INSERT(l,i)) = LENGTH(l) + 1
```

Figure 2-1: Algebraic Specification of Type LIST

We now return to the example of LIST and make several observations in the light of the terminology defined above.

1. The specification consists of two parts: the syntax part and the axioms part. The syntax part defines the names, the domains and the ranges of the functions which are the primitive operations on lists. The axioms part defines the semantics of these functions implicitly by stating their relationships to one another.

2. Noting the syntax of the functions, the set of constructor functions is {NIL, INSERT, DELETE}. Among these, NIL is a primitive constructor. The reason they are called the constructors is that any

instance of a list can be inductively constructed from an empty list (result of NIL) by an application of a sequence of INSERTs. For example, the list [3 5 2] can be constructed as a result of the expression

INSERT(INSERT(INSERT(NIL,2),5),3)

As such, any list can canonically represented by such expressions.

3. The set of observer functions is {ISELEMENT, HEAD, LENGTH}. They are called observers because the property of a list can be observed by their applications on a list. For example, the fact that the list canonically represented as INSERT(NIL,2) has only one element, can be observed by applying the LENGTH function to it as follows:

$$\text{LENGTH(INSERT(NIL,2))} = \text{LENGTH(NIL)}+1 = 1$$

From this example, we see that we can talk about the values of a data type without ever mentioning its structure. Any value is equivalent to an expression involving only the constructor functions. The axioms of the specification are written in such a way that the effect of applying an observer function on such canonical expressions can be derived by reducing the expression to values of known types. Thus, the algebraic specifications provide an elegant way of abstracting the behavior of data entities.

By the way of illustration, we shall now consider how the specification of the data type LIST helps in obtaining a deductive theory of programs dealing with lists of integers. First of all, the data type specification introduces a set of function symbols which can composed to form the expressions of the language. The value of an expression can be then be assigned to variables. For example, we can update the value of a list variable l by adding a new element n as follows: l:=INSERT(l,n).

Second, let us consider how the semantics of such an assignment statement are given. The traditional interpretive model for Hoare's logic of programs [HOAR] has a notion of state which consists of a valuation

$\mathcal{V}$:{variables of language}→{values}.

The value of an expression e with free variables x1,...,xn is given as:

$$\mathcal{V}(e) = e[\mathcal{V}(x1)/x1,...,\mathcal{V}(xn)/xn]$$

The notation P[y/x] denotes the expression obtained from P by substituting y for all free occurrences of a variable x in P. An assignment statement is thought of as one changing states. Specifically, the assignment statement x:=e results in a new state with a new valuation $\mathcal{V}'$ which agrees with the old valuation except at the point x and $\mathcal{V}'(x)=\mathcal{V}(e)$. By definition of $\mathcal{V}(e)$, it follows that only values of the variables are copied around by the assignment statement.

We can assert properties of variables in particular state by a formula in an assertion language, which may be quantified first order calculus with the new terms formed by composition of functions defined on the data types. For example, in a particular state if 5 is the first element of list l, we can express that in this assertion language as: HEAD(l)=5. Alternatively, we can talk about general properties of lists as well. For example, the fact that the lists become shorter in length after deletion of elements can be stated as:

$(\forall l)(l \neq NIL \Rightarrow (LENGTH(DELETE(l)) < LENGTH(L)))$

The formula from the assertion language can be used to state the partial correctness semantics of language statements. For example, the semantics of the assignment statement is given by the following Hoare-formula:

$$P[e/x]\ \{x:=e\}\ P,$$

which states that if P is to be true after the assignment, then P[e/x] must be true before the statement.

Finally, for proving a program A with respect to the pre- and post-conditions P and Q respectively, one has to prove the Hoare-formula P{A}Q. Using the partial correctness formula (called the proof-rules), the proof is reduced to implicational formulas of the form P⇒Q which are called verification conditions(VCs). A VC is a formula in the assertion language,

involving the functions of the data types only. The axioms of the data types are used to establish the truth of the VCs.

2.2 Specifying Shared Mutable Objects

As a first step towards the semantics of an object manipulation language, we must make the notion of an object precise. We view an object as a piece of storage which has unique identity and holds a current value at any instant of time. It is also common to form composite objects out of simpler objects. Each composite object refers to the storage of its components and a component may be shared among several composite objects. In the jargon of programming languages, such components are called shared mutable objects [Liskov 77]. A linked list referring to the atoms as found in the implementation of the language LISP, is a an example of such composite objects. Any update on a shared mutable object has side-effects on all the composite objects encompassing it. The semantics of an object-manipulation language, like the one given in the next section, must therefore include a precise characterization of the update operations on shared mutable objects. This section presents such a precise characterization.

Viewing an object as a piece of storage is very similar to the way the variables in a programming language are treated. Thus, it is tempting to express an update operation on an object as an assignment statement of the programming language, where the variable denotes the object and the expression denotes the new value of the object as a result of the update. But in the usual interpretation, the assignment statement affects only a single variable and cannot express unbounded side-effects on other composite objects, as is desired in case the variable happens to denote a shared component object. If we want to stick to the side-effect-free interpretation of the assignment statement, we need to be able to limit the effect of updating a shared object only to itself.

We now propose an approach which captures the effect of updating shared mutable objects within the side-effect-free interpretation of the assignment statement. The overall idea is to treat the value of a composite object being composed of the identities of its components. An update operation on a component object which leaves the identity unaffected, need not then change the values of the composite objects. This idea is materialized by introducing the notion of object-variable in the programming language and considering in turn the mechanisms to update an object and to form a composite object through these object-variables.

The notion of object-variables is similar to the Pascal pointer variables. In providing the semantics of Pascal pointers, the authors of [Luckham 79] have used the notion of a reference class, which is considered to be unbounded set of objects. For each base type, there is a reference class and a pointer of a base type points to an element of the associated reference class. The semantics of an update through a pointer variable is viewed as changing the reference class. Thus, even if two pointers refer to the same object and the object is updated, the effect of the update is visible to the other pointer also, because it refers to the same reference class. Thus the notion of reference class solves the problem of aliasing. We shall adopt similar mechanism for dealing with object-variables.

An object-variable denotes the identity of an object. Its state S is given by a pair $<\mathcal{A},\mathcal{V}>$, where the variable assignment $\mathcal{A}$ is a one-one partial map

$$\mathcal{A}\text{: \{object-variables\}} \rightarrow \text{\{object-identities\}}$$

and for a given type T of objects, the type state $\mathcal{V}$ is a total map

$$\mathcal{V}\text{: \{object-identities of type T\}} \rightarrow \text{\{values of type T\}}.$$

The current-value of an object denoted by an object-variable x is then $\mathcal{V}(\mathcal{A}(x))$, while its identity is $\mathcal{A}(x)$. The mechanism for updating the object denoted by x with a new value e is the update statement $x \Leftarrow e$. The interpretation of this

update statement is that if initial state is $S=<\mathcal{A},\mathcal{V}>$, we get a new state $S'=<\mathcal{A}',\mathcal{V}'>$ where

$$\mathcal{A}'=\mathcal{A}, \text{ and}$$
$$\mathcal{V}'(y) = \mathcal{V}(y) \text{ if } y\neq\mathcal{A}'(x)$$
$$= e \text{ if } y=\mathcal{A}'(x)$$

Thus, the update through an object-variable affects the type-state, but only for the object denoted by the variable and the new type-state reflects the new value e of this object.

The mechanism for forming a composite object denoted by variable y out of the component objects denoted by the variables $x_1,\ldots,x_n$ is again a update statement of the form $y\Leftarrow e(x_1,\ldots,x_n)$.The interpretation is that in the new state $S'=<\mathcal{A}',\mathcal{V}'>$, we have

$$\mathcal{A}' = \mathcal{A}, \text{ and}$$
$$\mathcal{V}'(z) = e(\mathcal{A}(x_1),\ldots,\mathcal{A}(x_n)) \text{ if } z=\mathcal{A}'(y)$$
$$= \mathcal{V}(z) \text{ if } z\neq\mathcal{A}'(y)$$

That is, only the variable assignment component is used for evaluating the object-variables appearing in an expression. Thus, the new value of y is composed of the identities of the component objects denoted by $x_1,\ldots,x_n$.

In the preceding interpretive model of object manipulation, we observe that the type-state provides the essential association of object identities to their current values. Updating an object of a given type changes the respective type-state and the value of an object can be retrieved as its image under the type-state. Thus, we can characterize the properties of type-states by identifying the following two functions as primitive operations:

CHANGE($\mathcal{V}$,i,e)::
changes the type-state $\mathcal{V}$ by associating a new value e with the object identity i.

VAL($\mathcal{V}$,i)::
retrieves the value associated with an object identity in a type-state $\mathcal{V}$.

An axiomatization of type-states can be expressed by the following relation between CHANGE and VAL:

$$\begin{aligned} &VAL(CHANGE(\mathcal{V}, i_1, e), i_2)) \\ &\quad = e \quad \text{if } i_1 = i_2 \\ &\quad = VAL(\mathcal{V}, i_2) \quad \text{otherwise.} \end{aligned}$$

We can now state the semantics of updating an object, expressed as an update x⇐e, in the axiomatic style of Hoare as follows:

(Axiom of object update)

$$P[CHANGE(\mathcal{V},x,e)/\mathcal{V}] \ \{x \Leftarrow e\} \ P$$

The above treatment of providing the precise semantics of updating shared mutable object has a wider applicability in the context of data abstraction methodology in general. There the algebraic specification technique has been criticized [Flon 79, Liskov 77] because the constructor functions cannot capture the effect of in-place update operations on objects; they merely produce a new value rather than modifying an already existing object. Philosophically, if a data type specification has to provide a behavioral abstraction of a class of objects, and as the behavior of composite objects is affected by the updates on the component objects, the behavioral abstraction of composite objects has to reflect such effects. The algebraic specification technique by itself cannot provide such behavioral abstractions. Our treatment of the semantics of updating shared mutable objects lends itself as a new and practical method for specifying the constructor operations of a data abstraction. In the following section 2.2.1, we first illustrate the method via an example and then in section 2.2.2, we summarize and contrast the method with previously known ones.

2.2.1 An Example of Mutable Data Abstraction

Suppose we want to specify a behavioral abstraction of lists of atoms as are found in typical implementation of LISP. In particular, the situation that we wish to capture, is shown in figure 2-2. There, we have a list object denoted by a variable l and an atom denoted by the variable n. In the initial state, the first element of the list is the atom denoted by n and the value of the atom is 5. After we update the value of this component atom by incrementing it, we want to reach a final state where the updated value of the atom as retrieved by accessing the first element of the list is the integer 6.

To specify abstractly the values of list, we first define algebraically a data type LISTVAL, shown in figure 2-3. The meanings of the functions are very similar to those used in the example in section 2.1 except for use of the atom-identities (ATOMID) in composing the list-values. This is in accordance with our notion that the value of a composite object consists of identities of its component. The abstraction of the atom-values, on the other hand, is simply given by the data type INTEGER whose specification is assumed here to be known. Specifically, the axiomatization of a constructor function SUCC to increment an integer is assumed.

Now we turn our attention to adding the state-transition semantics of constructor operations which are not captured in the algebraic specifications of data types LISTVAL and INTEGER. For the constructor operation on atoms corresponding to the function SUCC, we shall introduce a procedure $SUCC_p$ in the programming language and specify its semantics by translating it to our update statement through object-variables as follows:

$SUCC_p$(n):: n ⇐ SUCC(VAL(D#ATOM,n))

The symbol D#ATOM denotes the current type-state for object type ATOM and the function VAL is as defined previously to retrieve the current value of object denoted by a variable. The above definition of the procedure $SUCC_p$ captures our intention that it increments the value of an atom denoted by n

without affecting the object identity, under our interpretation of the update statement. Similarly, the state-transition semantics of the constructor operation INSERT on lists is expressed through the following definition of a procedure $INSERT_p$:

$INSERT_p(l,n)$ :: $l \Leftarrow INSERT(VAL(D\#LIST,l),n)$

The symbol D#LIST denotes the current type-state for the object type LIST.

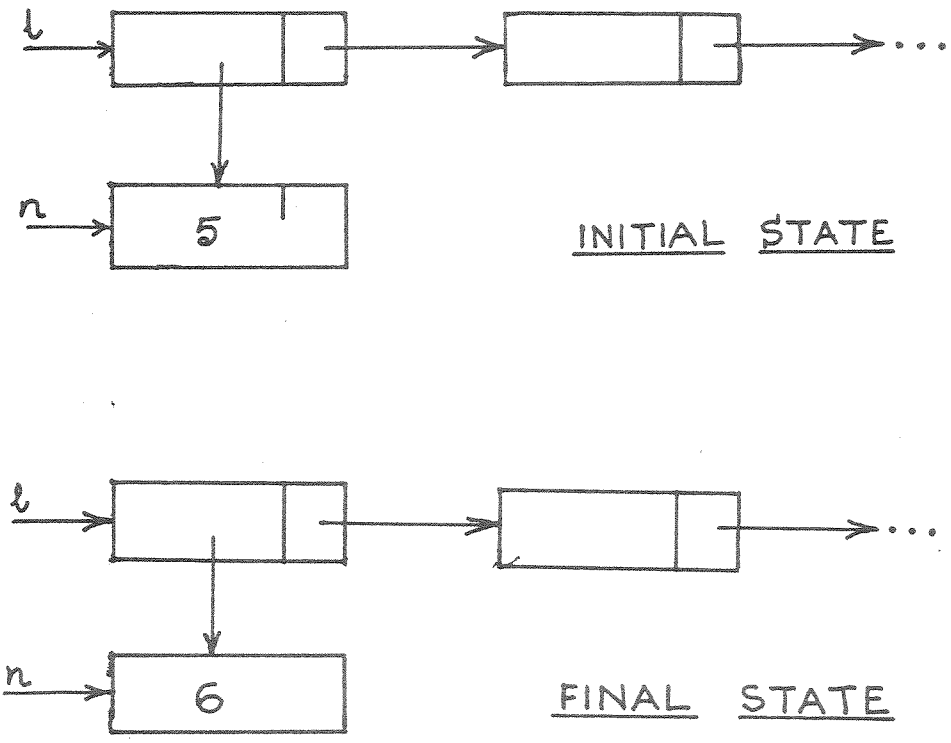

Figure 2-2: An Example of Sharing

Coming back to the situation shown in figure 2-2, we can now write a program A to increment the atom n as follows:

A:: $SUCC_p(n)$

The initial and the desired final conditions of this one-line program, as shown in figure 2-2, are asserted by the formulas P and Q as follows:

```
type LISTVAL
operations
   NIL:    → LISTVAL
   INSERT: LIST × ATOMID → LISTVAL
   HEAD:   LIST  → ATOMID
axioms
L1. HEAD(NIL) = undefined
L2. HEAD(INSERT(l,n)) = n
```

Figure 2-3: Data Type LISTVAL

```
     P:: HEAD(VAL(D#LIST,l))=n ∧ VAL(D#ATOM,n)=5
and, Q:: VAL(D#ATOM, HEAD(VAL(D#LIST,l)))=6
```

We now prove that indeed the program A achieves the desired effect i.e., we prove the formula P{A}Q. By using the definition of $SUCC_p$ and the axiom of object update, we get:

```
  Q':: VAL(CHANGE(D#ATOM,n,SUCC(VAL(D#ATOM,n))),
           HEAD(VAL(D#LIST,l)))=6
```

So, by the rule of consequence, we are left to prove P⇒Q'. By assuming the antecedent P, we get:

```
  Q'⇔ VAL(CHANGE(D#ATOM,n,SUCC(5)),n)) = 6
    ⇔ SUCC(5) = 6 by the axiomatization of type-state
    ⇔ true        provable in the assumed theory of INTEGER
```

Thus, the specifications of the data types INTEGER and LISTVAL, together with the axiomatization of type-states and the axiom of object update are adequate for proving the effect of updating a shared mutable object.

2.2.2 Discussion of the method

We summarize here the method of specifying data abstractions involving shared mutable objects, as illustrated in the previous section 2.2.1. We then contrast this method to that of Berzins [Berzins 79], which is the only other known attempt to specify such data abstractions.

The first step of our method is to specify an axiomatization of the primitive operations purely in terms of the functional abstractions provided by the algebraic specification technique. A data type thus specified, is intended to provide a behavioral abstraction of only the values of the objects of the data type. The next step of the method specifies the state-transition semantics desired for the constructor operations of the mutable data abstraction. This is achieved separately by introducing an equivalent procedure-call statement, one for each constructor operation, in the programming language; the semantics for the procedure-call statement is given by our axiom of object update. We are, however, assuming the availability of an axiomatization of type-state functions CHANGE and VAL. But such axiomatization is easily definable in the realm of algebraic specification technique. Thus, in our method, the specification of a mutable data abstraction is based upon a combination of algebraic specification techniques and Hoare's axiomatic techniques.

The advantage of our specification method is that it is a practical approach as far as proving the correctness of usage of data abstraction is concerned. We can recycle the verification techniques based on Hoare's logic of programs and the theorem provers based on the equational reasoning of algebraic specification technique to reasoning about shared mutable objects. We need not develop a new proof-theory. In contrast, the earlier work of Berzins, though using the same concepts of type-states, incorporated the state-transition semantics directly into the specification of the data types. His specification technique required an implicit parameter, representing a union of all type states, to every operation on the data types. This requirement leads to undue difficulties in the verification of the usage of the data types in programs because one has to invent extra symbols to denote the implicit parameter, one for each statement of the program under consideration. In fact, his work concentrated on verifying the correctness of implementation of data types, rather than on the verification of their usage in programs.

Our treatment of object-variables uses the same notions as used in in verification of Pascal programs with pointer variables in [Luckham 79]. The concept of type-state is essentially the same as their reference-class. However, our motivation to define shared mutable objects is different from theirs and as such solves an important problem in data abstraction methodology.

2.3 An Object-Oriented Language for Database Programming

We view a database as a collection of typed objects. The object types defined in the database schema characterize the domains of values that these objects can store. A data manipulation language operates on these objects. The common operations in data manipulation language are creating and deleting objects, updating and retrieving values from objects and finding objects according to some value-based search criteria. In this section, we shall develop a skeleton programming language (and its semantics) permitting such object-manipulation operations. In the light of the specification method introduced in the previous section, we shall here assume that the data types characterizing the values of objects are defined elsewhere and hence concentrate on only the object manipulation operations.

In the previous section, we have considered updating and retrieving values from existing objects. We defined the notion of type-states and identified two functions CHANGE and VAL on type-states as the basis of expressing the semantics of update and value-retrieval operations on objects which are denoted by object-variables. In this section, we shall identify some basic operations for creating, deleting and performing value-based search. As we shall see, these basic operations can be defined as functions on type-states and these functions are then used to define the semantics of the statements in the programming language. For the rest of this section, we shall consider in turn the processes of creation and deletion in section 2.3.1 and of value-based search in section 2.3.2. The complete specification will be collected in the figures 2-4 and 2-5.

2.3.1 Creation and Deletion of Objects

When an object of a given type is created, it is given a unique identity which is different from the identities of all the previously created objects of the same type. The value of a fresh object is defined to be some distinguished null value $\perp$ which is assumed to be in the domain of values of each type. Moreover, an object-variable is bound to this fresh object.

If we introduce the procedure-call CREATE(T,t) in the programming language, to create an object of type T and to bind it to the object-variable t, the statement is interpreted precisely as follows: Let $S=<\mathcal{A},\mathcal{V}>$ and $S'=<\mathcal{A}',\mathcal{V}'>$ be the initial and final states. Then,

$\mathcal{A}'(x)=\mathcal{A}(x)$ if $x\neq t$
$\quad =i$ if $x=t$, where i is a identity not used before.
and,
$\mathcal{V}'(z)=\mathcal{V}(z)\cup\{<i,\perp>\}$

In other words, object-creation changes the variable assignment at the point t, but also extends the type-state by adding a new pair associating the fresh identity i with the special null value $\perp$. We also need to make sure the identity i was never used before.

To extend a type-state and to test whether an identity is already in the domain of a type-state, we define the following two functions:

EXTEND($\mathcal{V}$,i)::
adds a pair $<i,\perp>$ to the type-state $\mathcal{V}$.

FRESH($\mathcal{V}$,i)::
is true if the identity i is never used before and is thus not included in the domain of type-state $\mathcal{V}$.

The axiomatization of the function FRESH and EXTEND is included in figure 2-4 where a complete specification of type-states is given an algebraically specified data type TYPESTATE which is parameterized by object types.

Now, we can state axiomatic semantics of the statement CREATE(T,t) as follows:

$(FRESH(\mathcal{V},i) \Rightarrow P[\mathcal{V}'/\mathcal{V}][i/t])$ {CREATE(T,t)} P
where $\mathcal{V}' \Leftrightarrow$ EXTEND($\mathcal{V}$,i) and i does not appear in P.

Intuitively, the axiom states that in order for $P(\mathcal{V},t)$ to be true after creation, if the object identity i was never used before, then $P(\mathcal{V}',i)$ must be true before creation.

Considering the delete operation, let DELETE(T,t) be the language statement to delete an object denoted by variable t of type T. We assume that this statement deletes only one object, as opposed to any propagation of deletion. In particular, deleting a composite object still leaves its component objects unaffected. In order to avoid reusing the identity of a deleted object, the deletion operation does not destroy the object identity. It merely updates the value of the object to a special error-value undefined. Like the value $\perp$, the value undefined is also included in the domains of all object values. As stated in the background section 2.1, there is an implicit axiom which states any function operating on this error value will also result in the error-value. This implicit axiom will therefore make any use of the deleted object illegal. As we have already got the mechanism for updating an object, the semantics of DELETE(T,t) is given by the following definition:

DELETE(T,t) $\Leftrightarrow$ t$\Leftarrow$undefined

This results in the proof-rule:

P[CHANGE($\mathcal{V}$,t,undefined)/$\mathcal{V}$] {DELETE(T,t)} P

2.3.2 Value-based Search

The value-based search operation on database objects specifies a property Q as a boolean expression called a qualification expression. More precisely, a qualification expression is a formula in the assertion language with exactly one free variable. The result of a value-based search is a set of object-identities such that each identity when substituted for the free variable in the qualification expression, makes the expression true. Necessarily, the terms in a

qualification expression involve the observer functions of the data types used to specify the object-values. We shall therefore defer the detailed examples of qualification expression until we arrive at a complete specification of a Network DML in chapter 3.

Again, we observe that the values of objects of a given type are available as images of their identities under associated type-state. Thus, we can define a basic function FIND($\mathcal{V}$,Q(x)) on a type-state $\mathcal{V}$ to select a set R of object identities where R={i|Q[i/x] is true}.

The axiomatization of this FIND function deserves some attention. First of all, the special value $\perp$ never satisfies a qualification expression and therefore, the freshly created object cannot be selected by FIND. This leads us to the following relation:

FIND(EXTEND($\mathcal{V}$,i),Q(x)) = FIND($\mathcal{V}$,Q(x))

Second, if we update an object with a new value and the new value satisfies the qualification expression, then the identity of this object should be included in the selected set of identities. This is captured by the following relation:

```
FIND(CHANGE(V,i,e),Q(x))
   = if Q(i)
        then {i}∪FIND(V,Q(x))
        else FIND(V,Q(x))
```

The result of FIND can be assigned to a set-valued variable s to get a retrieval statement as s:=FIND($\mathcal{V}$,Q(x)). Set-valued variables are treated as normal variables, as opposed to object-variables. Therefore, the usual axiom of assignment can be used to specify the semantics of a retrieval statement.

To summarize, we have introduced the following functions on type-states: EXTEND, CHANGE, VAL, FRESH, and FIND. They serve as a basis for defining the semantics of creation, update, deletion and value-based search statements in our object-oriented language. The functions on type-states are axiomatized by a parameterized abstract data type TYPESTATE[T], where an

object-type can be given as parameter. For the sake of completeness, a primitive constructor INIT is introduced, with the intention that it produces an empty type-state in which no object-identity is ever used. The complete specification of TYPESTATE[T] is shown in figure 2-4. The syntax and semantics of the language are also collected in figure 2-5 for easy reference. In the next chapter, we shall use this language as a starting point towards a DML for Network Databases.

```
type TYPESTATE [T]    (* instances written as D#T *)
requires
    T:{object-types defined in database schema}
    ID: indexed set of object identities,
        index being object-type
    OBJ: indexed set of object-value,
        index being object-type
    LISTID: {set of object-identities}
    QUAL:  {qualification expressions}
operations
    INIT: T →TYPESTATE
    EXTEND: TYPESTATE × ID →TYPESTATE
    CHANGE: TYPESTATE × ID × →TYPESTATE
    VAL:    TYPESTATE × ID →OBJ
    FIND:   TYPESTATE × QUAL →LISTID
    FRESH: TYPESTATE × ID →BOOLEAN
axioms
d1. VAL(INIT(T),i) = undefined
d2. VAL(EXTEND(D#T,i1),i2)
      = if i1=i2
           then ⊥
           else VAL(D#T,i2)
d3. VAL(CHANGE(D#T,i1,o),i2)
     = if i1=i2
          then o
          else  VAL(D#T,i2)

d4. FIND(INIT(T),Q) = {}
d5. FIND(EXTEND(D#T,i),Q) = FIND(D#T,Q)
d6. FIND(CHANGE(D#T,i,o),Q)
       = if Q(i)
            then {i}∪FIND(D#T,Q)
            else FIND(D#T,Q)

d7. FRESH(INIT(T),i) = true
d8. FRESH(EXTEND(D#T,i1),i2)
      =if i1=i2
          then false
          else FRESH(D#T,i2)
d9. FRESH(CHANGE(D#T,i1,o),i2)) = FRESH(D#T,i2)
d10. (* axiom of equality *)
     D#T=D#T' iff (∀i)((FRESH(D#T,i)=FRESH(D#T',i)
                      ∧ (VAL(D#T,i)=VAL(D#T',i)))
```

Figure 2-4: Axiomatization of TYPESTATE

(axiom of object creation)
(FRESH($\mathcal{V}$,i)$\Rightarrow$P[$\mathcal{V}$'/$\mathcal{V}$][i/t]) {CREATE(T,t)} P
where $\mathcal{V}$'$\Leftrightarrow$EXTEND($\mathcal{V}$,i) and i does not appear in P.

(axiom of object update)
P[CHANGE($\mathcal{V}$,x,e)/x] {x$\Leftarrow$e} P

(axiom of object deletion)
P[CHANGE($\mathcal{V}$,t,undefined)/t] {DELETE(T,t)} P

Figure 2-5: Semantics of Object-manipulation Statements

Chapter 3

A Logic for Network Databases

We stated in the introductory chapter that verifying transactions satisfy integrity constraints is an important problem. In this chapter, we provide a logic for Network Databases which permits us to establish, among other other properties, that a transaction does or does not preserve the integrity constraints. There are two essential components of this logic corresponding to the two languages used for database programming: the data definition language (DDL) and the data manipulation language (DML). The DDL logic provides the notations, for describing Network database structures and for expressing integrity constraints, and a proof-theory which can be used to prove properties of stored data in a Network database. The DML logic provides a Network DML to write the transactions, a way of expressing their correctness properties as annotations to the transactions and a proof-theory which, in conjunction with the DDL logic, permits us to prove the correctness of transactions. Here we shall not only define the formal systems in complete detail, but also illustrate their use through examples.

The construction of DDL and DML logics utilizes the specification techniques and follows the methodology, introduced in the previous chapter. From the previous chapter, we take the object-manipulation language and its axiomatic basis as our starting point here and propose several refinements to them in order to get the DDL and DML logics. First of all, we identify a set of primitive functions on Network-structured values - values of records and owner-coupled sets (OCS), and axiomatize them as two algebraically specified abstract data types RECORD and OCS. Second, we include these data type

specifications, together with the specification of the data type TYPESTATE, into a many-sorted first order calculus to obtain an assertion language and its proof-theory. The assertion language and its proof-theory serves a DDL logic. Third, we extend the object-manipulation language of previous chapter to obtain a Network DML. The particular extensions are for the object-creation statements where the created objects are given initial values corresponding to either empty-record or empty-OCS (OCS with only its owner record). Specific statements for updating Network objects, such as CONNECT and DISCONNECT, are introduced as variants of object-update statement, where the updated value is an expression involving the constructor functions defined on the data types RECORD and OCS. Finally, the proof-rules for the Network DML statements are given using the specific forms of the axiom of object-update and therefore, uses the functions of the new data types as well. The pre- and post-conditions of these proof-rules are formulas in the assertion language. Thus, the DML, its proof-rules and the DDL logic together form the DML logic.

Relatively little work has been done on DML logics [Casanova 80, Gardarin 79]. Though these efforts were similar to ours in spirit, they considered only relational data model. Our work deals with the Network data model which has certain important differences from the relational data model. First, in the Network model, in absence of integrity constraints to the contrary, several records of the same type can coexist with the same data values. In order to distinguish between records with duplicate contents, we need to use a different concept of equality, other than the usual value-equality. In our case, the equality is based on object-identities, rather than their values. Second, records can participate in OCS relationships, which may be independent of data values of these records. In other words, the membership in an OCS itself is information bearing over and above the value-based relationships. Thus, selection criteria for objects has to include OCS membership. We introduce a binary-valued function OWNS for the data type OCS to test the membership.

This function can be used in the qualification expressions for finding objects. Third, the members of an OCS are ordered. In our case, the ordering is based on values of the member records unlike FIRST and LAST insertion clauses of CODASYL DDL. In any case, the ordering leads to navigational access i.e., record-at-a-time processing, in contrast with the set-retrieval of relational databases. We provide primitives functions, POS and GET, on the data type OCS for navigation within an OCS. As we shall see, our formulation of navigational primitives avoids the notion of implicit "currency" pointers and leads to simpler proof-rules for individual DML statements.

Apart from serving as a basis for our DDL and DML logics, the axiomatized primitive functions on the data types RECORD and OCS, provide a clean and simple abstraction of the Network Data Model and its navigational operations. As such, they can be used to define interpretive semantics of existing navigational languages based on Network model, such as UDL [Date 80] and CODASYL DML [Codasyl 71]. Though our abstraction has left out some of the implementation-oriented features of CODASYL DML such as AREA constructs etc., any exercise in providing the formal semantics of existing languages may reveal inconsistencies which go undetected in their natural-language based descriptions. Also, precise semantics of the existing languages will make it possible to verify the correctness of the programs written in them.

From the point of view of database language design, our DML illustrates some of the important concepts, although it is rudimentary and lacks in many engineering aspects. First, it illustrates direct embedding of Network data types in the control structures of Pascal, as is advocated by recent database language proposals [Date 80, Wasserman 79, Schmidt 77]. Rather than interfacing the programming language to the database through I/O areas, our DML includes the database as a part of the program's memory space, allowing the object-variables to be used directly in the control constructs of the

language and reducing the complexity of the definition of the language semantics. Second, our DML makes explicit use of object-variables to manipulate database objects, instead of relying upon implicit currency pointers of CODASYL DML. As different statements of the language do not interact through the currency pointers (see [Biller 76] for such interaction between FIND NEXT and DELETE operations in CODASYL DML), we enjoy the independence of individual DML statements and can give their proof-rule easily. In fact, our design of the DML goes hand-in-hand with the consideration of the proof-rules, which is always a healthy practice.

The rest of the chapter is organized as follows. In section 3.1, we define the data types RECORD and OCS so as to provide a simple behavioral abstraction of the Network Data Model. To explore the capabilities of the identified primitive operations, we take some examples from other languages based on the Network Model and give their interpretive semantics in terms of these primitives. In section 3.2, we integrate the specifications of the data types into a many-sorted first order language to obtain an assertion language. We show how this language is capable of describing database structures for Network Model and is therefore a candidate as a DDL. In section 3.3, we define our DML and give its proof-rules by using formulas from the assertion language as the pre- and post-conditions of the language statements. Here we indicate how one may use these proof-rules to prove the correctness of programs written in this DML.

3.1 Network Structured Values

A traditional network schema $\underline{N}$ describes two sets of types, namely, the set of record types $\underline{T}$, and the set of owner-coupled set (OCS) types $\underline{L}$. A network database, which is an extension of $\underline{N}$, stores occurrences of these record types and OCS types. In terms of our object-oriented model, a database is a set of objects, where the set of object types $\underline{S}$ is the union of $\underline{L}$ and $\underline{T}$. A database state consists of a union of all the type-states, one type-state for each object type.

In this section, we shall be concerned only with characterizing the values of the records and OCS-s. The association of these values to the object identities is accomplished by the type-state for each object-type. In following, we define the primitive operations on records and OCS-s and axiomatize them by two parameterized abstract data types, RECORD[T] and OCS[L] respectively. After the introduction of these data types, we illustrate in subsection 3.1.3 that these primitive functions provide a useful abstraction of Network data model and its operations.

3.1.1 Axiomatization of Record Values

A record occurrence of type $T \in \underline{T}$ is a record valued object (record) whose value is a k-tuple <a1,a2,..,ak>, where each ai $\in$ DOM(Fi), Fi is a field of record type T, and DOM(Fi) is the domain of values associated with field Fi.

```
type  RECORD[T]
requires
     T: set of record types
     F: set of field types
     D: set of domains
operations
     EMPTY: T                  → RECORD
     WRITE: RECORD × F × D → RECORD
     READ:  RECORD × F  → D
axioms
r1. READ(EMPTY(T),F,d) = ⊥ (* each domain has the special
                              value ⊥ *)
r2. READ(WRITE(r,F1,d),F2)
    = if F1=F2
         then d
         else READ(r,F2)
r3. (* axiom of record value-equality *)
    r1 =record r2
            iff (∀F∈F) (READ(r1,F)=READ(r2,F))
```

Figure 3-1: Specification of Data Type RECORD

We define the following functions on record-values which are axiomatized as a data type RECORD[T], shown in figure 3-1.

EMPTY(T):: creates an empty record-value with the value $\perp$ for each of its fields.

WRITE(r,F,v):: stores value v in field F in record-value r.

READ(r,F):: extracts the value of field F from record-value r.

The equality of two record values is based on the fieldwise equality of the constituent values, as shown in axiom r3.

3.1.2 Axiomatization of OCS-values

An OCS occurrence of type $L \in \underline{L}$ is an OCS-valued (OCS) object whose value is a 2-tuple <t,m>, where t is the identifier of the owner record and m is an ordered set of identifiers of the member records. Thus, an OCS references both its owner record as well as the constituent member records. As a result, the effect of an in-place update of any referenced member record is visible on subsequent navigational retrievals via the referencing OCS.

We define the following primitive functions on OCS values which are axiomatized as the data type OCS[L], shown in figure 3-2.

MAKE(L,r):: creates an empty OCS-value of type L, with record r as its owner.

ADD(s,r):: add record r as a member to OCS-value s.

OWNER(s):: retrieves the identifier of owner record from an OCS-value s.

OWNS(s,i):: is true if record i is a member of OCS-value s.

POS(s,r,Q):: retrieves the position number of record r within the set of members in OCS-value s; the binary predicate Q(x,y) determines the ordering of the set of members.

```
type OCS[L]    (* written as s *)
requires
     RID: set of record identifiers
     T: set of record types
operations
     CREATE: L X RID  → OCS
     ADD:    OCS X RID    → OCS
     REMOVE: OCS X RID    → OCS
     OWNER:   OCS      → RID
     OWNS:   OCS X RID → BOOLEAN
     MEMBERS: OCS  → {RID}
     POS:    OCS X RID X ORDER →  INTEGER
     GET:    OCS X INTEGER X ORDER → RID
axioms
o1. OWNER(MAKE(L,r)) = r
o2. OWNER(ADD(s,r)) = OWNER(s)
o3. OWNS(MAKE(L,r1),r2) = false
o4. OWNS(ADD(s,r1),r2)
    = if r1=r2  then true
                else OWNS(s,r2)
o5. POS(MAKE(L,r1),r2,Q) = 0
o6. POS(ADD(s,r1),r2,Q)
    = if Q(r2,r1)  then POS(s,r2,Q)
                   else 1 + POS(s,r2,Q)
o7. GET(MAKE(L,r),n,Q) = nil
o8. GET(ADD(s,r),n,Q)
    = if n=POS(ADD(s,r),r,Q)
         then r
         else if n > POS(ADD(s,r),r,Q)
                then GET(s,n-1,Q)
                else GET(s,n,Q)

o9.  MEMBERS(MAKE(L,r)) = {}
o10. MEMBERS(ADD(s,r)) = {r} U MEMBERS(s)

o11. REMOVE(MAKE(L,r1),r2) = MAKE(L,r1)
o12. REMOVE(ADD(s,r1),r2)
     = if r1=r2
          then s
          else ADD(REMOVE(s,r2),r1)
o13. (* axiom of OCS value-equality *)
     s1 =ocs s2
         iff (owner(s1)=owner(s2))
             ∧ ((∀Q)(∀r)(OWNS(s1,r)⇒
                         OWNS(s2,r)∧(POS(s1,r,Q)
                                     =POS(s2,r,q)))
```

Figure 3-2: Specification of Data Type OCS

GET(s,n,Q):: retrieves the identifier of the member record which is in the nth position within the set of members in OCS-value s. The ordering is determined by binary predicate Q(x,y).

REMOVE(s,r):: removes a member record with identifier r from OCS-value s.

MEMBERS(s):: retrieves the set of identifiers of the member records from OCS-value s.

The equality of two OCS values, as shown in axiom o13, states that the owner records must be equal and every member of one OCS must be in the same position in the other under all ordering.

Navigational access within an OCS-value is supported by the two operations POS and GET. Based on the ordering predicate Q, function POS maps the identifiers of the member records to their ordinal numbers 1,..,N. Function GET performs the inverse mapping. The **ordering predicate** Q is a binary predicate defining the ordering of the member records within an OCS. For example, if employee records (EMP) under an OCS named DE (figure 3-3) are ordered in descending order of the salary field (SAL), the ordering predicate Q that relates two employee records e1 and e2, is

$$Q(e1,e2) \Leftrightarrow READ(VAL(D\#EMP,e1),SAL) > READ(VAL(D\#EMP,e2),SAL)$$

The axiomatization of the functions POS and GET deserve some attention. The basic idea is that the ordinal number of a member record may either be incremented by one as a result of adding another member record if it is "after" the newly added member according to the ordering predicate; otherwise, its ordinal number remains the same as before the addition. The ordering predicate Q(r2,r1) is true if r2 is "before" r1. This is directly expressed by axiom o6. Notice that, the axioms for POS are **well-grounded**, because each use of POS in the right-hand side of these two axioms uses a smaller instance of OCS-value as argument; s being smaller than ADD(s,r). Thus, the recursion of POS always terminates. On the other hand, the axiom o8 for GET is uses the

same size OCS value as argument to POS as the argument to GET on the lefthand side, though each recursive use of GET uses strictly smaller argument. Though a little complicated, the axiom is still well-grounded because recursion on POS always terminates.

3.1.3 Interpretive Semantics of Existing Languages

We believe that the definitions of these three data types TYPESTATE, RECORD and OCS provide a simple abstraction of the structures and primitive operations of network databases. Here we substantiate this claim informally by illustrating how these functions can be used to give the interpretive semantics of some of the common retrieval operations found in existing Network DMLs. The examples are based on a database schema shown diagramatically in figure 3-3.

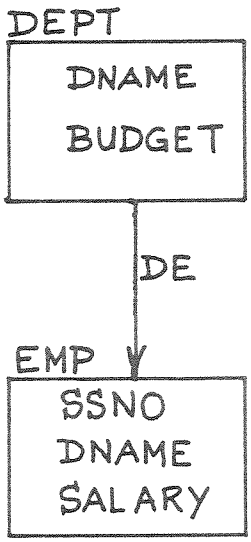

Figure 3-3: Bachman Diagram of An Example Schema

Example 1. Setting cursor variable [Date 80]

An employee record under a given department may be retrieved in UDL by the statement:

```
FIND FIRST EMP
   UNDER UNIQUE DEPT VIA DE
   WHERE DEPT.DNAME = 'research'  SET (e)
```

Here the UNDER clause specifies an OCS from which the first member record is to be selected. In our formulation, we shall encode this specification in a qualification expression as:

```
P:: (VAR x: D#DEPT[OWNER(D#DE[x])].DNAME
                                = 'research')
```

The semantics of the UDL statement is then given as:

```
de := FIRST(FIND(D#DE,P));
e := GET(VAL(D#DE,de),1,Q)
```

where the ordering predicate Q for the OCS DE is as shown in the previous subsection as an example. Note that the cursor variable e of the UDL statement is represented by a corresponding object-variable of type EMP record in the object-oriented language introduced in chapter 2. However, a cursor variable is implemented in UDL as having pointers to the selected record as well as to the selected ordered set. In our formulation, the cursor variable e must thus be represented by the pair of object-variables <e,de>. As far as the selected record is concerned, we could have written the interpretive semantics as one long expression involving the observer functions of the data types as follows:

```
e:= GET(VAL(D#DE,(FIRST(FIND(D#DE,P))),1,Q)
```

Example 2. Relative retrieval using cursor

Suppose the FIND statement in the example above has been used to set the cursor variable e. One can then use the following statement in UDL to select the next member record within the same OCS :

```
FIND UNIQUE (EMP AFTER e) SET (e)
```

This statement is interpreted in our model as:

```
e := (GET(VAL(D#DE,de),
                    (POS(VAL(D#DE,de),e,Q)+1),
                    Q)
```

where Q is the ordering predicate defined over the members of the OCS DE. Note that we have used POS to get the ordinal number of the member record pointed to by e, incremented the ordinal number and used the incremented number as argument to GET to retrieve the identity of the next record. This illustrates how the functions POS and GET provide navigation within an OCS.

Example 3. Multivariable Qualification

It should be noted that the primitive function FIND of data type TYPESTATE is the basis of set-oriented retrieval. However, as a typestate is concerned with objects of one type only, retrieval through FIND operation, though may be quite complex involving many boolean conjunctions, is limited to one-variable qualification expressions. Multi-variable retrievals can be effected by nesting of one FIND within the qualification expression of the next and so on. For example, find the budget of the department which employs the employee with SSNO=s, can be expressed as follows:

```
READ(VAL(D#DEPT,
         OWNER(VAL(D#DE,FIRST(FIND(D#DE,P))))),
     BUDGET)
```

where the qualification expression P is

```
P::(VAR x:OWNS(VAL(D#DE,x),FIRST(FIND(D#EMP,P1)))
```

The qualification P1 on the typestate for employees is as follows:

```
P1:: (VAR y: READ(VAL(D#EMP,y),SSNO)=s)
```

The features of the CODASYL DDL specifications [Codasyl 71] not handled in our formulation are:

- repeating groups
- multiple member types for an OCS type
- storage structure information.

The various specifications of ordering, membership class, and primary keys (fields for which duplicates are not allowed) will be treated in our formulation as integrity assertions. We shall give some examples of how these assertions can be expressed in section 3.2.

3.2 Database Assertion Language

In this section, we shall describe an assertion language which is used for the following purposes:

- as DDLs, they provide the syntax and semantics of the data structures and the consistency criteria;
- as assertion language of the DML logic, they characterize properties of database states and its proof-theory to prove verification conditions;
- as expression language for the assignments in the DML, i.e., for writing qualification and retrieval expressions.

We obtain this assertion language and its proof-theory by including the specifications of the three data types, TYPESTATE, RECORD and OCS, into a many-sorted first-order calculus. The functions of these data types can be composed to form the terms of this language. The axioms of these data types, a substitution rule (defined following) together with the tautologies and inference rules of first-order calculus, form the proof-theory of the language. To see why this language can be used to describe a database state, recall that a database state is a union of type-states, one for each object type and any type-state can be canonically represented by an expression involving only the constructor functions of the data type. For example, a database state having only one record of type EMP with some constant values s, d and r as values for the fields SSNO, DNAME and SALARY can be described by the following formula:

```
D#DEPT=INIT(DEPT) ∧ D#DE=INIT(DE) ∧ D#MGR=INIT(MGR)
∧ DEFINED(D#EMP,e) ∧ (D#EMP=CHANGE(EXTEND(INIT(EMP),e),
                                   WRITE(
                                   WRITE(
                                   WRITE(EMPTY,SSNO,s),
                                             DNAME,d),
                                             SALARY,r))
```

To use the language as a DDL, we observe that apart form introducing the symbols for fields, record types and OCS types, a Network

Schema will contain a set of consistency criteria which are essentially the properties of database states consisting of admissible values. In our assertion language, the parameters of the language coincide with the symbols declared in the schema and the consistency criteria are expressible as quantified formulas.

In the following, we shall define the syntax and semantics of the assertion language L and its proof-theory. Then we shall illustrate its use as a DDL. Finally, we shall how the terms like consistent database state can be defined in logical terms and give a method to determine whether a given state satisfies the consistency criteria.

3.2.1 Language L and its proof-theory

Here we introduce a many-sorted assertion language L with the following set of sorts M= {token, type-name, field-name}. The symbols of the language are as follows:

Logical Symbols

1. Parentheses and the usual logical connectives: (,),¬,∧,∨.
2. Variables: lower-case letters for sort **token**, upper-case letters for sorts **type-name** and **field-name**. All OCS and record-types defined in a database schema are variables of sort **type-name**.
3. Equality symbols: for each sort i there may be the predicate symbol $=_i$ said to be of sort (i,i). The equality symbols introduced for the data types TYPESTATE, RECORD and OCS are included.

Parameters

1. Quantifiers: for each sort i there is a universal quantifier symbol $\forall_i$.
2. Constant symbols: The constants of sort token include numbers, strings, $\perp$, undefined, and all instances of the data types TYPESTATE, RECORD, OCS and LISTID. The constant symbols for sort type-name are INTEGER, STRING, and the constants of sort field-name are the field-names defined in the database schema.
3. Predicate symbols: Apart from the predicate symbols defined on

integers, all boolean-valued function symbols defined for the data types TYPESTATE, OCS, RECORD and LISTID are included. Also, there is a **membership predicate** $\in$ of sort (token, type-name). The intended interpretation of $\in$(t,T) is that t is an instance of type T, i.e., it is equivalent to ISELEMENT(FIND(D#T,all),t).

4. Function symbols: Usual functions defined on integers and strings together with all the non-boolean valued functions defined for the data types TYPESTATE, OCS, RECORD and LISTID.

Terms and wffs are defined as in one-sorted first-order languages, except that sort compatibility must be respected when using quantifiers, predicate symbols, and function symbols.

A many-sorted **structure** A for L is a function from the set of parameters of L assigning

1. to the quantifier symbol $\forall_i$, a non-empty set U_i, called the domain of sort i;

2. to each predicate symbol p of sort $(i_1,...,i_n)$, a relation
$$p_A \subseteq U_{i_1} \times ... \times U_{i_n};$$

3. to each constant symbol c of sort i, an element c_A of U_i;

4. to each function symbol f of sort $(i_1,...,i_n,i_{n+1})$, a function
$$f_A: U_{i_1} \times ... \times U_{i_n} \rightarrow U_{i_{n+1}}.$$

A **state** I for L is a function from the set of parameters and variables of L such that I restricted to the parameters of L is a structure A of L and I assigns to each variable x of L of sort i an element of the domain U_i. The notion of wff P of L being **valid** in a state I is that when each free variable of sort i in P is given a value from domain U_i, P becomes true. We shall write $I \models P$ to mean P is valid in state I.

The logical axioms and rules for many-sorted first-order languages are

those of first-order languages, again taking into account sorts. The theory T of the language L includes all logical axioms and rules plus a set D of formulas of L, the **nonlogical axioms** of T. The set D consists of all the axioms defined in the specifications of the data types TYPESTATE, OCS, RECORD and LISTID. When a formula P of L is derivable from a set F of formulas of L using the axioms and rules of T, we write $F \vdash_T P$.

Notations: To make the formulas of L more readable, we introduce the following syntactic abbreviations:

```
D#T[t] ⇔ VAL(D#T,t);
D#T[t<--e] ⇔ CHANGE(D#T,t,e);
r.F ⇔ READ(r,F)
```

In the light of these abbreviations, the axiom d3 of data type TYPESTATE is rewritten as :

```
D#T[t1<--e][t2] = if t1=t2
                    then e
                    else D#T[t2]
```

3.2.2 Language L as DDL

We now give precision to some concepts of Network Data Model within the framework of the assertion language L.

A **network schema** N is a 4-tuple (L,T,F,C), where

1. $\underline{L}=\{L_1,...,L_m\}$ is a set of distinct variables of L, the OCS names of N, where L_i is of sort **type-name**, i in [1:m]. Each L_i itself is a pair $<T_j,T_k>$, where T_j and T_k are in T and are variables of L of sort **type-name**.

2. $\underline{T}=\{T_1,...,T_n\}$ is a set of distinct variables of L, the record type-names of N, of sort **type-name**. Each T_i is a tuple $<F_1,...,F_j>$ where j depends on i and each F_k, k in [1:j], is constant of L of sort **field-name**.

3. $\underline{F}=\{F_1,...,F_p\}$ is a set of distinct constants of L, the field-names of N, all of sort **field-names**.

4. $\underline{C}$ is set of wffs of L containing the following:

OCS formation:
for each L_i in $\underline{L}$, where $L_i = \langle T_{1_i}, T_{2_i} \rangle$,

the formula

$$(\forall l)(\in(l, D\#L_i) \Rightarrow (\forall t)(OWNS(D\#L_i[l], t) \Rightarrow \in(t, D\#T_{2_i})) \wedge (\forall t)((OWNER(D\#L_i[l]) = t) \Rightarrow \in(t, D\#T_{1_i})))$$

record formation:
For each T_i in $\underline{T}$, where $T_i = \langle F_{1_i}, \ldots, F_{p_i} \rangle$,

the formula

$$((\forall t) \in(t, D\#T_i) \Rightarrow \bigwedge_{k \text{ in } [1:p]} ((\forall x)((D\#T_i[t].F_{k_i} = x) \Rightarrow \in(x, DOM(F_{k_i})))))$$

consistency criteria: other wffs. ///

The OCS formation formula states that every OCS occurrence has existing owner and member occurrences of proper types. The record formation rule states that every record occurrence has values in its fields picked from proper domains.

Now we can define a **database state** $I_{\underline{N}}$ to be a valuation of $\underline{N}$ such that each variable of sort type-name is assigned to a type-state instance and each token-variable is assigned to either a object-identity, an instance of data types RECORD, OCS or LISTID, or a value in the domain of a field. A **consistent database state** I is one for which $I \models P$ for each P in $\underline{C}$.

3.2.3 Expressing Consistency Criteria in L

We now introduce a simple example that illustrates the ability of our assertion language to express properties of database states. Consider the database schema shown in figure 3-4. Order clauses and membership class clauses are missing from the schema. We express these, instead, as integrity assertions.

```
SCHEMA PERSONNEL
TYPE   DEPT = RECORD  (* DEPARTMENT *)
                 DNAME: STRING;
                 BUDGET: INTEGER
               END;
       EMP  = RECORD  (* EMPLOYEE *)
                 SSNO: STRING;
                 DNAME: STRING;
                 SALARY: INTEGER
               END;
       DE   = OCS      (* DEPARTMENT'S EMPLOYEES *)
                 OWNER: DEPT;
                 MEMBER: EMP
               END;
       MGR  = OCS      (* MANAGER'S SUBORDINATES *)
                 OWNER: EMP;
                 MEMBER:EMP
               END;
END PERSONNEL
```

Figure 3-4: Example Database Schema

Example 1. Automatic Membership

In OCS type DE the members are automatic. This is expressed as:

(C1):: (∀e ∈ EMP) (∃de ∈ DE): OWNS(D#DE[de],e)

Here e and de are variables of type object identifier bound to the object types EMP and DE. Note that D#DE[de] is our notation for VAL(D#DE,de) and is required for the type conversion from object identifier to its value.