CONCEPTS and METHODS
for
DATABASE DESIGN

Paul C. Gilmore

Technical Report 87-31
August 1987

Abstract

This report consists of drafts of chapters of a book prepared as course material for CSCI 404 at the University of British Columbia

INDEX

CHAPTER 1. INTRODUCTION

Sections:

1. RECORD KEEPING
2. DATA and DATA PROCESSING
3. MACHINE ASSISTED DATA PROCESSING
4. DATA PROCESSING PROGRAMS
5. FILE MANAGEMENT SYSTEMS
6. DATABASE MANAGEMENT SYSTEMS
7. INFORMATION NEEDS ANALYSIS AND DATABASE DESIGN
8. FOURTH GENERATION LANGUAGES
9. KNOWLEDGE BASE SYSTEMS
10. OVERVIEW OF BOOK

CHAPTER 2. BASIC CONCEPTS

Sections:

1. ENTITIES and SETS
    1.1. Domain of a Set
    1.2. Extensions Varying with Time
    1.3. Ordered Sets and Tuples
    1.4. Cartesian Products and Associations

2. DECLARATIONS and ARITY DOMAINS
    2.1. Base Set Declarations
    2.2. Naming Sets
    2.3. Arity Domains

3. DEGREE DECLARATIONS
    3.1. Degree of an Entity for an Association
    3.2. Business Practices and Degrees
    3.3. Notation for Degree Declarations
    3.4. Partial, Total, Singlevalued, Multivalued, and Functional Associations
    3.5. Interpretation of Degrees of an Association with Domain an Association
    3.6. Degrees of Nonbinary Sets

4. VALUE SETS and DEFINED SETS
    4.1. Strings and Primitive Value Sets
    4.2. Declarations of Defined Sets
    4.3. Defined Value Sets

5. ATTRIBUTES
    5.1. Functional Attributes

5.2. Identifiers

6. PARTITIONS of DECLARED SETS
6.1. Choosing Base Sets
6.2. Domain Predecessors

7. A SET SCHEMA for the XYZ CORPORATION

8. DOMAIN GRAPHS and TREES
8.1. Directed Graphs
8.2. Directed Graphs as Domain Graphs
8.3. Paths and Cycles
8.4. Trees from Undirected Cyclic Graphs
8.5. Degrees for Edges of Domain Graphs and Trees

9. TYPES and ELEMENTARY ASSERTIONS
9.1. The Extended Schema schema* and Types
9.2. Terms and Elementary Assertions
9.3. Type Assigning Elementary Assertions
9.4. Equality Assertions

CHAPTER 3. THE LANGUAGE DEFINE
Sections:
1. DOMAIN+VARIABLE DECLARATIONS

2. A MANAGEMENT SYSTEM for SET SCHEMAS
2.1. Internal Surrogates and Tuples of Surrogates
2.2. Users' View of the System
2.3. Immediate Predecessors

3. ELEMENTARY and EQUALITY ASSERTION CLAUSES
3.1. Assigning Truth Values to Elementary and Equality Assertions

4. THE BOOLEAN TRUTH VALUES and OPERATORS
4.1. Truth Values
4.2. Conjunctions and Disjunctions
4.3. Negation
4.4. Assertions wirth Identical Truth Values

5. QUANTIFIERS
5.1. Informal Introduction toExistential Quantifiers
5.2. Informal Introduction to Universal Quantifiers
5.3. Quantifier Variable Declarations
5.4. Assigning Truth Values to Quantified Assertions

6. NESTED DECLARATIONS

7. PARAMETERIZED SET and TUPLE NAMES
- 7.1. Parameterized Set Names in Quantifier Prefixes
- 7.2. System Declared Functions
- 7.3. Ordered Sets and Parameterized Tuple Names
- 7.4. Nested Parameterized Names
- 7.5. The Form and Meaning of Parameterized Set Names

8. QUERIES

9. The CONSISTENCY of TRUTH VALUE ASSIGNMENTS

10. EXTENSIONS of DEFINE

CHAPTER 4. INFORMATION NEEDS ANALYSIS
Sections:

1. NAMING SETS
   - 1.1. Restrictions on the naming of Sets

2. GUIDING PRINCIPLES for the CHOICE OF BASE SETS

3. The CHOICE of VALUE SETS

4. NULL VALUES

5. ENTITY-RELATIONSHIP DIAGRAMS

6. DATA STRUCTURE or BACHMAN DIAGRAMS

7. IS_A HIERARCHIES

CHAPTER 5. DESIGNING TABLES
Sections:

[Gil87a] is attached

CHAPTER 6. DESIGNING NETWORKS
Sections:

To be supplied

REFERENCES

CHAPTER 1. INTRODUCTION

1. Record Keeping

Record keeping is fundamental to all cooperative human activity and to much individual activity as well, and evidence of it is as old as writing. The human brain is capable of recording prodigous amounts of diverse information, but it is not an accurate, reliable, and verifiable means for keeping most of the elementary information vital to the operation of human societies. It would be impossible to operate a chequing account at a bank, for example, if it was necessary for the bank and an individual to agree on the selection of an individual to remember all the transactions of the account and report on the current balance. No individual chosen could be counted on to remember correctly all the transactions necessary, to be available whenever called upon for information, and if that individual's memory was the sole repository for the information, there would be no means to settle disputes as to its accuracy.

The verb "to record" comes from the latin "to remember". For remembering such a personal thing as future engagements, individualized engagement calendars are widely used. For remembering the withdrawals, cheques written, and deposits for chequing accounts, registers are provided to individuals by banks so that such transactions can be written down as they occur, and so that they can be remembered when necessary. The bank also sets down in some form of writing their experience with transactions for the account, with the result that a basis for settling disputes between the individual and the bank is provided by the two records.

Since record keeping is so widely and frequently used, short cuts are necessary to decrease the time involved and to increase the accuracy of the records. An engagement calendar for a year has for each day a designated space for recording the engagements of the day. An individual may write 'Bob's birthday' into the space for a particular day, say August 17. The format of the calendar, when properly understood, permits the person to look up that date and read any entries that have been made. If the person remembers who Bob is, then the person is reminded that August 17 is his birthday. The entry 'Bob's birthday' by itself does not suggest the date of the birthday, but recording its presence in a designated space does. Similarly, the sequence of strings of characters '5/8/87', '821', 'UBC bookstore', '39.46' has no meaning by itself, but does take on real significance if it is recorded in a cheque register, the first under 'Date', the second under 'Cheque No.', the third under 'Cheque Issued to or Description of Deposit', and the fourth under 'Cheque Amount'. Each of the strings must be supplemented by additional information to be unambiguously interpreted, but under one convention these entries can be interpreted as abbreviating 'Dated August 5th, 1987, cheque number 821 was written to the order of the UBC bookstore in the amount of $39.46'. The four short strings are not only easier to write than the full sentence, but it is easier to review

and process the entries in the cheque register than it would be the full sentences that they abbreviate.

2. Data and Data Processing

Each of the strings of characters '5/8/87', '821', 'UBC bookstore', and '39.46' is a **datum**, sometimes called a **data item**. The **data** (plural of 'datum') entered into the cheque register is the source of information on transactions in the chequing account. Information on transactions is obtained by **interpreting** the data in the register. For example '5/8/87', because it appears in the date column of the register is interpreted as the date; if it is entered as day/month/year, then the data item is interpreted as the date 5 August, 1987, while if it is entered as month/day/year it is interpreted as the date May 8, 1987. The **format** of the register and the data entered into it, is therefore fundamental to the interpretation of data entered into it. If the data is entered in the proper place and in the proper form, the cheque register becomes a reliable and verifiable source of correct information on the transactions. But if data is not entered, or not entered in the proper columns in the correct format, then confusion rather than good information is the result. Therefore essential to the cheque register is a discipline of careful data entry and a precise **data description** setting out the format of the register itself and of the data that may be entered into it.

To enter or change data, to read and interpret data, or to read data and to calculate the current balance are all common examples of the **processing** of data for the cheque register. But other examples abound. A person may wish to calculate the total of all deposits made in the last month, the total of all cheques to charities in the past tax year, or the total interest paid to the account, all examples of specialized data processing that is sometimes called **data manipulation**, since it does not involve entering, changing or deleting data. There is no limit on the amount and kind of data manipulation that an individual could do on a cheque register, the limit is only on the patience, time, and needs of the individual. If no mechanical or electronic aids are available, the individual's limit of patience and time is likely to be quickly reached, no matter the needs; but with such aids, more needs can be met within the limit of patience and time. This is the promise of machine assisted data processing for individuals, as well as for enterprises of all kinds.

3. Machine Assisted Data Processing

Data manipulation often involves large amounts of simple computations. The tedious nature of the task has long been recognized. For example, Pascal wrote in an "Advertisement" for a machine that he had designed and built in 1642, "Dear reader, this notice will serve to inform you that I submit to the public a small machine of my invention, by means of which you alone may, without any effort, perform all the operations of arithmetic, and may be relieved of the work which has

often times fatigued your spirit, when you have worked with the counters or with the pen" , while Leibniz, who thirty years later improved on Pascal's machine, wrote ".. it is unworthy of excellent men to lose hours like slaves in the labor of calculation which could safely be relegated to anyone else if machines were used". [D.E. Smith, A Source Book in Mathematics, Vol. 1, Dover, 1959].

Pascal's and Leibniz's machines were mechanical calculators with features that are still present in today's machines. As calculators they served to assist in the computations that are part of data processing, but not in the repetitive **data entry** that is also a part. If a cheque register is maintained only in written form, then each time a calculation is made on a mechanical calculator, data must be read from the register and entered into the calculator. Repetitive reentry of data is not only tedious, but it is also time consuming and error prone. To eliminate the necessity for it requires that a means be found to store data in a machine readable form that can be repetitively processed.

Cards with holes punched in them to record data were the next major step forward in mechanical data processing. By sensing where holes are punched in a card, a machine can "read" the data encoded in a card. Thus if data entry is thought of as the punching of holes in cards, then it need only be done once, no matter how often the data is to be manipulated. The invention of the punched card for storing data is attributed to Jacquard who in 1805 invented a loom that used such cards for controlling the patterns produced in the cloth woven by it.

The use of the card for true data processing began with the United States census of 1890. Machines invented by Hollerith were employed to record the raw data from the census in punched card form, and to manipulate the data by sorting and tabulating the cards. The necessity of the Hollerith machines for the census, and the effect of them on the results derived from the census can be judged from the following quote from [Robert P. Porter, "The Eleventh Census', *Proceedings of the American Statistical Association*, no. 15 (1891)]:

> The Eleventh Census handled the records of 63,000,000 people and 150,000 minor civil divisions. One detail (characteristic) alone required the punching of one billion holes. Because the electrical tabulating system of Mr. Hollerith permitted easy counting, certain questions were asked for the first time. Examples of these were:
>
> Number of children borth
> Number of children living
> Number of family speaking English
>
> By use of the electric tabulating machine it became possible to aggregate from the schedules all the information which appears in any way possible. Heretofore such aggregations had been limited. With the machines, complex aggregations can be evolved at no more expense than the simple ones.

With data manipulation made easier, needs can be satisfied that would otherwise be neglected. The questions that were posed depended upon the **base** of census data, or **database**, of punched cards for their answers; more recent terminology would call them **queries** for the database. The quote illustrates a continuing theme of data processing: When queries can be more easily answered, that is when data manipulation is simplified and its cost reduced, information needs can be satisfied that would otherwise be neglected.

As detailed in [Herman H. Goldstine, The Computer, from Pascal to von Neumann, Princeton University Press, 1972] Hollerith's machines evolved into the punched card machines that remained fundamental to data processing, including what is more commonly called scientific computation, until well after the first commercial uses of electronic computers in 1950.

4. Data Processing Programs

The punched card of the Jacquard loom recorded data of a very special kind. The holes in the card for such a loom were instructions to the machine as to which threads were to be manipulated to form the intended pattern in the woven cloth. Each card was a **program** for the loom. They were not data processing programs since the function of the loom was to process threads, not data. But the concept that they have in common with data processing programs for a modern computer is that the same machine was programmed to do many different things.

Beginning approximately in 1960 data processing programs were increasingly written in higher-level languages such as Fortran, Cobol, and PL/1. Each such **application program**, as they are often called, was written for a specific purpose, but they all depended upon one or more files of records of data. A payroll program for printing pay cheques would require data on the regular and overtime hours worked for hourly employees, other data on government and employer dictated withholding for taxes and pensions, and still more data on name and addresses and perhaps bank accounts of employees into which salary was to be deposited. Some of this data would be required by a program that provided quarterly reports and payments to governments for taxes withheld, but that program would require further data on the reporting requirements. The number, complexity, and size of the files required for the data processing programs of a typical enterprise have grown at a rapid pace as more and more applications were seen to be essential for or useful to the operation of the enterprise. The only brake on the introduction of new useful programs has been the cost of creating and maintaining them.

It was noted earlier that the ingredients for a successful cheque register are a discipline of careful data entry and a precise data description setting out the format of the register itself and of the data that may be entered into it. These ingredients are even more critical for the maintainance of a files used by one or more data

processing programs. The commands of a program reading from or writing to a file, that is the **input/output commands**, will only execute properly if they conform exactly to the format of the data specified for the file, and the whole program can only produce correct results if the data kept in the file is correct. These elementary facts, given the large number of existing programs that have been created and must be maintained, and the continuous demand for new programs, have profoundly influenced the direction of software development.

5. File Management Systems

Data has traditionally been recorded on paper and manually maintained in folders, drawers, and filing cabinets. The advent of computers has not diminished the need for such traditional storage methods, but has rather supplemented it. A **database** in the current sense, is a computer maintained collection of files of records stored in high speed memory, or on drums, disks, tapes, and mass storage devices. The creation and maintainance of such a collection requires:

-Interpreting data declarations made by users
-Deciding how and where data is to be stored
-Adding and deleting data from a file as directed by a user
-Retrieving data from a file at the request of a user.

These have traditionally become the responsibilities of the file management part of an operating system.

A file management system caters to users with their own individual files. Files that must be accessed by a program are declared in the program; a file that must be accessed by more than one program, must be declared in each of the programs. Every change to a file requires a change in the declarations of each program accessing the file.

A file management system gives its users full freedom to structure the data within the file. A user can use any data structure to record desired data, and must include in every program accessing the file the code necessary to process the data structures used in the file.

With all its advantages, a file management system nevertheless has serious disadvantages as well:

A user must take full responsibility for adding, deleting, and retrieving data from a file, and for backout in case of system failure during these operations.

To retrieve information from the database in response to a query, it is necessary to write an application program specifically written for the query.

When files are shared the corrupting of a file by one program or user, affects other

programs or users that must access the file.

When the format of data within a file is changed in any way, the declaration of every program accessing the file must be changed as well. When large numbers of data processing programs access the same files, the declarations of the formats of the files are repeated many times, and the cost in programming effort of making even minor changes can be prohibitive.

The problems that arise from the use of file management systems have been collectively referred to as the "maintenance mess" in chapter 1 of *Software Maintenance, The Problem and Its Solutions*, by James Martin and Carma McClure, Prentice-Hall, 1983.

6. Database Management Systems

Some, but not all of the "maintenance mess" can be cleaned up through the use of a database management system. Such a system caters to users and programs with shared files. If required, every user can share every file, with sharing only restricted by security considerations and application needs.

A database management system provides a high level **conceptual** or **logical** view of the data recorded in the files of the database through the use of a **data model**.

All declarations of data format and all commands for adding, deleting, and retrieving data from the database are expressed in a high level, generally nonprocedural, language that uses the data model as a description of the data in the database.

Centralized control over the declarations of data can be maintained by a **database administrator**. Decisions as to where and in what format the data is to be stored can be made to ensure good performance of the storage and retrieval commands, and to reinforce the reliability and security of the database.

Simplified individual views of the database can be provided that are specifically tailored for particular application programs and users. This too can increase the security of the database by limiting a user's knowledge of the database to just that data required for the applications for which the user is responsible.

A database management system provides three basic functions to its users:

1. The system accepts the declaration of a **schema** that describes the format of data to be entered into the database in terms of the data model supported by the system.

2. Update, add, and remove commands compatible with the declared schema change the data recorded in the database as required.

3. Queries and individual views of the database are expressed as definitions.

7. Information Needs Analysis and Database Design

Given a database management system, the design and implementation of a database for an enterprise to be supported by the system, is a complex process that can be broken down into roughly four steps:

1.Information needs analysis:

An informal process during which information is obtained about the data that is to be recorded in the database. The process requires an examination of the data currently being process, either manually or by machine, as well as an understanding of how the data needs of the enterprise are likely to evolve. The information gathered during this stage is needed during the subsequent stages of design in order to:

- clarify the meaning and use of data to ensure that the analysts' and users' have a common understanding of how the data is to be interpreted; and
- ensure that the data is properly represented during the formal stage of design.

During the later operation of the database the information is needed:

- by the management system to maintain the integrity of the database
- users to help them understand the database and correctly interpret retrieved data.

This stage frequently requires the examination of manually maintained databases as well as existing computer maintained databases.

2. Formal schema design:

The data model supported by a database management system determines the kind of schemas used to specify formats for the data to be entered into the database. During this stage schemas must be designed that will ensure that any data determined to be necessary for the enterprise, can be completely and accurately recorded in the database.

3. Physical design:

A database management system generally allows a user to specify how and where data is to be stored to ensure efficient retrieval. During this stage must be determined which data structures supported by the management system are to be used to store data, and where in the physical storage devices available to the system are files of these structures to be recorded.

4. Load & test:

Actual data is loaded into the database during this stage and the design is tested for completeness, accuracy, and efficiency.

These steps are rarely followed in a simple sequential order, but rather later stages of design contribute to earlier stages with the consequence that several cycles through all the stages may be necessary to arrive at a suitable design. But as the needs of the enterprise evolve and grow, so must also the design of the database. However, a good initial design, when supported by a good management system, does shoud provide the basis for many years of evolution and expansion.

Of the four stages, the information needs analysis stage is often the most difficult and time consuming, for it requires a unified view of the information needs of the enterprise rarely achieved by one person. Although one person may be responsible for the information needs analysis for an enterprise, many different people are likely to be involved if the enterprise is of any size or complexity. For only by questioning users of data and examining how they use it, is it possible to know what information is obtained from the data by interpreting it. An understanding of simply the format of data being used is not enough, since that will not lead to an understanding, for example, of how business practices of the enterprise affect the relationships between different collections of data.

Several methods have evolved for undertaking an information needs analysis. The most important of these as far as this book is concerned is the entity-relationship approach of Chen that evolved from earlier methods developed by Bachman. These methods will be discussed at greater length in chapters 2 and 4.

8. Fourth Generation Languages

The use of database management systems will reduce some of the "maintenance mess" arising from the use of file management systems. For example, queries can be expressed in the data manipulation language of the management system, rather than in the third generation application languages like FORTRAN,COBOL, PL/1, PASCAL, or ADA. These third generation languages, like the languages of the two preceding generations, namely machine language and assemply language, can only be written and understood by trained programmers. So the maintainance of programs written to provide answers to routine queries will also require the services of trained programmers. This why the need for trained programmers has in the past grown at such a rapid pace.

The generally nonprocedural fourth generation data manipulation language of a modern database management system, on the other hand, can be understood by users of the system. They can pose queries in the language and have them answered by the system without the intervention of any trained programmers. Responsibility for the maintainance of the database of an enterprise can be left in the hands of specialized personnel, but this task does not grow in size and complexity as the number of application programs and queries written for it grows.

The only part of the "mess" left in the hands of trained programmers by early database management systems was the maintainance of more complex data processing programs such as report generators. But the newer fourth generation languages have extended the data manipulation languages of database managment systems to provide languages that integrate the specification of data processing programs and routine queries into, and reduce even more the need for trained programmers.

9. Knowledge Bases and Fifth Generation Languages

A database management system is used by an enterprise to maintain a repository of data from which, through proper interpretation, the information essential to the operation of the enterprise can be obtained. The advantages cited by Date in [Date83] for the use of a database management system are the reduction of redundancy, assistance in the elimination of inconsistencies and inaccuracies, the sharing of data by many users, and the enforcement of standards and security restrictions. As the information recognized as essential to an enterprise grows in sophistication and subtlety, so too must the database. More advanced databases are now called knowledge bases, since the interpretation of the data recorded in them results in qualitatively different kinds of information from that in traditional databases. Nevertheless the advantages cited for database management systems must not be lost for knowledge base management systems intended for wide application.

Enterprises operate in a world in which much of what should be known is not, and in which some of what is known is incorrect. Database management systems must assist enterprises in reducing ignorance and error. Such systems must be able to remind their users of missing data, and notify them of what might be unanticipated consequences of proposed updates. To the extent that it is possible for what may be a distributed repository, the internal consistency of the repository should be maintained.

Because of the sophistication of the data recorded in a knowledge base and the difficulties involved in its interpretation, increasingly sophisticated languages are needed as data manipulation languages. For example, to maintain the accessability of the knowledge base to untrained personnel data manipulation languages have been designed that are intended to be as easy to use as natural languages such as English. Although it is not possible to remove from a user the responsibility to understand and formulate complex queries, it is possible to permit the formulation of such queries in a language comprehensible to the casual user.

10. Overview of the Book

The purpose of this text is to provide an understanding of some of the concepts basic

to database and knowledge base systems. The goal is to give a full overview of the design process from information needs analysis for an enterprise through to the design of a schema for a particular database managment system, and at the same time deal with some aspects of physical implementation. The intuitive appeal of the entity-relationship approach to information needs analysis has lead to its widespread use, and it belongs as a part of any introductory course on database.; at the same time a business oriented view of information rather than a data view can be presented, an important broadening of perspective.

Tables are presentation data structures; the name 'relation' given to them in the relational model gives that model its name Codd70, 72, 79].The simplicity of the table view of data has resulted in the widespread use of relational database management systems. As a consequence a number of methods for designing relational databases have been described. A recent survey of such methods appears in [TYF86]. The design of a table schema from a set schema is also a natural topic of an introductory course on database. Finally a discussion of the network or DBTG model of data gives an introduction to at least one form of implementation, but is also important because of the wide continued use of network database management systems.

An introductory course covering the three topics, entity-relationship design methods, relational schema design, and implementations via a network model, becomes rather disjointed, very crammed, and somewhat superficial when these subjects are taught in a traditional way. The intuitive entity-relationship design methods are informal and imprecise, and require considerable practice with examples to master. The design of relational schema via traditional normalization methods can be presented in a more formal way, but the methods are imprecise in their goals, at times outright contradictory, and dependent upon a dry and unintutive theory. The teaching of a query language like SQL for the relational model is a necessity, but the language requires a lot of practice to master its more complex features. Finally, the construction of network implementations of relational schema has been another art that is difficult to teach, much less master.

This book provides a unified treatment of these subjects through the use of the SET model and the language DEFINE.

The SET model is based on the concept of set or class, one of the most fundamental concepts of mathematics. The concept has been incorporated into scientific and natural languages, as well as used extensively in database theory and practice. It is explicitly used in the relational data model and in the entity-relationship approach to information needs analysis. It has also been used in the entity set [SAAF73], semantic [HaMc81], and functional data models [Ship81, LyKe86], in the system TAXIS [MW80], in the techniques described in [FuNe86], and is implicitly used in the network data model [TaFr76]. But the concept of set used is an intuitive one and

is combined with other related but independent concepts. The SET model, on the other hand, uses set and ordered pair as its only fundamental concepts, with the mathematical foundation for these concepts being provided by the provably consistent set theories of [Gil86], while other needed concepts are defined in terms of these using the specification/and data manipulation language DEFINE.

In chapter 2 the groundwork is laid for the SET model, and some experience provided in an information needs analysis for a mythical XYZ corporation. The result of such an analysis is a collection of declared sets is a set schema that describes a formal SET model of the information needs. In chapter 4 some of the finer points of information needs analysis are presented after necessary definitions are provided in chapter 3.

The language DEFINE is introduced in chapter 3 to permit the definition of a set in terms of previously declared sets. Such definitions are necessary not only for the information needs analysis forming the basis for a SET database, but also for the formulation of queries for such a database.

One of the major advantages of the SET model is demonstrated in chapter 5, where tables are introduced as defined sets in a set schema. By defining an appropriate collection of tables as a table schema , a table view of a set schema can be provided. A simple technique is described for constructing a table view of a database schema that was first described in [Gil87a,b]. The technique uses an algorithm for constructing a table view that faithfully captures all the information of the database schema, provided that a collection of integrity constraints for the relational model are maintained. The technique also suggests some of the limitations of the relational model that are now becoming more widely recognized [TrLo87]. The chapter ends with a discussion of the data manipulation language SQL of the relational model.

In chapter 6 the network model is discussed and a technique described for designing network schemas. A network schema consists of a collection of what are called DBTG (for Data Base Task Group) sets [TaFr76]. An implementation of a network schema is described in terms of pointers that may provide some understanding of how a database for a set schema might be implemented. A simple technique is described for constructing a network view of a set schema that faithfully and efficiently captures all the information of the database schema, again provided that network integrity constraints are maintained. In the same chapter appears a brief discussion of the hierarchical model of data.

Like the entity-relationship approach to information needs analysis, the SET model is object-oriented in the sense currently being used for this term [Cox86]. A chapter to be written will discuss the object-oriented approach being used in system design.

Numbering convention for sections and subsections.

Introductory sections are not provided for chapters. Rather an introduction to a chapter is provided prior to the numbered sections.
Sections within a chapter are numbered beginning with 1, and subsections are similarly numbered. References to sections will be in the form:
chapter 2 section 9.2.
When the chapter number is omitted, the reference is to the current chapter.

Numbering conventions for figures.
In order to assist in the location of figures, figures will be given the number of the subsection in which they appear (no figures appear in the introduction to chapters); if there is more than one figure in a subsection, then they are numbered beginning with 1. The convention for with references to subsections is used in references to figures. For example,
chapter 2 figure 9.1
refers to the only figure appearing in section 9.1 of chapter 2, while
chapter 2 figure 9.3.2
refers to the second figure appearing in section 9.3 of chapter 2.

CHAPTER 2. BASIC CONCEPTS

In this chapter the basic concepts of the SET model are described. Since they form the foundation for the remainder of the book, it is important that they be thoroughly understood. At the same time the chapter provides an introduction to information systems needs analysis. This is the process by which the information needs of a company are determined. The exposition here is presented as an information needs analysis for the XYZ company.

1. ENTITIES and SETS

An **entity** is "a thing that has real and individual existence in reality or in the mind", while an object is "a thing that can be seen or touched; material thing" [Webster's New World Dictionary]. An object, such as a car, has real and individual existence in reality, while an entity may be an abstract thing such as colour with a real and individual existence in the mind. Entities, and not just objects, are important for databases, since it is often necessary to record information about abstract things.

A **set** is a selected collection of entities possessing a common property. The collection of entitites form the **extension** of the set, while the common property is the **intension** of the set. For example, the set EMPLOYEE of persons currently employed by the XYZ company is a set with intension "persons currently employed by the XYZ company" and with extension those persons who are currently employed by the XYZ company. An entity satisfying the intension of a set, and therefore in its extension, is said to be a **member** of the set. Thus if e is a current employee of the XYZ company, then e is a member of EMPLOYEE; this is expressed symbolicly as

e:EMPLOYEE.

Here ':' is a shorthand for 'is a member of'; traditionally '$\in$' , the Greek letter epsilon, has been used for this purpose, but the frequent use made of the notation makes the change of font inconvenient. Besides, the use of ':' in place of epsilon is now well established in programming languages in type declarations.

It is necessary to specify the intension of a set in order that the membership of the set can be clearly understood. For example, although the name of the set EMPLOYEE is suggestive of its intension, it is necessary to know its intension in order to determine that past employees, for example, are not to be members of EMPLOYEE. Similarly a name 'DEPARTMENT' for a set declared for the XYZ corporation may suggest different sets to different people. To one it may suggest the set of current departments, and to another it may suggest all departments that the corporation has had in the past and has in the present. An intension 'the currently recognized department' removes the ambiguity.

In database terminology a set is often referred to as an **entity set**. Here the adjective

"entity" does not change the meaning of set. It is added to emphasize that the members of the set may be physical or abstract things. Any entity about which information is to be recorded or, like strings of characters, are used to record information, may be members of a set. Throughout the book "entity" will not be added as an adjective to "set".

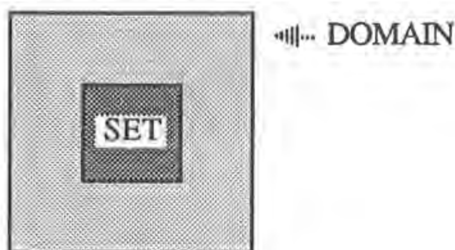
No set can be presumed to pre-exist. Each set identified in an information needs analysis of an enterprise must be **declared** by giving it a name and stating its intension. In section 2 a format for the **declaration** of sets will be described. The collection of all declarations obtained during the needs analysis for a particular enterprise is called the **set schema** for the enterprise.

1.1. Domain of a Set

The intension of a declared set should specify the **domain** from which the members of the set are selected. For example, the intension "the set of male members of EMPLOYEE" for a set MALE specifies EMPLOYEE to be the domain of the set MALE. The domain of a set must be a set that has been previously declared. But since no sets can be assumed to preexist, and the process of declaring sets must be started somewhere, some sets must be declared without a domain; such a set is called a **primitive set**. For example, the set EMPLOYEE as given above does not specify a domain from which its members are to be selected, and is therefore primitive. However, if a set PERSON for example was declared first, then EMPLOYEE could be declared to have PERSON as its domain.

The relationship between the extensions of a set and its declared domain is illustrated in figure 1.1.

FIGURE 1.1



The whole area inside the larger square, including the more darkly shaded area inside the smaller square, represents the extension of the domain of the set, while the smaller square represents the extension of the set. For example, the larger square could represent the extension of EMPLOYEE while the smaller square could represent the extension of MALE.

If EMPLOYEE is declared to be the domain of MALE, it is also said to be an **immediate domain predecessor** of MALE.

A **is a subset of** B if every member of A is also a member of B. Necessarily a set is a subset of its domain. Two sets are **extensionally identical** if each is a subset of the other. Necessarily two sets with the same intensions are extensionally identical, but two sets with different intensions may still be extensionally identical.

1.2. Extensions Varying with Time

Sets like EMPLOYEE that are identified during an information needs analysis have most usually **time dependent** extensions. This means that, although their intensions remain unchanged throughout time, a changing world results in a changing extension. EMPLOYEE certainly has a time dependent extension, as its intension clearly indicates. Another example is the set of cars licensed by the province of British Columbia. This set has a well defined extension at any time, but it is an extension that changes as new cars are licensed and old cars are removed from the road. However some sets of importance to databases have a **time independent** extension, or at least have extensions that vary very little. For example, the set of letters in the English alphabet has an extension that for the purposes of databases never changes with time. Similarly the set of digits 0, 1, ... ,9 has a time independent extension.

Two sets with time dependent extensions may have identical extensions at one time, and not identical extensions at another. For example, let MALE be the set of persons of the male sex currently employed by the XYZ company. Then EMPLOYEE and MALE would be extensionally identical at any time that the XYZ company had only male employees.

Again one set may be a subset of another as an accident of time, should they have time dependent extensions, or one may be a subset of another because of their intensions. MALE is a subset of EMPLOYEE at all times because of the intensions of the two sets. However, if MANAGER is declared to be the set of managers of departments of the XYZ company, then MANAGER may be a subset of MALE at a particular moment of time, and not a subset at another moment of time.

1.3. Ordered Sets and Tuples

A set never has duplicate members; a member appears in a set once and only once. However when the extension of a set is listed in some fashion, such as in a file, or a table, an order is given implicitly or explicitly in the listing to the extension of the set; that is, there is a first element in the listing that is a member of the set, a second element that is a member, and so on, and a member of the set may appear more than once as an element of the listing.

A set whose extension has been ordered in some fashion is called an **ordered set**. A

member of an ordered set is as before an entity that satisfies the intension of the set. An **element** of an ordered set is a member of the set together with its position in the ordering. Necessarily an ordered set has no more members than it does elements.

It is important to know that specifying an order for a set does not change the intension or extension of the set. An ordered set is just a set with an order that is to be imposed on its extension. The order specified may be only evident when the set is declared, or when the extension of the set is displayed in some fashion.

A **tuple** is an ordered set with finitely many elements explicitly listed. The notation <a1, ... , an> is used to denote the tuple with elements a_1, ... , a_n in the given order. For example, <1,2,1> is a tuple with two members 1 and 2 and with three elements, the first being 1, the second 2, and the third 1. The number of elements n in the tuple is called the **arity** of the tuple.The arity of <1, 2, 1> is 3. <'ABC', 'CAR'> is a tuple of arity two with members 'ABC' and 'CAR', and with two elements listed in the order 'ABC', and 'CAR'. A tuple of arity two is also called a **pair**, a tuple of arity three a **triple**, and a tuple of arity four a **quadruple.**

A tuple of arity one is not regarded as being distinct from the single entity that is its element; for example, <'ABC'> is a tuple with one element but will be regarded simply as another way of writing ABC. Tuples may be members of tuples. For example, <3,<1,2 >,3 > is a triple with members 3 and <1,2 >, and with first element 3, second element <1,2> and third element 3.

1.4. Cartesian Products and Associations

The **cartesian product** AxB of two sets A and B is the set of pairs <a,b> for which a:A and b:B. For example, if THREE is the set {1,2,3} and TB is the set {a,b} then THREExTB is the set {<1,a>,<1,b>,<2,a>,<2,b>,<3,a>,<3,b>}. The cartesian product of a set with itself may also be formed; for example, TBxTB is the set {<a,a>,<a,b>,<b,a>,<b,b>}.

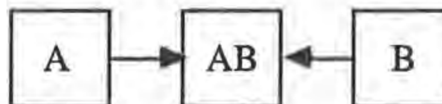
Cartesian products of any number of sets may be formed. The cartesian product A_1x ... x A_n of sets A_1, ... , A_n is the set of all tuples <a_1, ... , a_n > with n elements, where a_1 is a member of A_1, ... , and a_n is a member of A_n. When n =1, the cartesian product is just the set A_1.

A set of **arity** n is a set whose only members are tuples with n elements. For example, the set THREExTB, or any subset of it, is of arity 2. A set of arity n ≥2 is called an **association.** A **binary** association is an association of arity 2, a **ternary** of arity 3, and a **quadernary** association one of arity 4. By an **association** without qualification will always be meant a binary association, since attention will largely be restricted to them.

Given two declared sets, say EMPLOYEE and DEPARTMENT, the cartesian product EMPLOYEExDEPARTMENT of them may be thought of as being implicitly declared. This means that EMPLOYEExDEPARTMENT can be declared to be the domain of sets declared after EMPLOYEE and DEPARTMENT have been declared.

Given declared sets A and B, any set AB which is declared to have AxB as its domain is an **association** with domain AxB. The sets A and B are called **immediate domain predecessors** of AB. It is also said to be an **association between A and B**. The sets A and B are called **immediate domain predecessors** of AB. The relationship between AB and its immediate domain predecessors A and B is illustrated in figure 1.4.1:

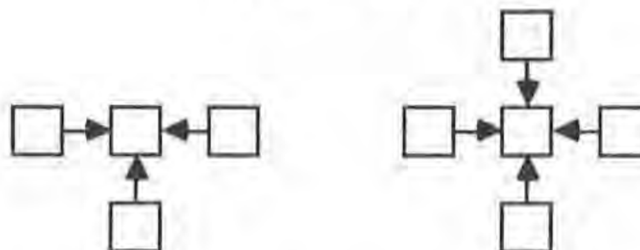
FIGURE 1.4.1



Diagrams such as this have a long history in databases. The first of this kind were introduced by Bachman in [Bach69] and were called **data structure diagrams**; they have since been also called **Bachman diagrams**. Chen in [Chen76] introduced another related diagram that he called **entity-relationship diagrams**. The diagram of figure 2.2 has elements in common with both of these earlier diagrams. They are called **domain diagrams**, since they illustrate the relationships between a set and the sets whose cartesian product form the domain of the set. The arrows directed from A to AB and from B to AB in figure 1.4.1 indicate that the domain of AB is either AxB or BxA, but does not indicate which of these is actually the case; that can only be determined from the set declarations.

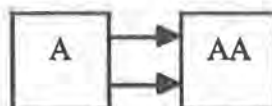
Domain diagrams will be used throughout the book. On those rare occasions when associations of arity greater than two must be portrayed, they are illustrated as shown in figure 1.4.2 for ternary and quadernary sets.

FIGURE 1.4.2



An association may be declared between a set and itself. For example, AA may be declared to be a subset of AxA as illustrated in the domain diagram of figure 1.4.3.

FIGURE 1.4.3



Let AB be an association between the sets A and B, as illustrated in figure 1.4.1, and let a:A and b:B; that is, let a be a member of A and b be a member of B. Then using the set membership notation, <a, b >:AB expresses that <a, b > is a member of AB, which is to say that a is AB associated with b. A convenient infix notation to express the same thing is a:AB:b.

2. DECLARATIONS and ARITY DOMAINS

During an information needs analysis of an organization, sets discovered to be central to those needs must be named so that they can be discussed, and their domain and intension must be recorded. The declaration of a set is a formal statement that gives this information about the set. In this section a declaration for **base** sets is introduced; that is for sets such as EMPLOYEE, DEPARTMENT, and EMPDEPT with intensions that can only be interpreted by humans. They are called 'base' since they are the foundation upon which databases are built.

2.1. Base Set Declarations

The declaration for base sets all take the form:

setnm for {domain declaration | degree declaration | comment}.

Here setnm is the name of the set, domain declaration is a machine interpretable clause that states the domain of the set, degree declaration is a second machine interpretable clause which may be blank or take a form described in section 4, and comment is a human interpretable statement in any language or format of a user's choosing describing the intension of the set. For example, the declaration

EMPLOYEE for {|| current employees}

declares the set with setnm 'EMPLOYEE', domain declaration blank, degree declaration blank, and comment 'current employees'. The name of the set is therefore 'EMPLOYEE'. The fact that domain declaration is blank for EMPLOYEE means that the set is primitive. The comment is an abbreviated statement of the intension of the set; the set is to consist of all current employees of the XYZ company. Similarly

DEPARTMENT for {|| a currently approved department}

is a declaration of the primitive base set DEPARTMENT. An example of a declaration of a nonprimitive base set is

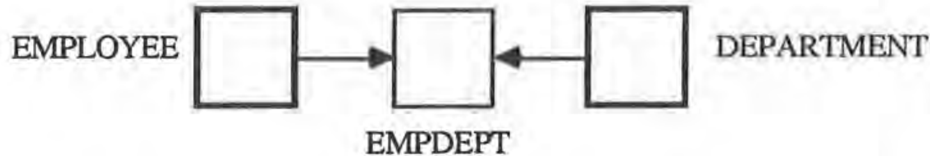
EMPDEPT for {EMPLOYEExDEPARTMENT|<1,1>,<1,*>| associates an employee with his/her department}

The set EMPDEPT has domain EMPLOYEExDEPARTMENT; that is, it is a

nonprimitive set since its domain is the cartesian product of two previously declared sets. The degree declaration '<1,1>,<1,*>' is a machine interpretable clause with a meaning that is described in the next section. As before comment describes the intension of the set.

The relationship between the three sets is illustrated in the domain diagram of figure 2.1, which is figure 1.4.1 with the sets renamed.

FIGURE 2.1



Here the boxes for the primitive sets are drawn using broad lines so that they can be easily recognized in the diagram. As noted before, it is not possible to determine from the diagram whether the domain of EMPDEPT is EMPLOYEExDEPARTMENT or DEPARTMENTxEMPLOYEE, and it does also not state the intensions of the sets and associations. The diagram is therefore not a substitute for the declarations of the sets, but simply a way of illustrating the relationships among the domains of the sets.

Other examples of sets discovered during an information needs analysis of the XYZ company are:

MANAGE for {EMPDEPT|<0,1>,<1,1>| associates manager of dept with dept}.
PROJECT for {|| projects to which employees may be temporarily assigned},
EMPPROJ for {EMPLOYEExPROJECT|| associates employee with project}, and
LEADER for {EMPPROJ|<0,*>,<1,1>| associates leader of project with project}.

2.2. Naming Sets

It is assumed that the declaration of a set uniquely identifies the set. That is, if a declaration is repeated, the second declaration will not be interpreted as the declaration of another set with the same name, domain, and intension, but rather the second declaration will be ignored. However, it is a common practice in database design to have two sets with distinct declarations but the same name. The practice is so widespread, and its value so evident, that it will be followed in this book. Since the name of a set is regarded as an abbreviation of its full declaration, the practice has the consequence that the name of a set will not necessarily identify it. However, names of sets are rarely used in isolation, and from the context in which they are

used it will usually be possible to determine the set that they are naming. In section 1 of chapter 4 simple restrictions are stated on the naming of sets that ensure that in most contexts the name of a set will uniquely identify it.

The choice of a name for a declared association should be made with some care, since a wrong choice can be misleading. For example, the name 'EMPDEPT' has been chosen rather than a name such as 'IS_IN', or 'HAS_MEMBER', because 'EMPDEPT' does not suggest a direction to the assignment, while the latter names do. Looking at the association from the point of view of an employee, the employee has been assigned to, that is IS_IN, a department. Looking at the association from the point of view of a department, the department has an employee as a member, that is 'HAS_MEMBER' employee. These special points of view are important for users of databases, but not immediately important to their design. This topic will be discussed again in chapter 3 when the language DEFINE is available for defining such associations as 'IS_IN' and 'HAS_MEMBER' in terms of 'EMPDEPT'; there the sets IS_IN and HAS_MEMBER will be seen to be aliases of the set EMPDEPT.

2.3. Arity Domains

The associations MANAGE and LEADER both may have a previously declared association as its domain. For example, the domain of MANAGE is EMPDEPT, while the domain of EMPDEPT is known from its declaration to be EMPLOYEExDEPARTMENT. The fact that MANAGE is a binary association can therefore be inferred from these domain declarations.

EMPLOYEExDEPARTMENT is the **arity domain** of MANAGE defined as follows:
The **arity domain** of a primitive set is the set itself; the **arity domain** of a set with domain a cartesian product of previously declared sets, is that cartesian product; the **arity domain** of a set A with domain B, is the arity domain of B.

'Arity domain' is so called because the arity of a set can then be determined from it: The arity of a primitive set is known to be one, the arity of a cartesian product is the number of sets in the product, while the arity of any set that is not an arity domain is the arity of its arity domain.

Thus since the arity domain of MANAGE is EMPLOYEExDEPARTMENT, its arity is two. Similarly the arity domain of EMPPROJ is EMPLOYEExPROJECT and the domain of LEADER is EMPPROJ, therefore the arity domain of LEADER is EMPLOYEExPROJECT. The arity of LEADER is therefore two; it is a binary set.

It is important to distinguish the arity domain of a set from the domain of the set. The domain of a set is declared when the set is declared. The arity domain of a set,

on the other hand, must be calculated from the domain declaration of the set and of other previously declared sets.

The fact that the arity of primitive set is always one has an important consequence: From the definition of arity it is possible to determine the arity of any set from its arity domain, and the arity of an arity domain is either that of a primitive set or that of the cartesian product of sets. As a consequence it is not possible to declare sets of mixed arity; that is sets with members say <1,2> and <1,2,1> that are tuples of different arity. It is of course possible to declare a set of tuples consisting of tuples of different arity, say with members <1,<1,2>> and <3,<2,1>>.

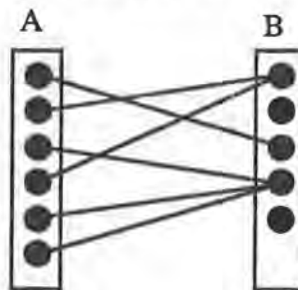
3. DEGREE DECLARATIONS

The information contained in the degree declaration of a base such as EMPDEPT is so important for the design of a databases that it is put in a machine interpretable form. Degrees can be declared for any nonprimitive base set, but generally degrees are declared only for binary associations, so these will be discussed first.

3.1. Degree of an Entity for an Association

Let AB be a base association between the sets A and B and let a be any member of A. The **degree of a for AB at a given time** is the number of members b of B for which <a, b>:AB; that is, the number of members of B that are AB associated with a at the given time. Similarly the **degree of b for AB at a given time** is the number of members a of A that are AB associated with b at the given time. Consider for example the situation illustrated in figure 3.1.1.

FIGURE 3.1.1

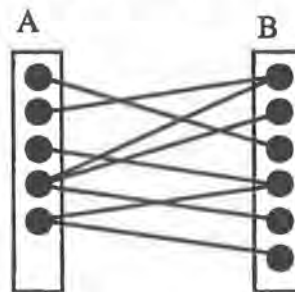


In this figure the members of A are represented by the small black circles located within the rectangle marked A, the members of B are represented by the small circles located within the rectangle marked B, and the members of AB are represented by the pairs of circles that are joined with a line. The figure gives a picture of the memberships of A, B, and AB at a particular given time. At that time the degrees of all the members of A are 1 since there is exactly one line leaving each member of A. Reading from top to bottom, the degrees of the members of B are respectively 2, 0, 1, 3, and 0.

Incidentally, the use of 'degree' in this context corresponds exactly with the use of the term in graph theory. Figure 3.1.1 illustrates what is called a bipartite graph with nodes that are members of the sets A and B and with undirected edges joining a member of A with a member of B. In graph theory terminology, the degree of a node is the number of edges connected to the node.[Ore, Oystein, Theory of Graphs, American Mathematical Society, 1962].

Since the association AB is a base set, it will have a time dependent extension so that the degree of any particular a or b may change with time. In figure 3.1.2 the memberships of A, B, and AB are illustrated at another later time.

FIGURE 3.1.2



Since the time of figure 3.1.1, the membership of A has decreased by 1, the membership of B increased by 1, and the membership of AB has been changed but not altered in number. Reading from top to bottom, the degrees of the members of A are 1, 1, 1, 3, and 2, and the degrees of the members of B are 2, 1, 2, 1, 1, and 1.

3.2. Business Practices and Degrees

Business practices for an enterprise such as the XYZ corporation often dictate that the degrees of entities for an association are limited in some fashion. For example, in the XYZ corporation an employee must be assigned to exactly one department. This means that for any employee e, there is exactly one department d for which <e,d>:EMPDEPT, so that the degree of e for EMPDEPT is always 1. On the other hand, an employee may be assigned to zero or more projects, which means that the degree of an employee for EMPPROJ may be any integer including zero.

The smallest possible degree that an element from A can have for the association AB is 0, while the largest might be any integer. The smallest possible degree that an element from A can have for AB is called the **lower degree of AB on A**. For example, the lower degree of EMPDEPT for EMPLOYEE is 1, while the lower degree of EMPPROJ for EMPLOYEE is 0. Similarly, the **upper degree of AB on A** , is the largest degree that an element from A can have for AB. For example, the upper degree of EMPDEPT on EMPLOYEE is 1, while the upper degree of EMPPROJ on EMPLOYEE may be any integer. The **lower and upper degrees of AB on B** are similarly defined.

Although the lower and upper degrees of an association on a set may take on any integer value, it is rarely necessary to know precisely what that value must be. It is generally sufficient to know whether the lower bound is 0 or is greater than 0, and whether the upper bound is 1 or greater than 1. Therefore the values of the lower degrees will be restricted to being either 0 or 1, and the upper degrees to being 1 or *, with the latter upper degree meaning 'unrestricted'. Hence if AB has a lower degree of 1 on A, then at all times every member of A is AB associated with at least one member of B, although a member may be AB associated with more than one member. If it has an upper degree of * on A , then at some time there may be a member of A that is AB associated with more than one member of B; if it has an upper degree of 1 on A, then at no time may a member of A be AB associated with more than one member of B.

Consider figure 3.1.1 for example. The degree of AB for each member of A is 1 at the time for which the figure illustrates the memberships. If a business practice dictates that that should be the case at all time, then both the lower and upper degrees of AB on A are 1. However, the fact that the degree of AB for each member of A is 1 may be just an accident of time, so that from the figure alone the upper and lower degrees of AB on A cannot be determined. On the other hand, the lower and upper degrees of AB on B can be determined. The smallest of the degrees of AB for members of B is 0, and the largest is 3. Since at one moment of time the smallest of these degrees is 0, the lower degree of AB on B is 0, and since at one moment of time the largest of these degrees is 3, which is greater than 1, the upper degree of AB on B is *.

From the figure 3.1.2, on the other hand, the lower degree of AB on neither A nor B can be determined; business practices must dictate the lower degrees . The upper degrees of AB on A and B, on the other hand, can both be determined to be *.

3.3. Notation for Degree Declarations

In the declaration of a base set AB that is an association with domain AxB, two pairs of degrees must be supplied, the lower and upper degrees of AB on A, and the lower and upper degrees of AB on B. The format used for the <u>degree declaration</u> clause of a declaration is

<ldA,udA>,<ldB,udB>,

where ldA and udA are respectively the lower and upper degrees of AB on A, and ldB and udB the lower and upper degrees of AB on B. For example,from the degree declaration '<1,1>,<1,*>' for EMPDEPT can be concluded that the lower and upper degrees of EMPDEPT on EMPLOYEE are 1, and that the lower and upper degrees on DEPARTMENT are 0 and *. They express that an employee must be assigned to exactly one department, and that every department must have at least one employee assigned to it.

The degree declaration for EMPPROJ is blank, which has the same meaning as the degree declaration '<0,*>,<0,*>'. The rationale for this interpretation of a blank degree declaration is the following: A lower degree of 0 and an upper degree of * does not constrain in any way the degrees of entities for an association, for the degree must always be a nonnegative integer, and may be any nonnegative integer. Therefore if a blank degree declaration is to mean that the degrees are unconstrained, then it has the same meaning as the degree declaration '<0,*>,<0,*>'.

The degrees of an association can label the arrows of a domain diagram, so that the diagram can provide additional information. For example, in figure 3.3 the degrees of the association EMPDEPT label the arrows of the domain diagram of figure 2.1.

FIGURE 3.3

EMPLOYEE <1,1> <1,*> DEPARTMENT

EMPDEPT

This device will be frequently employed.

Because the lower degree 0 and the upper degree * do not enforce any constraints on the membership of an association, they should not be declared if the tighter lower degree 1 or tighter upper degree 1 can be declared. Failure to declare the tightest possible bounds results in a larger burden being placed on the users of the resulting database. They must insure in updating the membership of an association that the business practices of the enterprise are respected. The importance of declaring the tightest degrees possible will become evident in later chapters.

3.4. Partial, Total, Singlevalued, Multivalued, and Functional Associations

Definitions related to lower and upper degrees are useful as shorthand: An association AB between A and B is said to be **partial** on A if its lower degree on A is 0, **total** on A if its lower degree on A is 1, to be **singlevalued** on A if its upper degree on A is 1, and is said to be **multivalued** on A if its upper degree on A is *. It is said to be **functional** on A if it is total and single valued on A. Definitions of partial, total, singlevalued, multivalued , and functional on B, are defined similarly.

The association EMPDEPT is total on EMPLOYEE and on DEPARTMENT. It is singlevalued on EMPLOYEE, but multivalued on DEPARTMENT. EMPPROJ on the other hand is neither total nor singlevalued on either EMPLOYEE or PROJECT.

3.5. Interpretation of Degrees of an Association with Domain an Association

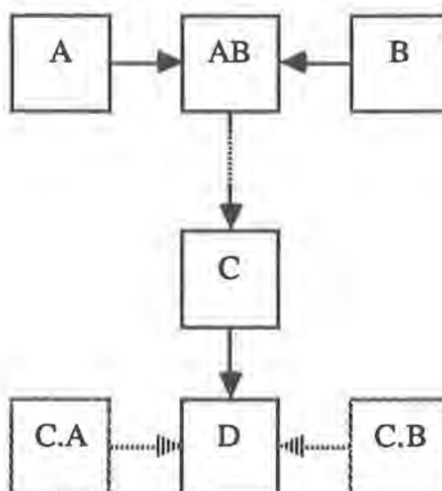
The degrees of an association, which has another association as its domain, can, when properly interpreted, express subtle but important facts about business practices. For example, the domain of LEADER is EMPPROJ, while its arity domain is EMPLOYEExPROJECT. The degrees of LEADER are declared to be <0,*> and <1,1>. Although the arity domain of Leader is a cartesian product, its domain is not, so that it is not clear how the degrees of LEADER are to be interpreted. The degrees can encode different business practices depending on how they are interpreted.

Consider first the simplest interpretation, for which the degrees are interpreted relative to the sets EMPLOYEE and PROJECT, the sets whose cartesian product is the arity domain of LEADER. Under this interpretation the degrees <0,*> of LEADER on EMPLOYEE express that not every employee is the leader of a project, and that an employee can be leader of any number of projects. The degrees <1,1> of LEADER on PROJECT express that every project has a unique leader, so that in fact no project can be without employees. Indeed, the lower degree 1 of LEADER on PROJECT amounts to a tightening of the lower degree 0 of EMPPROJ on PROJECT. Under this simplest interpretation of the degrees, therefore, every project has a leader and has at least one employee assigned to it. If that is the business practice of the XYZ corporation, then the lower degree of EMPPROJ on PROJECT should be declared to be 1, rather than 0.

There is a second interpretation of the degrees, however, under which they express a different business practice, namely that a project need not have an employee assigned to it, but if it does, then it must have a unique leader assigned to it; that is, a leader must be the first employee assigned to a project. This interpretation is obtained if the degrees are understood to be relative to the set of projects to which employees have been assigned, rather than relative to PROJECT.

The set of projects to which employees have been assigned is called the **projection** of EMPPROJ on PROJECT, and is denoted by EMPPROJ.PROJECT. Similarly, the set of employees that have been assigned to projects is the projection of EMPPROJ on EMPLOYEE and is denoted by EMPPROJ.EMPLOYEE. This is illustrated for a more general situation in the next figure:

FIGURE 3.5



The base association D has domain C. Between AB and C may be zero or more other associations each with the previous association as its domain. The arity domain of D and of C is therefore AxB. The degrees of D are interpreted relative to the projections C.A and C.B of its domain on the sets A and B forming the cartesian product AxB that is its arity domain. A, B, C, and D, in this figure could be EMPLOYEE, PROJECT, EMPPROJ, and LEADER. C.A is then the employees assigned to projects and C.B the projects to which employees have been assigned.

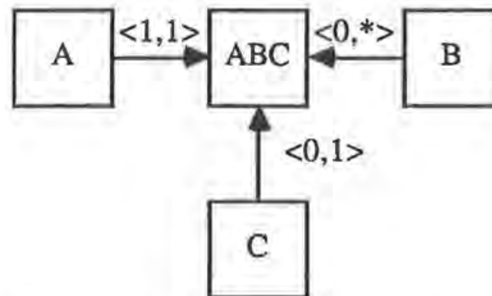
The sets A, B, C, and D, in the figure might also be EMPLOYEE, DEPARTMENT, EMPDEPT, and MANAGE, with C.A the employees assigned to departments and C.B the departments to which employees have been assigned. Since the lower degrees of EMPDEPT on EMPLOYEE and DEPARTMENT are both 1, the projection EMPDEPT.EMPLOYEE of EMPDEPT on EMPLOYEE has the same extension as EMPLOYEE and the projection EMPDEPT.DEPARTMENT of EMPDEPT on DEPARTMENT has the same extension as DEPARTMENT. Therefore it does not matter whether the degrees <0,1> and <1,1> declared for MANAGE are interpreted relative to the projections EMPDEPT.EMPLOYEE and EMPDEPT.DEPARTMENT or relative to EMPLOYEE and DEPARTMENT. Under either interpretation the degrees express that not every employee is a manager of a department, that an employee may manage at most one department, and that every department has one and only one manager. Because EMPDEPT is the domain of MANAGE, necessarily the manager of a department is assigned to the department.

Undeclared sets such as C.A and C.B in figure 3.5 may be included in domain diagrams. It is necessary to include them if the degrees of an association such as D are to label arrows of the diagram as suggested in section 3.3, since these degrees label the arrows from the projections to the association.

3.6. Degrees of Nonbinary Associations

Ternary and quadernary associations were illustrated in figure 1.4.2. These are associations with domain the cartesian product of three, respectively four, sets. For example, the relationship between an association ABC with domain AxBxC and its immediate domain predecessors A, B, and C, is illustrated in the domain diagram of figure 3.6.1. Degrees of ABC on each of its immediate domain predecessors label the arrows of the diagram.

FIGURE 3.6.1



The members of ABC are triples <a,b,c> for which a:A, b:B, and c:C. The lower degree 1 of ABC on A means that

for each a:A, there is a <b,c>:BxC such that <a,b,c>:ABC.

The upper degree 1 of ABC on A means that

for each a:A, there is at most one <b,c>:BxC for which <a,b,c>:ABC.

Similarly, the degrees <0,*> of ABC on B mean that

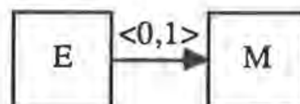
for each b:B, there may be zero or more <a,c>:AxC for which <a,b,c>:ABC.

The degrees <0,1> of ABC on C mean that

for each c:C, there is at mose one <a,b>:AxB for which <a,b,c>:ABC.

If the set MALE is declared to have EMPLOYEE as its domain, EMPLOYEE is an immediate domain predecessor of MALE, but it is the only one. The relationship between MALE (M) and its only immediate domain predecessor EMPLOYEE (E) is illustrated in figure 3.6.2.

FIGURE 3.6.2



The degrees <0,1> of M on E mean that

for each e:E, there is at most one m:M for which e=m.

This is to be expected unless the XYZ corporation has a policy of not hiring females, in which case the degrees should be <1,1>. But then it would be unnecessary to declare M since it would have the same extension as E.

An upper degree of * declared for M on E would not be a tight bound and can therefore be prohibited. Therefore the only reasonable degrees to declare for M on E are <0,1>. For this reason, degrees are never declared for sets with a single immediate domain predecessor. For example, EMPPROJ is the only immediate domain predecessor of LEADER, so no degrees are declared for LEADER on EMPPROJ. The two pairs of degrees declared for LEADER are the degrees on the projections EMPPROJ.EMPLOYEE and EMPPROJ.PROJECT.

4. VALUE SETS and DEFINED SETS

Only base sets have been declared so far. They have been called "base" since they are the foundation sets upon which databases are built. They may be either primitive, like EMPLOYEE, DEPARTMENT, and PROJECT or nonprimitive like EMPDEPT, MANAGE, EMPPROJ, and LEADER. The intension of a base set is expressed in a language and style of the declarer's choice in the comment part of its declaration. As a consequence only humans can determine what entities are members of a base set. But one of the purposes of a database is to use machines to record information about the membership of such sets. But in order for a machine to assist in recording such information, some means must be provided for humans to represent the information in a machine readable and writeable form.

4.1. Strings and Primitive Value Sets

The primary means by which information is passed between humans and machines is through the reading and writing of strings of characters. Strings of machine readable and writeable characters can appear as values of fields of records that a computer can store and process internally, and can print in some human readable form. Humans can write such strings in a way that a computer can read them.Therefore the declaration of sets of such strings, called **value sets**, is essential for every information system. Value sets are distinguished from base sets such as EMPLOYEE, DEPARTMENT, and EMPDEPT primarily by the fact that their members can be recognized and manipulated by a computer.

A primitive set is one whose membership is not drawn from some previously declared set; a primitive set has no domain, or at best it can be thought of as its own domain. Primitive value sets are essentially the basic data types of a programming language. For the purposes of this book, the primitive value sets will be taken to be STRING, INTEGER,and REAL. The precise intensions of these sets need not be stated since they are "built into" the programming language used to support any information system. It is sufficient to know that STRING is the set of all strings of machine readable and writeable characters, that INTEGER is the set of integer number representations, and that REAL is the set of real number representations.

4.2. Declarations of Defined Sets

Primitive value sets are the first examples of **defined** sets. Unlike base sets, the membership of defined sets can be determined by a machine. Like base sets, however, they must be declared. The declaration of a defined set takes the form:

setnm for {domain+variable declaration | assertion | comment }.

It differs from the declaration of a base set in two ways: First the domain declaration clause of a base set is replaced with the domain+variable declaration clause which may never be blank, and second, the degree declaration clause of a base

set is replaced with an assertion clause that is either blank or an assertion of the language DEFINE described in chapter 3. A nonblank assertion clause, together with the domain+variable declaration clause, describes the intension of the set in a machine interpretable form. As with base set declarations, the comment clause is a statement in any language or form of a users' choosing and is used to describe the intension of the set informally.

The formulation of the intension of a defined set in the language DEFINE requires greater care and precision than its expression for human interpretation only. Further its precise form is often not needed until the last stages of an information system design. Good system design therefore dictates that the task of stating the intension of a defined set in the language DEFINE be delayed as long as possible. For this reason, assertion may be left blank and completed when required. A declaration with assertion blank can still be recognized as the declaration of a defined set by the variable declarations appearing in the first clause of the declaration. For example, declarations for the primitive value sets are:

STRING for {x:STRING || the finite sequences of characters recognized by the system},
INTEGER for {x:INTEGER || the integers recognized by the system},
REAL for {x:REAL || the real numbers recognized by the system}, and
BOOLEAN for {x:BOOLEAN || the boolean truth values recognized by the system}.

Since the membership of these sets is taken to be "built into the system", it will never be necessary to complete the blank assertion clause; nevertheless the fact that the variable 'x' is declared along with the domain in the first part makes these declarations recognizable as declarations of defined sets. The fact that they are declared to be their own domain in the domain+variable declaration clause means that they are declarations of primitive sets.

4.3. Defined Value Sets

The primitive value sets are rarely used directly to record information about entities; defined subsets of them are used instead. For example, the set of strings of digits of length 5, or the strings of characters of length 20 are defined value sets. Another example of a set that may be declared as a defined value set is

COLOUR for {x:STRING| x='RED' **or** x= 'ORANGE' **or** x='YELLOW' **or** x='GREEN' **or** x='BLUE' **or** x='INDIGO' **or** x='VIOLET'}.

Here '**or**' is the usual boolean disjunction of logic. The members of the set are strings that are the names of the basic colours. An example of a ternary value set is the following:

VDATE for {x:INTEGER, y:INTEGER, z:INTEGER|| a value set for dates}.

This particularly broad declaration of a date value set permits freedom of choice for later specialization to whatever format is desired. The assertion clause is left blank because the choice of a particular format to be used for dates in the database need

not be made until very late in the design of the database. If a more detailed declaration is desired, value sets VDAY, VMONTH, and VYEAR can be declared and VDATE declared to be

VDATE for {x:VDAY, y:VMONTH, z:VYEAR|| a value set for dates}.

The names of these value sets all begin with the letter 'V' as a reminder that the set is a value set. There is no requirement to do so, VDATE could just as well be given the name 'DATE', but discussions are often clarified when value sets are given distinctive names. This is especially so when associations are given names similar to value sets, a common practice in database design.

Because the committment to a particular format for the members of a value set need not be made until very late in the design of a database, the format should in the early stages be only loosely described. For example, to postpone all detailed decisions on the format of dates, VDATE can be declared to be

VDATE for {x:INTEGER|| a value set for dates}.

It is this declaration that will be assumed for most of the further discussions. Similarly, although strings representing addresses will unquestionably be highly structured, and probably defined in terms of value sets such as VSTREET# , VSTREET, VCITY, VPROV, and VPOSTALCODE, initially it is sufficient to give a declaration such as

VADDRESS for {x:STRING|| a value set for addresses}.

As these examples illustrate, value sets may be primitive defined as with STRING, or defined with domain a previously declared value set as with COLOUR or defined with domain the cartesian product of previously declared value sets, as with one declaration of VDATE and of VADDRESS. They must be declared before information can be recorded in a data base. For example, it is not possible to list the members of the set DEPARTMENT without having some way of them in a machine readable and writeable way. A possible value set for this purpose is:

DEPTNAME for {x:STRING | x='FINANCE' **or** x='SHIPPING' **or**
x='PERSONNEL' **or** x='PURCHASING' | a value set for deptname}.

Here specific strings have been selected to name the existing departments, as was done in the case of COLOUR. But such a value set would be a poor choice since it would complicate the changing of names of departments and the adding and removal of departments. A better value set of wider use would be

VNAME for {x:STRING | {x:L:} ≤ 20 | a value set for names},

where the notation '{x:L:}' expresses the length of x, that is, the number of characters in the string. DEPTNAME is a subset of VNAME, while VNAME might be used as a source of names for employees for example.

Because of the uncertainty as to what value sets will actually be needed, however, it may be better to use neither of these declarations at first and simply declare

VNAME for {x:STRING || a value set for names}.

The blank second part of the declaration can be completed when necessary, while

the comment gives sufficient information about the use of the set.

5. ATTRIBUTES

To attribute a property or characteristic to an entity is to say that the entity has the property or characteristic. For example, to attribute maleness to John Smith is to say that he is male, or to attribute the address '1783 Sunset Drive' to him is to say that that his address. Attributes are the means by which such attributions may be made.

An **attribute** of a set A is an association AV between the set A and a value set V. Thus the domain of AV may be AxV or VxA; however, the convention will be followed of having the set A of which AV is an attribute as the first set in the cartesian product so that the domain of AV is taken to be AxV. The set V is called **the value set of the attribute**. A **value** of AV for a member a of A, is a member v of V for which <a,v>:AV; it is from this use of 'value' that value sets get their name. An attribute on A is said to be **partial, total, singlevalued, multivalued**, and **functional**, if it is respectively total, single valued, multivalued, and functional association on the set A. Should AV be functional, then for each member a of A there is a unique member v of V for which <a,v>:AV ; that value is referred to as **the value** of AV for a.

5.1. Functional Attributes

Functional attributes are by far the commonest form of attribute, and it is common practice to refer to a functional attribute as just an attribute, and refer to nonfunctional attributes as partial or multivalued attributes. This practice will be followed in this book.

An example of an attribute is the following name attribute.
NAME for {EMPLOYEExVNAME |<1,1>,<0,* >| name attribute for employee}.
NAME associates with each member of EMPLOYEE the string in the value set VNAME that represents the name of the employee. Note that NAME is not a value set since EMPLOYEE is not a value set. It is an attribute of EMPLOYEE, and in particular a functional attribute. The value of NAME for a member e of EMPLOYEE, is the unique member nm of VNAME for which the pair <e,nm> is a member of NAME.

Similarly an attribute that associates an address with each employee can be declared:
ADDRESS for {EMPLOYEExVADDRESS |<1,1>,<0,* >| address attribute of employee}.
It does not matter for either of these declarations what particular format for VNAME and VADDRESS is finally chosen, NAME and ADDRESS will not have to

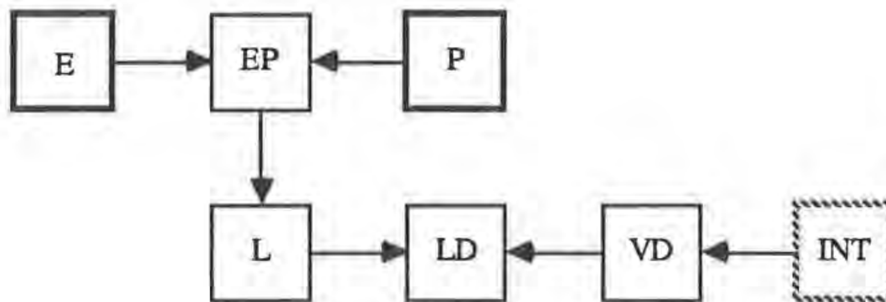
be redeclared if these value sets are redeclared.

Attributes can be declared on any set, including associations. For example, it is necessary to record the date on which the leader of a project is assigned to the project since it is the date of the first serious committment to the project. The appropriate attribute is therefore

LEADERDATE for {LEADERxVDATE |<1,1>,<0,*>| date leader assigned to project}

This means that a member of LEADERDATE consists of a pair <<e,p>,d> with its first element <e,p> a member of LEADER. The domain diagram of figure 5.1 illustrates the declaration of LEADERDATE.

FIGURE 5.1



Although functional attributes are the commonest, an attribute need neither be total nor single valued, as with the following attribute of PROJECT.

REVIEWDATE for {PROJECTxVDATE || dates for project review}.

By the convention for declarations of base sets, the blank second part of the declaration means the same thing as the degree declaration '<0,*>,<0,*>'. Therefore review dates are required for only some of the projects, and any number of review dates may be required.

5.2. Identifiers

A name can be used to identify an entity if the name is unique to the entity. For example 'public relations' would identify a department of the XYZ corporation if there was only one department with that name. The public relations department might have more than one name, for example, it might also be known as the community relations department, but still 'public relations' and 'community relations' will identify the same department as long as no other department has been given one of these names. Assume now that a name attribute NAME with value set VNAME is to be declared that will provide names for members of DEPARTMENT that identify them:

NAME for {DEPARTMENTxVNAME | <u>degree declaration</u> | name of dept identifies it}.

The degrees of NAME on VNAME must clearly be declared to be <0,1>, the lower degree 0 because not every member of VNAME will be the name of a department, and the upper degree 1 because if a member of VNAME is the name of a department, it should be the name of exactly one department. With these degrees declared it would be possible for a department to have more than one name.

Although it is convenient to permit more than one name for a department, especially for name changes, there are other advantages to insisting that a department have a unique name that arise during the processing of data. For this reason it is common practice to require that the members of a declared set be identifiable with values of an attribute that is functional on the set. Thus the NAME attribute for DEPARTMENT is declared

NAME for {DEPARTMENTxVNAME |<1,1>,< 0,1>| dept has unique name}.

The degrees <1,1> of NAME on DEPARTMENT ensure that every department has a single name assigned to it, while the degrees <0,1> ensure that a name is assigned to exactly one department. There is, therefore, a one-to-one correspondance between departments and their names.

An attribute such as NAME is an **identifier** of DEPARTMENT, since it is functional on DEPARTMENT and singlevalued on VNAME. The NAME attribute of EMPLOYEE, on the other hand, is not an identifier of EMPLOYEE since two employees may have the same name.

An attribute AV with value set V on a set A that is an identifier of A associates a unique member v of V with each member a of A. Because V is a value set, v can be written and read by both humans and machines and can therefore be used to identify a in fields or records.

Incidentally, although there are now two sets with the same name 'NAME', they can be distinguished by their domains. One is an attribute of EMPLOYEE while the second is an attribute of DEPARTMENT. A third set with the same name is also declared as an identifier for PROJECT:

NAME for {PROJECTxVNAME |<1,1>,< 0,1>| a proj has a unique name}.

Every primitive base set requires an identifier. It is the practice of the XYZ company to identify employees with 5 digit integers. Therefore a value set for the employee number identifier of EMPLOYEE is

VEMP# for {x:INTEGER | $10000 \leq x \leq 99999$ | a value set for emp#},

and an identifier is

EMP# for {EMPLOYEExVEMP# |<1,1>,< 0,1>| emp has unique emp#}.

Although every primitive base set requires an identifier, a value set does not since its members can be read and written by humans and machines, and therefore identify themselves. Indeed any set of machine readable and writeable strings,

whether it is a value set or not, needs no identifier. However all such sets may have attributes declared for them, and the attributes may include identifiers. For example, a set of very long strings, such as the chapters in this book, can be provided with an identifier that represents the long strings by short ones, such as the chapter names.

6. PARTITIONS of DECLARED SETS and DOMAIN PREDECESSORS

A set is declared to be either primitive or nonprimitive, and either base or defined, so that there are four fundamental kinds of sets: primitive base, nonprimitive base, primitive defined, and nonprimitive defined. The only examples of defined sets introduced so far, primitive or nonprimitive, have been of value sets. But not all defined sets are value sets, as one example in section 6.1 illustrates. A summary of the relevance of the four kinds of sets is provided in section 6.2, and a guiding principle to be used in choosing base sets is stated in section 6.3. Finally in section 6.4 the important domain predecessor between declared sets is defined.

6.1. Nonvalue Defined Sets

The XYZ corporation finds it necessary to record the sex of its employees. The value set and attribute for sex are declared:

VSEX for {x:STRING | x='MALE **or** x='FEMALE' | value set for sex attribute}

and

SEX for {EMPLOYEExVSEX |<1,1>,<0,*>| sex attribute}.

The set of male employees has EMPLOYEE as its domain, and it can be declared as a base set

BMALE for {EMPLOYEE || male employees}

or it can be declared as a defined set

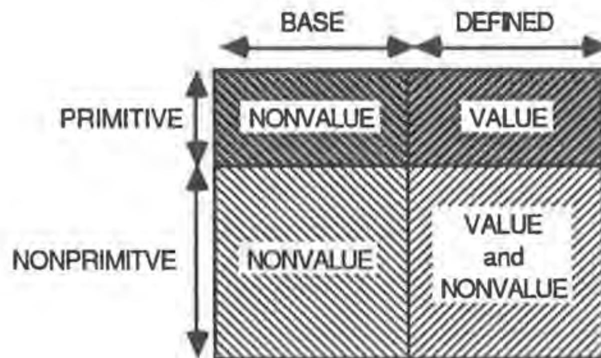
MALE for {x:EMPLOYEE | <x,'MALE'>:SEX | male employees}.

Many other examples of defined sets that are not value sets will be given in chapter 3 where the language DEFINE is described.

6.2. Two Partionings of Declared Sets

The primitive value sets are the only examples of primitive defined sets introduced so far, so that the kinds of sets introduced can be summarized in the diagram of figure 6.1.

FIGURE 6.1



The large square consisting of four smaller rectangles represents all possible declared sets. A declared set is either primitive or nonprimitive, and is either base or defined. The horizontal line through the square represents the partitioning of the sets into primitive and nonprimitive, while the vertical line represents the partitioning into base and defined. A primitive set is one declared without a domain, or equivalently declared with itself as its domain, while a nonprimitive set has the cartesian product of one or more previously declared sets declared as its domain. Base sets have intensions expressed in human understandable terms only, while defined sets have intensions expressed in a machine interpretable language such as DEFINE. A value set is a defined set of machine readable and writeable strings of characters that is either primitive, or has the cartesian product of one or more previously declared value sets as its domain. A nonprimitive base set on the other hand may have the cartesian product of any previously declared sets as its domain. Nonprimitve defined sets may be value or nonvalue sets. But for the present, all primitive defined sets declared must be value sets. The restriction is introduced now to avoid unnecessary complications at this stage. In a later chapter dealing with extensions to the language DEFINE, the consequences of removing the restriction will be fully discussed.

As noted before, one consequence of restricting the primitive defined sets to being built in value sets is that sets of mixed arity cannot be declared; for example, a set cannot have both a pair and a triple as a member.

6.3. Choosing Base Sets

The primitive defined sets are assumed to be declared prior to the information needs analysis of an enterprise, and the choice of them has little relevance for an information needs analysis. But the choice of the primitive base sets is fundamental to a successful analysis. For each nonprimitive set, base or defined, has one or more primitive sets as domain predecessors. But the choice of all the base sets, primitive or nonprimitive, is important for a successful information neeeds analysis for an enterprise. In chapter 4 some guiding principles for the choice of base sets are

described and justified. But one such principle should be evident already.

A goal of an information needs analysis is the construction of a set schema with as few coherent base sets as are necessary for the needs of the enterprise. For as will be seen, users of a management system must maintain the membership of all base sets, while the system itself can maintain the membership of defined sets. For example, users must maintain the membership of SEX, but the system can maintain the membership of MALE. Therefore to ensure that as few base sets as possible are declared, a set that can be declared as a defined set should never be declared as base. For example, two declarations were given above for the set of male employees, one as the base set BMALE and one as the defined set MALE. Clearly the declaration as MALE is to be preferred over the one as BMALE. For one unfortunate consequence of choosing BMALE would be that the users of the system would not only have to maintain the membership of SEX, but would also at the same time maintain the membership of BMALE and ensure that at all times it is the same as the membership of MALE. This is not only an unnecessary burden for the users, but also errors are very likely to occur. A guiding principle for the choice of base sets should be, therefore:

Don't declare a set as base if it can be declared to be defined.

Other such principles will be given in chapter 4.

6.4. Domain Predecessors

The fact that the domain of a declared set is the cartesian product of one or more previously declared sets is of fundamental importance. For it means that the declared sets of a set schema for an enterprise have a natural ordering. For example, the sets EMPLOYEE and PROJECT had to be declared before the set EMPPROJ since it has EMPLOYEExPROJECT as its domain; they immediately precede EMPPROJ in the order of declarations.

A declared set A is an **immediate domain predecessor** of a set B, if the domain of B is the cartesian product of one or more sets which includes A.

So EMPLOYEE and PROJECT are immediate domain predecessors of EMPPROJ. But also LEADER and VDATE are immediate domain predecessors of LEADERDATE, while EMPPROJ is an immediate domain predecessor of LEADER. Thus each of EMPLOYEE, PROJECT, EMPPROJ, LEADER, and VDATE precede LEADERDATE in the order of declarations.

A declared set A is a **domain predecessor** of a declared set C, if A is an immediate domain predecessor of C, or if there is a declared set B such that A is a domain predecessor of B and B is a domain predecessor of C.

This definition of domain predecessor is typical of a simple but important class of

recursive definitions. Domain predecessor is the **transitive closure** of immediate domain predecessor.

7. A SET SCHEMA for the XYZ CORPORATION

During an information needs analysis of an organization, many sets will be declared. The collection of all declared sets for an organization is called a **set schema** for the organization. In this section a simple set schema for the XYZ corporation is described to illustrate some of the concepts introduced. Rather than declaring seperately each of the sets recognized during an information needs analysis for the XYZ company, they are simply listed in the table of figure 7.1. This table will be referred to many times as a source of examples. The table is not a substitute for full declarations since, for example, no assertion clauses fre listed for defined sets. It also does not include the primitive value sets. By the database schema of figure 7.1 will be meant the schema with STRING and INTEGER, as well as all the sets of the table.

FIGURE 7.1

DECLARED SET TABLE

NAME	DOMAIN	COMMENT
EMPLOYEE		current employee
DEPARTMENT		currently approved department
EMPDEPT	EMPLOYEExDEPARTMENT	emp assigned to dept
VNAME	STRING	value set for name attributes
NAME	EMPLOYEExVNAME	name attribute for employee
NAME	DEPARTMENTxVNAME	name is identifier for department
VEMP#	INTEGER	value set for employee numbers
EMP#	EMPLOYEExVEMP#	emp# is identifier for employee
VSEX	STRING	value set for sex attribute
SEX	EMPLOYEExSEX	sex attribute for employee
VADDRESS	STRING	value set for address attributes
ADDRESS	EMPLOYEExVADDRESS	address attribute for employee
MANAGE	EMPDEPT	manager assigned to dept
PROJECT		a current or planned project
NAME	PROJECTxVNAME	name is identifier for project
EMPPROJ	EMPLOYEExPROJECT	emp assigned to proj
LEADER	EMPPROJ	leader of proj
VDATE	INTEGER	value set for date attributes
REVIEWDATE	PROJECTxVDATE	some projects have review dates date
LEADERDATE	LEADERxVDATE	date leader is assigned to project
REVIEWDATE	PROJECTxVDATE	dates for project review
MALE	EMPLOYEE	male employees
FEMALE	EMPLOYEE	female employees
SOCCER	MALE	players on all male soccer team
BASEBALL	EMPLOYEE	players on baseball team

Each row of the table of figure 7.1 corresponds to the declaration of a single set.

The order in which the rows are listed is not completely arbitrary. The row, corresponding to a set that is a domain predecessor of a second set, is listed before the row corresponding to the second set. This of course does not completely specify the order of the rows; the order of the rows listed in the table is just one of many possible orders in which the sets could be declared.

The degrees of the base sets that are associations are given in the companion table of figure 7.2. In that table by 'left set' and 'right set' is meant the first and second sets appearing in the cartesian product that is declared as the domain of the set.

FIGURE 7.2

ASSOCIATION TABLE

ASSOCIATION	LEFT SET	LDEGREE		RIGHT SET	RDEGREE	
		LWR	UPR		LWR	UPR
EMPDEPT	EMPLOYEE	1	1	DEPARTMENT	1	*
EMP#	EMPLOYEE	1	1	VEMP#	0	1
NAME	EMPLOYEE	1	1	VNAME	0	*
ADDRESS	EMPLOYEE	1	1	VADDRESS	0	*
SEX	EMPLOYEE	1	1	VSEX	0	*
NAME	DEPARTMENT	1	1	VNAME	0	1
MANAGE	EMPDEPT. EMPLOYEE	0	1	EMPDEPT. DEPARTMENT	1	1
NAME	PROJECT	1	1	VNAME	0	1
EMPPROJ	EMPLOYEE	0	*	PROJECT	0	*
LEADER	EMPPROJ. EMPLOYEE	0	*	EMPPROJ. PROJECT	1	1
REVIEWDATE	PROJECT	0	*	VDATE	0	*
LEADERDATE	LEADER	1	1	VDATE	0	*

Each of the tables of figures 7.1 and 7.2 is an example of a common device used to record infomation. A very important prerequisite to database design is the understanding of the precise meaning of such tables. The language DEFINE, described in the next chapter, provides the means for defining the intension of a table in terms of declared sets and associations, while all of chapter 5 is devoted to the design of tables from set schemas.

Copyright Paul C Gilmore, all rights reserved,

8. DOMAIN GRAPHS and TREES

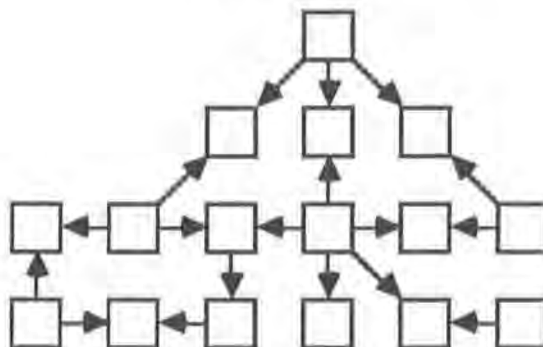
Domain diagrams have been used to illustrate the relationships between the domains of declared sets. They consist of boxes labelled with declared sets and one or more arrows directed between some of the boxes. An arrow is directed from one box to another if the first box is labelled with an immediate domain predecessor of the set labelling the second. A domain diagram is an illustration of a directed graph called a domain graph defined in section 8.2 after an introduction to directed graphs in section 8.1. In section 8.3 further definitions are provided and applied in section 8.4 where domain trees are defined and illustrated. Finally in section 8.5 the arrows of domain graphs and trees are labelled with the degrees of associations defined in section 3.

8.1. Directed Graphs

A **directed graph** is defined by a set of entities called **nodes**, and a set of pairs <<n1,n2>,m>, where <n1,n2> is a pair of distinct nodes called an **edge**, and m is an integer, $m \geq 1$, called the **multiplicity** of the edge. The edge is said to be **directed** from the first node n1 of the pair, called the **tail** of the edge, to the second node n2 of the pair, called the **head** of the edge; it is said to **connect** the two nodes that are elements of it.

Figure 8.1. is an illustration of a directed graph with edges all of multiplicity 1.

FIGURE 8.1



Ignoring the sets labelling the nodes, domain diagrams are also illustrations of directed graphs. For example, figure 2.1 illustrates a directed graph with three nodes, with edges of multiplicity 1 directed from the nodes labelled with EMPLOYEE and with DEPARTMENT to the node labelled with EMPDEPT. Figure 5.1 is similarly an illustration of a directed graph with edges of multiplicity 1. Figure 1.4.3 illustrates a directed graph with two nodes and a single edge of multiplicty 2.

Edges of multiplicity greater than 1 are uncommon, so the multiplicity of an edge will be assumed to be 1 unless a higher multiplicity is specifically stated.

A **head node** of a directed graph is a node that is the head of some edge, while a **tail node** is a node that is the tail of some edge. Nodes that are not head nodes are called **bottom** nodes because all edges connected to them are directed away from them. The directed graph illustrated in figure 8.1 has 5 bottom nodes. Nodes that are not tail nodes are called **top** nodes because all edges connected to them are directed towards them. The directed graph illustrated in figure 8.1 has 8 top nodes.

8.2. Directed Graphs as Domain Graphs

Domain diagrams are illustrations of directed graphs with nodes labelled with declared sets of a given set schema. A directed graph is a **domain graph for a set schema** if its nodes are labelled with declared sets so as to satisfy the following four conditions:

DG1. Each node of the graph is labelled with one declared set of the schema;

DG2. An edge is directed from one node to another if and only if the first node is labelled with an immediate domain predecessor of the second;

DG3. The multiplicity of an edge <n1,n2> for which a set A labels n1 and a set B labels n2, is the multiplicity of A as a domain predecessor of B; and

DG4. No set labels more than one node of the graph.

All of the domain diagrams appearing in figures so far have all been domain graphs for the set schema of figure 7.1, or of some unspecified schema. For example, figure 1.4.3 illustrates a domain graph for a set schema in which AxA is declared as the domain of AA.

It is important to note that cartesian products of sets do not label nodes of a domain graph. For example, although EMPLOYEExDEPARTMENT is the domain of EMPDEPT, it does not label a node of figure 2.1, only the immediate domain predecessors of EMPDEPT do.

Although primitive sets generally label bottom nodes of a domain graph, a nonprimitive set may also if none of its immediate domain predecessors label nodes of the graph. However no primitive set can label a head node of the graph.

For a database schema there is a single domain graph in which every set of the database schema labels a node. This domain graph is referred to as **the maximal domain graph** of the database schema, or just **the domain graph** for short. Every domain graph for a database schema is a **node subgraph** of the maximal domain graph; that is it is obtained from the domain graph by selecting a subset of the nodes, and selecting all edges between the selected subset of nodes.

Figure 8.1 is an illustration of a directed graph that can be converted into a domain graph by labelling its nodes with <ins>some</ins> of the declared sets in the table of figure 7.1.

The reader is urged to find a labelling of the nodes of this directed graph that satisfies the conditions DG1-DG4.

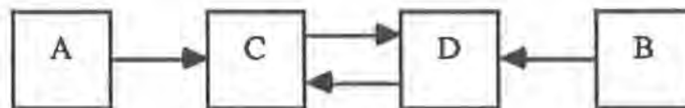
8.3. Paths and Cycles

Given a directed graph, a **directed path** of the graph is a sequence of nodes $n_1, \ldots, n_k$, where $k \geq 1$, such that $<n_i,n_{i+1}>$ is an edge of the graph for $1 \leq i \leq k$. The **length** of the path is k-1. For example, in figure 6.1 the nodes labelled E, EP, L, and LD, form a directed path of length 3; the nodes labelled with INT and VD form a path of length 1; and any one of the nodes forms a path of length 0.

A **directed cycle** is a path for which n_1 is the same node as n_k. A graph is **directed cyclic** if it has a directed cycle, and is otherwise called **directed acyclic.**

A domain graph is necessarily directed acyclic, since an immediate domain predecessor of a declared set must be declared before the set can be declared. Consider, for example, figure 8.3.1.

FIGURE 8.3.1



The path from the node labelled C to the node labelled D and back to the node labelled C is a directed cycle. For this to be a domain graph it would be necessary to declare AxD or DxA as the domain of C, so that D must be declared before C is declared. But it would also require that BxC or CxB be declared as the domain of D, so that C must be declared before D is declared. There can, therefore, be no set schema with the illustrated domain graph.

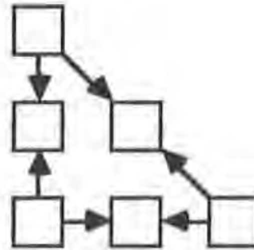
An **undirected path** of a directed graph is a sequence of nodes $n_1, \ldots, n_k$, where $k \geq 1$, such that either $<n_i,n_{i+1}>$ or $<n_{i+1},n_i>$ is an edge of the graph for $1 \leq i \leq k$. The **length** of the path is k-1. An **undirected cycle** is an undirected path for which n_1 is the same node as n_k. A directed graph is called **undirected cyclic** if it has an undirected cycle, and otherwise **undirected acyclic.** Necessarily each edge of an undirected acyclic graph has multiplicity 1, since two edges between the same two nodes would form a cycle of length 2.

~~Domain graphs, although always directed acyclic, are commonly undirected cyclic. For example, in figure 8.3.2 an undirected cycle of the directed graph of figure 8.1 is illustrated.~~

Copyright Paul C Gilmore, all rights reserved,

Domain graphs, although always directed acyclic, are commonly undirected cyclic. For example, in figure 8.3.2 an undirected cycle of the directed graph of figure 8.1 is illustrated.

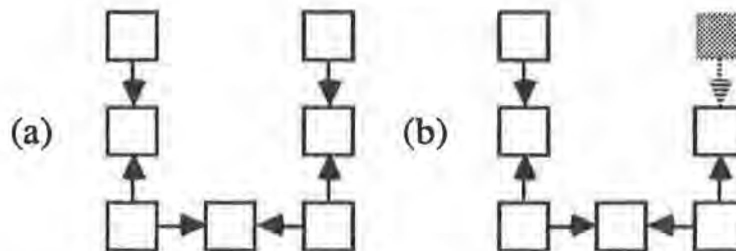
FIGURE 8.3.2



8.4. Trees from Undirected Cyclic Graphs

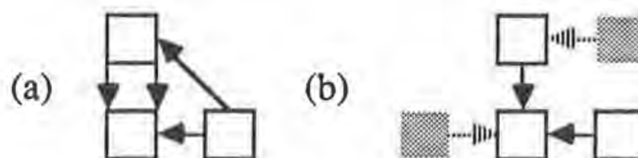
An undirected acyclic directed graph is called a **tree**. In figure 8.4.1 (a) a tree is illustrated that has been obtained from the directed graph of figure 8.3.2 by duplicating the uppermost node of the figure. In 8.4.1 (b) the same tree is illustrated but with the duplicate node, and the edge of which it is tail, drawn differently to emphasize the new node.

FIGURE 8.4.1



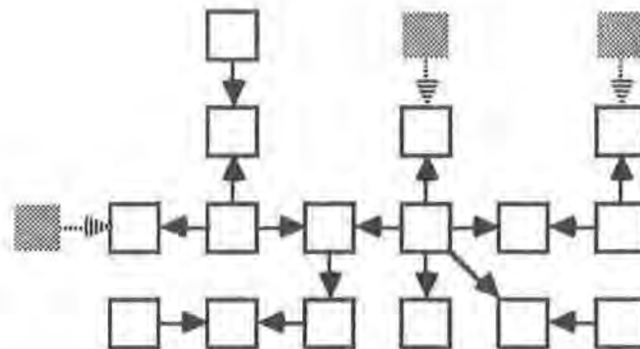
This method of obtaining a tree from a directed graph by breaking one or more undirected cycles will be often applied, and when it is, the tree will be illustrated as in figure 8.4.1 (b). For example, in figure 8.4.2 (a) an undirected cyclic graph is illustrated, while in (b) a tree obtained from it is illustrated. Two duplicate nodes are necessary in this case because two cycles had to be broken.

FIGURE 8.4.2



The tree illustrated in figure 8.4.3 has been obtained similarly from the directed graph illustrated in figure 8.1.

FIGURE 8.4.3



An important property of trees, not possessed by graphs, is that between any two nodes of the tree there is one and only one undirected path connecting them. This is a consequence of the tree being undirected acyclic. For if there were two paths between two nodes n1 and n2, then one path could be used to go from n1 to n2 and the second path used to go back to n1; that is, there would be an undirected cycle in the tree, which contradicts the definition of tree.

A **domain tree for a set schema,** like a domain graph, has its nodes labelled with declared sets of the schema, but the same set may label more than one node. The domain trees used in chapter 5 are all either domain graphs, or are obtained from domain graphs by breaking undirected cycles in the manner described.

8.5. Degrees for Edges of Domain Graphs and Trees

Base associations have degrees declared for them, one pair of degrees for each immediate domain predecessor of the association. These degrees have have been used as labels for the arrows of domain diagrams as described in sections 3.3 and 3.6. They can also label the edges of domain graphs and trees. Since degrees are not declared for defined sets, not all edges of a domain graph or tree can be labelled with declared degrees.

Degrees are not declared for defined sets because the system is to maintain the membership of defined according to the intension of the set stated in the language DEFINE. Business practices that affect the membership of defined sets must be expressed in the intensions of the sets. So it should be possible to calculate the degrees of defined sets on their immediate predecessors.

Some extensions of the language DEFINE so increase its power that it becomes impossible in general to calculate the degrees of defined sets. But for the form of DEFINE described in chapter 3, it is generally quite easy to calculate the degrees of

defined sets on their immediate domain predecessors. This is an important assumption for the methods of chapter 5 where it will be assumed that every edge of a domain graph or tree is labelled with its lower and upper degrees.

9. TYPES and ELEMENTARY ASSERTIONS

In the next chapter a description of the language DEFINE is given. Several definitions preliminary to that description are given here. Throughout the section schema will denote the current set of declared sets. For example, schema may be the set DSET of declared sets with members INTEGER, STRING, and the declared sets listed in figure 7.1. Examples illustrating the definitions will be given using DSET.

9.1. The Extended Schema schema* and Types

The members of schema are explicitly declared sets each with a distinct declaration. Cartesian products of declared sets are implicitly declared. It is useful to have a common way of referring to declared sets and the nested cartesian products of names of declared sets of a set schema. For this purpose schema*, the **extended schema**, is defined as follows:

1. Any member of schema is a member of schema*; and
2. the cartesian product of any number of sets from schema* is a member of schema*.

For example, if DSET is the set schema of figure 7.1, LEADERxVDATE is a member of DSET* as is also (EMPLOYEExPROJECT)xINTEGER. The latter is a nested cartesian product of primitive sets only and is an example of what is called a type, defined as follows:

Let pschema be the subset of schema consisting of primitive sets only. Then a **type** for schema is any member of pschema*.

For example, if schema is DSET, then pschema is the set with members

INTEGER, STRING, EMPLOYEE, DEPARTMENT, PROJECT

and a type for DSET is any one of these sets, or any cartesian product of these sets, or any cartesian product of a type for DSET.

The immediate domain predecessor and arity domain associations between members of a set schema schema have been previously introduced. One further association, the **type** association, is needed in preparation for chapter 3. It assigns a type to each member of schema* as follows:

1. The **type** of a primitive set is the set itself;
2. the **type** of the set A_1x ... xA_k is TA_1x ... xTA_k, where TA_i is the type of A_i; and
3. the **type** of a declared set A with domain B is the type of B.

For example, the type of LEADER is EMPLOYEExPROJECT, since the domain of

LEADER is EMPPROJ, the domain of EMPPROJ is EMPLOYEExPROJECT, and the type of EMPLOYEExPROJECT is itself. Similarly the type of LEADERDATE is (EMPLOYEExPROJECT)xINTEGER.

9.2. Terms and Elementary Associations

A **constant** is any member of a primitive value set. A **term** is a variable, a constant, or a tuple of terms. Examples of terms are

x, y, z, u, v, w, 'MALE', 123, <x, 'MALE' >,<x,123>, <<x,123>,'MALE'>, and <x,<x,'FEMALE'>>.

The expressions 'x:EMPLOYEE', and '<x,'MALE'>:SEX' that have been used earlier in the chapter are examples of **elementary assertions.** The general form of an elementary assertion for a set schema schema is

trm:setnm,

where trm is a term, called the **term of the assertion**, and setnm is the name of a set in schema*, called the **set of the assertion.**

A second permitted infix form of an elementary assertion

<trm1, trm2>:setnm,

is the expression

trm1:setnm:trm2.

However, since this is simply an alternative notation introduced for convenience, it is unnecessary to mention it seperately in most discussions relating to elementary assertions. Only in section 6 of chapter 3 dealing with parameterized set names will this form of elementary assertion reappear.

9.3. Type Assigning Elementary Associations

An elementary assertion trm:setnm is said to **assign** a member set of schema* to an occurrence of a constant or a variable in trm if

1. trm consists of a single constant or variable, and setnm is set; or if
2. trm:setnm is $<trm_1, \ldots , trm_k>:setnm_1 \text{ x } \ldots \text{ x } setnm_k$, where $k \geq 2$, the occurrence is in trm_i, and $trm_i:setnm_i$ assigns set to the occurrence; or if
3. trm is $<trm_1, \ldots , trm_k>$, where $k \geq 1$, dsetnm is the immediate domain predecessor of setnm, and $<trm_1, \ldots , trm_k>:dsetnm$ assigns set to the occurrence.

For example, the sets assigned to the occurrences of 'x' and 'y' in the term '<x,y>' of the elementary assertion

<x,y>:LEADERDATE

are LEADER and VDATE, since LEADERxVDATE is the immediate domain predecessor of LEADERDATE.

The sets assigned to the first and second occurrences of 'x' in the term of the elementary assertion

<<x,x>,v>:LEADERDATE

are assigned by

<x,x>:LEADER

and therefore by

<x,x>:EMPPROJ

and therefore by

<x,x>:EMPLOYEExPROJECT.

The first occurrence is assigned EMPLOYEE, and the second occurrence is assigned PROJECT. The set assigned to 'v' in '<<x,x>,v>' is VDATE. Should 'v' be replaced by 'MALE', then the occurrence of the constant 'MALE' would also have VDATE assigned to it.

Another example of an elementary assertion that assigns two different sets to two occurrences of the same variable is

<x,<x,y>>:SOCCERxLEADER.

The first occurrence of 'x' is assigned SOCCER, while the second is assigned EMPLOYEE.

The following are examples of elementary assertions that fail to assign sets for all occurrences of variables and constants in their terms:

<x,y>:EMPLOYEE and <x,<x,v>>:LEADERDATE.

An elementary assertion for a set schema is said to be **type assigning** if

1. it assigns a member of $\underline{schema}^*$ to each occurrence of a variable or a constant in its term;
2. the type of the set to which an occurrence of a constant is assigned is the primitive value set of which the constant is a member; and
2. each occurrence of a variable in its term is assigned a set of the same type as any other occurrence of the same variable.

Elementary assertions that are not type assigning are never used. Therefore henceforth by 'elementary assertion' for a set schema will be meant 'type-assigning elementary assertion' for the schema.

If each constant and variable occurring in a term $\underline{trm}$ is assigned a type, then $\underline{trm}$ is assigned a type also as follows:

1. If $\underline{trm}$ is a constant or variable, then the type assigned to that constant or variable is the type assigned to $\underline{trm}$; and
2. If $\underline{trm}$ is $<\underline{trm}_1, \dots, \underline{trm}_k>$, where $k \geq 2$, and $\underline{trm}_i$ is assigned $\underline{type}_i$, then $\underline{trm}$ is assigned the type $<\underline{type}_1, \dots, \underline{type}_k>$.

It follows, therefore, that an elementary assertion assigns a type to its term.

9.4. Equality Assertions

An equality assertion is an expression of the form
trm1=trm2,
where trm1 and trm2 are terms.

Examples of equality assertions are
x=<y,'MALE'>, 123=x, <x,<x,v>>=<x,y>
Unlike elementary assertions, an equality assertion does not assign types to variables occurring in its terms. However, when types are assigned to the variables occurring in the terms of an equality assertion, the assertion can only be interpreted if both terms are assigned the same type.

An occurrence of a variable in an assertion assert of DEFINE, as defined in chapter 3, may have its occurrence in an elementary assertion that is a part of assert. Except for the possibility of ambiguities arising from the same name being used for distinct sets, each such occurrence has therefore a unique type assigned to it. Restrictions on the definition of assertions, such as the restriction of elementary assertion to type assigning elementary assertion, removes the possibility of conflicts in types.

CHAPTER 3 THE LANGUAGE DEFINE

This chapter is concerned with the form and meaning of declarations of defined sets. It assumes a familiarity with chapter 2, especially the definitions provided in section 9 of that chapter.

The declaration of a defined set is made in the context of a set schema schema consisting of the previously declared base and defined sets. The format for the declarations of defined sets introduced in section 4.2 of chapter 2 is

dsetnm for {d+v-decl | assert | comment},

where d+v decl is the domain+variable declaration clause, called briefly the **declaration clause** in this chapter, and assert the assertion clause.

The machine interpretable declaration and assertion clauses have two purposes:

1. A domain for dsetnm is declared in d+v-decl that is a member of the extended schema schema*; and
2. The intension of dsetnm, which determines its membership, is declared in d+v-decl and assert.

For example, MALE was declared in section 6.1 of chapter 2 as follows:

MALE for {x:EMPLOYEE | <x,'MALE'>:SEX | male employees}.

The declaration clause declares EMPLOYEE to be the domain of MALE . That clause together with the assertion clause declares that the membership of MALE is to consist of those members x of EMPLOYEE for which <x,'MALE'> is a member of SEX.

As will be seen in section 1 where declaration clauses are discussed, it is a simple matter to restrict the format of a declaration of a defined set in order that the first of the two purposes is served. But it is more difficult to restrict the format so as to ensure that the second of the purposes is served. It is necessary to introduce two restrictions that d+v-decl and assert must satisfy. They are

1. Every variable with a **free** occurrence in assert must be declared in d+v-decl; and
2. The clauses d+v-decl and assert must be **type compatible.**

The meaning of 'free' and 'type compatible' will be defined in stages the first stage in section 1, and later stages in sections 3 through 6 as increasingly complex forms of assert are introduced. The first of the two restrictions is called the **free variable** restriction, and the second the **type compatibility** restriction.

That some restriction on the pairs of declaration and assertion clauses admitted should be considered, should be evident from the following expression:

N for {x:PROJECT | <x,'MALE'>:SEX | nonsense}.

The type assigned to 'x' in the declaration clause is PROJECT, while the type

assigned to it in the assertion clause is EMPLOYEE; the declaration and assertion clauses are not type compatible. Such a declaration could, of course, be taken to be a declaration for an empty subset of PROJECT, but the better course is the prohibitition of it. Not only does the prohibition result in a simple and clear interpretation for declarations of defined sets, but it has practical benefits as well since it results in the partitioning of the universe of all entities by the primitive sets.

A set schema schema is declared so that a database management system may maintain a database with this schema. In section 2 a sketch is given of what a database management system is expected to do with schema, and the interpretation the system is expected to give to a declaration with an elementary or equality assertion as its assertion clause. Sections 3-6 complete the description and at the same time provide a full definition of the assertions of DEFINE. The interpretation of assertion clauses given in sections 2-6 is the one that users of a management system for set schemas must understand. Only then can they devise set declarations that properly express their intent, both for declaring new members of a set schema, and for formulating queries for the database based on the set schema.

The language of DEFINE has its basis in set theories that are described in [Gil86a]. The assertions of DEFINE are constructed from elementary and equality assertions described in section 3, using the boolean truth values **true** and **false** and the boolean operators **and**, **or**, and **not**, described in section 4, and the quantifiers **[For some** ...] and **[For all** ...] described in section 5. In section 6 a functional notation is introduced through the use of parameterized set names.

The remaining topics dealt with in the chapter are nested declarations in section 7, the formulation of user queries in section 8, the consistency of the interpretation of assertion in section 9, and extensions of DEFINE in section 10.

One assumption simplifies the task of describing an interpretation of the assertions of DEFINE. Although declaring distinct sets with the same name can result in ambiguities, the assumption will be made that they do not occur; that is, it is assumed that each name of a declared set occurring in an assertion identifies a unique member of the set of currently declared sets. Set naming rules that minimize the risk of ambigities, as well as possible mechanisms for resolving them, will be discussed in chapter 4.

1. DOMAIN+VARIABLE DECLARATIONS

A declaration clause d+v-decl is an elementary assertion

trm:setnm,

necessarily type assigning as defined in section 9 of chapter 2. The domain declared by the declaration is the set setnm. Since it is a member of schema*, the

requirement that the domain of a set be declared before the set can be declared is always satisfied.

The **immediate domain predecessors** of dsetnm are defined as follows:

Should setnm be $setnm_1$x ... x $setnm_k$, where k≥2, then $setnm_1$, ... , $setnm_k$ are the immediate domain predecessors of dsetnm; otherwise setnm is the only immediate define predecessor.

A declaration clause of the form

$<trm_1, \ldots, trm_k>:setnm_1 \text{x} \ldots \text{x}\ setnm_k$

may also be written in the form

$trm_1:setnm_1, \ldots, trm_k:setnm_k$.

This second form, however, has the same meaning as the first. Therefore it may be assumed that only the first form is used.

As described in section 9 of chapter 2, a clause d+v-decl assigns a type to each variable occurring in its term, and the clause assert may assign a type to each variable occurring in it. For the clauses to be **type compatible**, a type assigned to a variable by assert must be the same as the type assigned to it by d+v-decl.

2. A MANAGEMENT SYSTEM for SET SCHEMAS

As described in chapter 1, a database management system should perform three basic functions for its users. First it assists in the creation of databases by accepting a set schema and preparing data structures designed to record the memberships of sets declared in the schema. Second, it modifies its record of the membership of declared sets by reacting to update commands given to it by users. Finally, it responds to queries posed as declarations of defined sets by listing the membership of the sets.

A first highly simplified description is given in section 2.1 of how a management system records the memberships of sets. Fundamental to this description is the concept of an **internal surrogate** for an entity. A thorough description involves implementation issues that must await chapter ???.

A users' view of the management system is sketched in section 2.2. For the present it is assumed that a user may command the system to add a new member to, or remove an existing member from, a declared base set. The form of such a command, as well as the meaning that can be given to it when it is applied to a defined set, rather than a base set, will be discussed in chapter ???.

The section concludes with a definition and discussion of the **immediate predecessor**

association in section 2.3. It is an extension of the immediate domain predecessor association.

2.1. Internal Surrogates and Tuples of Surrogates

The intension of a primitive base set such as EMPLOYEE is described in its declaration in terms that only a human being can understand. Once that intension has been described, it is possible for a human being to decide that a particular entity is a member of the set and pass to a management system values of the entity's attributes in the expectation that they will be recorded for later retrieval. For example, if it is determined that a given person is a member of the set EMPLOYEE, then values of the attributes EMP#, NAME, ADDRESS, and SEX for the person may be passed.

For the system to record values of attributes, it must have some means of associating them with the same entity, and for this purpose it creates an internal surrogate for the entity. Such a surrogate may, for example, be the value of an identifier of the entity, such as the value of EMP# for a member of EMPLOYEE, but more usually it is simply an integer chosen by the system at the time that it is first needed and that is unknown to any user of the system.

All values of attributes of an entity passed to an information system, including the value of an identifier for the primitive base set of which the entity is a member, are associated by the system with the internal surrogate of the entity. For example, if e is an internal surrogate for the person that is a member of EMPLOYEE, then the values of EMP#, NAME, ADDRESS, and SEX for the person, say respectively e#, nm, add, and sex, are passed to the system. Should these values be confirmed by the system to be members respectively of the value sets VEMP#, VNAME, VADDRESS, and VSEX, then they are associated with e by the system.

From the user's point of view the system is recording the pairs <e,e#>, <e,nm>, <e,add>, and <e,sex>, as members of the nonprimitive base sets EMP#, NAME, ADDRESS, and SEX, although how it actually does so is irrelevent to the user. The values of identifiers for the members of primitive base sets ensure a one-to-one correspondance between the entities of the external world and the internal surrogates representing them in the memory of the management system.

The pairs <e,e#>, <e,nm>, <e,add>, and <e,sex>, are examples of what are called tuples of surrogates, defined as follows:

> A **tuple of surrogates** is an internal surrogate for a member of a primitive base set, a constant, or a tuple of tuples of surrogates.

Other examples of tuples of surrogates are

> e, 'MALE', 123, and <e,'MALE'>,

where e is the internal surrogate for a member of EMPLOYEE.

An internal surrogate that is a member of a primitive base set is aid to have that set as its **type**. A tuple of surrogates $\langle \underline{tpsr}_1, \ldots, \underline{tpsr}_k \rangle$ is said to have the **type** $\underline{tp}_1 \times \ldots \times \underline{tp}_k$, where $\underline{tpsr}_i$ has the type $\underline{tpsr}_i$.

2.2. Users' View of the System

As far as users are concerned, the system can be thought of as maintaining the membership of all primitive base sets through the creation of internal surrogates representing the members added to the sets by the users. The membership of EMPLOYEE, for example, can be thought of as consisting at any time of the internal surrogates that have been recorded as its members by the system.

Further, the membership of the primitive value sets, that is of the primitive defined sets INTEGER, STRING, and REAL, can be thought of as being maintained in a **virtual** sense; the system does not actually maintain a list of all the members of these sets, but is able to decide of any string presented to it whether it is a member of one of them. Similarly the membership of nonprimitive values sets such as the value sets of the attributes of EMPLOYEE can be thought of as being maintained in a virtual sense also; the system can be understood to be capable of interpreting the intension of such a set and of determining whether a given string satisfies the intension.

The members of nonprimitive base sets can be thought of as being tuples of surrogates. For example, the members of SEX are the pairs <e,sex> for which e is an internal surrogate of a member of EMPLOYEE, sex identifies a member of VSEX, and sex has been passed to the system as the value of SEX for the employee with internal surrogate e.

Just as the membership of the value sets can be thought of as being maintained in a virtual sense by the system, the membership of a nonprimitive defined set such as MALE can also be so regarded. For example, the variable 'x' occuring in the declaration of MALE is assigned EMPLOYEE as its range. This means that the possible values for 'x' are the internal surrogates that are currently members of EMPLOYEE, and that 'x' can be **bound to** one of these internal surrogates. Once 'x' has been so bound, say to e, then it is possible to assert that x is a member of EMPLOYEE. Also when 'x' has been so bound, <x,'MALE'> can be regarded as being bound to <e,'MALE'>, and it is possible to assert that <x,'MALE'> is or is not a member of SEX.

At any given time, therefore, the membership of MALE consists of those internal surrogates e for which x is a member of EMPLOYEE and <x,'MALE'> is a member of SEX, when 'x' is bound to e. Since the membership of MALE can be

calculated at any time, the membership can remain virtual and need not be recorded in the **actual** manner that the membership of EMPLOYEE is recorded as a list of internal surrogates.

For the sake of efficiency the system can calculate the membership of any defined set and make the membership actual, but without considerations of efficiency, the membership of all defined sets can remain virtual. For the purposes of this chapter it is irrelevent whether the membership of a defined set is maintained as an actual or a virtual list.

Similarly the system can be thought of as recording the membership of any set in schema*. For example, should (EMPLOYEExPROJECT)xINTEGER be the type of the tuple of surrogates <<e,p>,d>, then the system can be assumed to record those <<e,p>,d> that are members of LEADERDATE.

Terms as well as variables can be bound to tuples of surrogates: Should each variable that occurs in a term trm be bound to a tuple of surrogates, then trm itself is **bound** to a tuple of surrogates defined as follows:

1. Should trm be a constant, then trm is bound to that constant;
2. Should trm be a variable var, then trm is bound to the tuple of surrogates to which var is bound; and
3. Should trm be $\langle trm_1, \ldots, trm_k \rangle$, then trm is bound to the tuple $\langle tup_1, \ldots, tup_k \rangle$, where trm_i is bound to tup_i.

Using this notion of the binding of terms to internal surrogates, the role a declaration clause trm:setnm plays in the specification of the intension of a defined set can now be better understood. The variables occurring in trm are necessarily assigned types by the clause in such a way that the type assigned to trm is the type of setnm. When those variables are assigned tuples of surrogates from the sets that are their types, trm is assigned a tuple of surrogates of the same type as setnm. The extension of the declared set, therefore, is drawn from those tuples of surrogates that are members of setnm.

To list the membership of a set for a user, the system replaces internal surrogates with identifiers. For example, to list the membership of MALE, the system can be thought of as binding 'x' in turn to each member e of EMPLOYEE and then printing for the user the values of EMP# for those e for which <x,'MALE'> is a member of SEX.

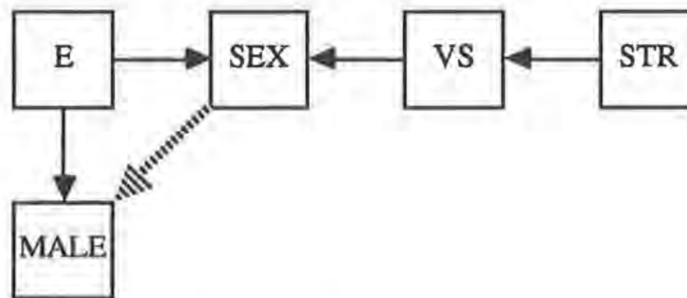
2.3. Immediate Predecessors

The immediate domain predecessors of a defined set are, as described in section 1.1, determined by the declaration clause. The **immediate define predecessors** of the set

are similarly determined by the assertion clause. For example, the set EMPLOYEE is the only immediate domain predecessor of MALE, while SEX is its only immediate define predecessor. As different forms of assertion clauses are considered in sections 3 through 6, the immediate define predecessor association will be defined for each form. Since the declaration of a base set does not have an assertion clause, a base set has no immediate define predecessors.

An **immediate predecessor** of a declared set, either base or defined, is a set that is either an immediate domain predecessor or an immediate define predecessor of it. For example, both EMPLOYEE and SEX are immediate predecessors of MALE. The domain graphs introduced in chapter 2 can be extended to **predecessor** graphs by adding edges <n1,n2> for which the node n1 is labelled with an immediate define predecessor of the set labelling n2. For example, in figure 2.3, a predecessor graph is illustrated with nodes labelled with the sets EMPLOYEE (E), SEX, VSEX (VS), STRING (STR), and MALE.

FIGURE 2.3



A different style of arrow is used in this figure to illustrate the single edge arising from the immediate define association.

Just as the domain predecessor association was defined to be the transitive closure of the immediate domain predecessor association, so is the predecessor association the transitive closure of the immediate predecessor association:
A declared set A is a **predecessor** of a declared set C, if A is an immediate predecessor of C, or if there is a declared set B such that A is a predecessor of B and B is a predecessor of C.

The only domain predecessor of MALE is EMPLOYEE. But the predecessors of MALE are EMPLOYEE, SEX, VSEX, and STRING.

Domain graphs are directed acyclic because each immediate domain predecessor of a set must be declared prior to the declaration of the set. Similarly predecessor graphs are also directed acyclic because each immediate define predecessor of a defined set must also be declared prior to the declaration of the set.

Illustrations of predecessor graphs can be useful for users of a management system for a set schema, while the graphs themselves are useful for the system itself.

3. ELEMENTARY and EQUALITY ASSERTION CLAUSES

In this section declarations are considered of the form
dsetnm for {trm:setnm | assert | comment},
where assert is either an elementary assertion
trma:setnma,
or an equality assertion
trm1=trm2.

Each occurrence of a variable in trma:setnma or trm1=trm2 is a **free** occurrence. By the free variable restriction, therefore, every variable occurring in assert must occur in trm. Therefore each variable occurring in assert has a type assigned to it by trm:setnm. Should assert be trma:setnma, then the variable also has a type assigned to it by trma:setnma. For the declaration and assertion clauses to be **type compatible** it is necessary that the types be the same for each variable occurring in assert. Should assert be trm1=trm2, then for the clauses to be **type compatible** it is necessary that the type assigned to trm1 be the same as the type assigned to trm2.

The declaration and assertion clauses of the declaration of MALE are type compatible. An example, although an artificial one, of a declaration with an equality assertion as its assertion clause is:
EL for {x:EMPPROJ, <y,z>:LEADER | x=<y,z> | an example declaration}.
Since the type assigned to 'x' is EMPLOYEExPROJECT, the type assigned to 'y' is EMPLOYEE, and the type assigned to 'z' is PROJECT, the type assigned to both 'x' and '<y,z>' is EMPLOYEExPROJECT. The members of EL consist of those pairs <<e,p>,<e,p>> for which <e,p> is a member of LEADER.

The **immediate define predecessors** of dsetnm are defined as follows:

1. Let assert be trma:setnma. Should setnma be $setnm_1$ x ... x $setnm_k$, where k≥2, then $setnm_1$, ... , $setnm_k$ are the immediate define predecessors of dsetnm; otherwise setnma is the only immediate define predecessor.
2. Let assert be trm1=trm2. Then dsetnm has no immediate define predecessors.

3.1. Assigning Truth Values to Elementary and Equality Assertions

The declaration and assertion clauses trm:setnm and assert of the declaration of dsetnm are type compatible, and every variable occurring in assert occurrs in trm. Let the variables occurring in trm be bound to tuples of surrogates from their types

in such a way that trm is bound to a tuple of surrogates that is a member of setnm.

Consider first the case that assert is trma:setnma. Each variable occurring in trma is necessarily bound to tuples of surrogates for which trma is bound to a tuple of surrogates tpsr that is has the type of setnma. Then tpsr may or may not be a member of setnma. If it is, trma:setnma is assigned the truth value **true**, and if it is not trma:setnma is assigned the truth value **false**.

Consider now the case that assert is trm1=trm2. The terms trm1 and trm2 are assigned the same type and necessarily each is assigned a tuple of surrogates that is of the same type. The tuples of surrogates may or may not be identical. If they are, trm1=trm2 is assigned the truth value **true**, and if they are not trm1=trm2 is assigned the truth value **false**.

4. THE BOOLEAN TRUTH VALUES and OPERATORS

In this section declarations are considered of the form
dsetnm for {trm:setnm | assert | comment},
where assertion is one of three forms, truth values discussed in section 4.1, conjunctions or disjunctions discussed in section 4.2, and negations discussed in section 4.3. In section 4.4 truth functional identities for assertion clauses are listed.

4.1. Truth Values

The assertion clause assert may be just one of the truth values **true** or **false**.

No variable occurs **free** in a truth value. Necessarily therefore the free variable and type compatibility restrictions are satisfied for any declaration clause trm:setnm when the assertion clause assert consists of a truth value. In that case the set dsetnm has no immediate define predecessors.

For example, the declaration
WORKER for {x:EMPLOYEE | **true** | an alias for EMPLOYEE}
essentially provides a different name for the set EMPLOYEE, or what is called an **alias** for EMPLOYEE. The sets WORKER and EMPLOYEE are always extensionally identical, no matter the membership of EMPLOYEE, because the assertion clause is always assigned the truth value **true**.

The declaration
EMPTY for {x:EMPLOYEE | **false** | an empty set}
on the other hand never has any members, for its assertion clause can never be assigned the truth value **true**.

4.2. Conjunctions and Disjunctions

The assertion clause of a declaration may be the **conjunction**
(assert1 **and** assert2),
of the assertions assert1 and assert2, or may be the **disjunction**
(assert1 **or** assert2)
of them, provided that each of assert1 and assert2 separately satisfies the free variable and the type compatibility restrictions with the declaration clause trm:setnm. The parenthesis '(' and ')' may be omitted for repeated conjunctions and disjunctions and where they are not required to determine the arguments of the operators **or** and **and**.

The assertions assertions assert1 and assert2 may be elementary or equality assertions, truth values **true** and **false**, or assertions making use of any number and mixture of conjunctions and disjunctions, or assertions making use of negations and quantifiers introduced below.

A free occurrence of a variable in assert1 or assert2 is a **free** occurrence in the conjunction and the disjunction. So that if assert1 and assert2 separately satisfy the free variable restriction, so must the conjunction and disjunction as well. Similarly, since an occurrence of a variable in the conjunction or disjunction must be an occurrence in either assert1 or assert2, and since assert1 and assert2 must separately satisfy the type compatibility restriction, the conjunction and disjunction satisfy the restriction as well. The **immediate define predecessors** of dsetnm when assert is either (assert1 **and** assert2) or(assert1 **or** assert2) are those it would have if assert were assert1 as well as those it would have if assert were assert2.

When each variable with a free occurrence in assert1 and assert2 has been bound to an internal surrogate that is a member of its type, then each of assert1 and assert2 is assigned a truth value. The truth value that is then assigned to their conjunction and disjunction is determined from truth tables given in figure 4.2. These truth tables are the conventional ones for conjunction and disjunction.

FIGURE 4.2

assert1 \ assert2	**and**	**true**	**false**
	true	**true**	**false**
	false	**false**	**false**

assert1 \ assert2	**or**	**true**	**false**
	true	**true**	**true**
	false	**true**	**false**

An example of a declaration using **or** is the following:

B+F for {x:EMPLOYEE | x:BASEBALL **or** x:FEMALE}.

The set B+F has as its members the employees who are members of BASEBALL, or who are members of FEMALE. An employee who is a member of both sets is a member of B+F. The **or** connective is always used in this nonexclusive sense of allowing one or the other or both cases to be true.

Another example using **or** is the following,

RU for {x:PROJECT | <x,'RECRUITING'>:NAME **or**
<x,'UNITEDWAY'>:NAME}.

If 'RECRUITING' and 'UNITEDWAY' are members of the value set for the attribute NAME of PROJECT, then the members of RU are employees assigned to one or the other or both of the projects.

A declaration using **and** is:

SB for {x:EMPLOYEE | x:SOCCER **and** x:BASEBALL}.

The members of the set SB are members of both SOCCER and BASEBALL.

The next example of the use of **and** declares a set of triples.

DEP for {x:DEPARTMENT, y:EMPLOYEE, z:PROJECT |
<x,y>:EMPDEPT **and** <x,z>:EMPPROJ}

The members of DEP consist of the triples <d,e,p> of surrogates for which e has been assigned to both d and p.

A declarations using both **and** and **or** is :

A for {x:EMPLOYEE, y:PROJECT | <x,y>:EMPPROJ **and**
(<y,'RECRUITING'>:NAME **or** <y,'UNITEDWAY>:NAME)}

The members of this set consist of the pairs <e,p> of surrogates for which the employee with surrogate e is assigned to a project with surrogate p that has the name 'RECRUITING' or the name 'UNITEDWAY'.

The parentheses used in the assertion clause indicate the order in which the boolean connectives are to be applied. The **or** connective is applied first, and the **and** operator applied second. An equivalent declaration using the previous example RU is:

A for {x:EMPLOYEE, y:PROJECT | <x,y>:EMPPROJ **and** y:RU}.

The choice of the name 'A' for the set illustrates that it is not always necessary to invent descriptive names for sets. Definitions of sets, it will be seen, are often only needed for contexts in which brevity of name is desirable and in which the intension of the set is immediately at hand to clarify the meaning of the name. These contexts include nested declarations of defined sets discussed in section 5.

Exercise: Verify for each of the declarations that the assertion clause is acceptable for the domain+variable declaration clause.

4.3. Negation

The assertion clause of a declaration may be the **negation**

not assert,

of the assertion assert provided assert satisfies the free variable and type compatibility restrictions with the declaration trm:setnm.

The assertion assert may be an elementary or equality assertion, a truth value **true** or **false**, or an assertion making use of any number and mixture of conjunctions and disjunctions, or an assertion making use of quantifiers introduced below.

A free occurrence of a variable in assert is a **free** occurrence in **not** assert. Therefore if assert satisfies the free variable restriction with trm:setnm, so does trm:setnm. Similarly, if assert satisfies the type compatibility restriction with trm:setnm, so does trm:setnm. The **immediate define predecessors** of dsetnm when its assertion clause is **not** assert are those it would have if its assertion clause were assert.

Under the assumption that a truth value has been assigned to assert, the negation is assigned the truth value given in the truth table of figure 4.3. This table is the usual truth table for negation.

FIGURE 4.3

	not	
assert	**true**	**false**
	false	**true**

A declaration using **not** is:

NONSOCCER for {x:MALE | **not** x:SOCCER | male employees not playing soccer}.

The members of NONSOCCER is the set of internal surrogates e of employees that are male and do not play soccer.

Note that a declaration such as

NONSENSE for {x:DEPARTMENT| **not** x:EMPLOYEE | nothing defined}

is not properly formulated because the assertion clause '**not** x:EMPLOYEE' is not type compatible with the domain+variable declaration clause 'x:DEPARTMENT'. The type assigned to 'x' by the first is EMPLOYEE, while the type assigned to 'x' by the second is DEPARTMENT.

4.4. Assertions with Identical Truth Values

From the truth tables for conjunction, disjunction, and negation, it can be seen that certain pairs of assertions always have the same truth values. The pairs listed in figure 4.4 are some of the more obvious ones.

FIGURE 4.4

assert **or true**	**true**
assert **or false**	assert

assert **and true**	assert
assert **and false**	**false**
not (assert1 **or** assert2)	**not** assert1 **and not** assert2
not (assert1 **and** assert2)	**not** assert1 **or not** assert2

5. THE QUANTIFIERS

This section is concerned with the form and meaning of declarations
dsetnm for {trm:setnm | [**quan** qvardec](assert) | comment}
where the assertion clause
[**quan** qvardec](assert),
is called a **quantified assertion** provided it satisfies a type compatibility restriction described in section 5.3.

The assertion assert is called the **scope** of the **quantifier prefix** [**quan** qvardec]. The parentheses '(' and ')' may be omitted when assert is itself a quantified assertion, or when it is an elementary or equality assertion. The expression **quan** in the quantifier prefix, called the **quantifier**, is either '**For some**' or '**For all**', called respectively the **existential** and the **universal** quantifier.

The expression qvardec, called the **quantifier variable declaration**, has for the present the same form as a domain+variable declaration, namely an elementary assertion. As before the variable declaration
$< trm_1, \ldots, trm_k >: setnm_1 \text{ x } \ldots \text{ x } setnm_k$
may also be written
$trm_1 : setnm_1, \ldots, trm_k : setnm_k$.
In section 6 a more general form of quantifier variable declaration is admitted.

As noted above, the definition of the type compatibility restriction that [**quan** qvardec](assert) must satisfy to be a quantified assertion is postponed until section 5.3. Also postponed are the definitions of free occurrence and of the immediate domain predecessors of dsetnm, as well as a discussion of the free variable and type compatibility restrictions that the declaration and assertion clauses must satisfy. This permits an informal introduction to quantified assertions to be given first.

Many examples of set declarations with quantified assertion clauses appear in sections 5.1 and 5.2, with examples of the existential quantifier given in section 5.1, and the universal in section 5.2. In section 5.4 a detailed treatment of the assignment of truth values to quantified assertions is provided.

5.1. Informal Introduction to Existential Quantifiers

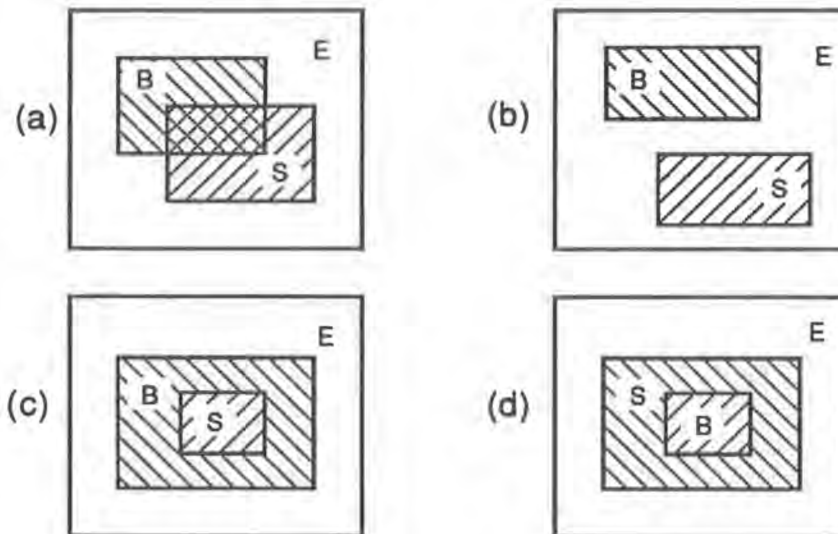
The meaning of the existential quantifier can be illustrated with a simple example. Consider the quantified assertion
(1) [**For some** x:BASEBALL] x:SOCCER.
In this assertion '[**For some** x:BASEBALL]' is the quantifier prefix, and

'x:SOCCER' is the scope of the quantifier prefix. There is only one occurrence of the variable'x' in the scope and it is said to be **bound** by the quantifier prefix.

Informally, the assertion (1) is true if there is a baseball player who plays soccer, and false if there is not one. In particular, the assertion is false if no employee is playing baseball. More precisely, EMPLOYEE is the type of both BASEBALL and SOCCER. If there is an internal surrogate e in EMPLOYEE that is a member of both BASEBALL and SOCCER, then (1) is true. Stating the same thing in another way, each of 'x:BASEBALL' and 'x:SOCCER' is assigned a truth value whenever 'x' is bound to an internal surrogate e that is a member of EMPLOYEE. If there is an e for which both are assigned the truth value **true**, then (1) is assigned the truth value **true** also; if there is no such e, then (1) is assigned the truth value **false**.

The possible relationships between the extensions of SOCCER and BASEBALL within the extension of EMPLOYEE is illustrated in figure 5.2. In that figure the names of the sets have been abbreviated to the first letter of the names. In all of the situtations illustrated, with the exception of (b) where it is false, the assertion (1) is true.

FIGURE 5.1



There is a simple connection between the existential quantifier and the boolean connective **or**. Let $\underline{b}_1, \ldots, \underline{b}_k$ be a list of the internal surrogates that are members of BASEBALL. Let $x_1, \ldots, x_k$ be distinct variables of type EMPLOYEE that are bound respectively to $\underline{b}_1, \ldots, \underline{b}_k$. Then the assertion (1) has the same truth value as the assertion

(2) x_1:SOCCER **or** ... **or** x_k:SOCCER.

If one of the employees corresponding to the internal surrogates does play soccer, then assertion (2) is true, and if none do, then it is false.

The assertion (2) has the same truth value as (1) only when $\underline{b}_1, \ldots, \underline{b}_k$ are the internal surrogates of all employees who play baseball; should the list of baseball playing employees change, it will be necessary to change (2) in order to have the same truth value as (1). This is the primary reason that quantified assertions are needed and are not be replaced with an assertion like (2). The assertion (1) does, however, have the same meaning as

(3) **[For some** x:SOCCER] x:BASEBALL.

The following is an example of a declaration of a defined set with a quantified assertion clause.

EMPPROJ.EMPLOYEE for {x:EMPLOYEE |
  [For some y:PROJECT] <x,y>:EMPPROJ |
    the projection of EMPPROJ on EMPLOYEE}.

This declaration declares the projection operator that was used in section 3.5 of chapter 2. The immediate predecessors of EMPPROJ.EMPLOYEE are EMPLOYEE, PROJECT, and EMPPROJ.

The members of EMPPROJ.EMPLOYEE are those employees assigned to projects. The existential quantifier has the effect of converting the pairs <$\underline{e}$, $\underline{p}$> that are members of EMPPROJ into a subset of EMPLOYEE by "projecting" on the first element of the pair.

The set of employees of just the projects RECRUITING and UNITEDWAY can be similarly declared by applying an existential quantifier to the association A defined in section 4.2.

A.EMPLOYEE for {x:EMPLOYEE | **[For some** y:PROJECT] <x,y>:A}

Another example of the use of existential quantifiers is provided by the next declaration.

EMPLOYEE for { u:VEMP#, v:VNAME, w:VADDRESS, z:VSEX |
  [For some x:EMPLOYEE] (<x,u>:EMP# **and** <x,v>:NAME
    and <x,w>:ADDRESS **and** <x,z>:SEX)| the employee table }

The association being defined can be given the name 'EMPLOYEE', even though that name has already been used, because the two sets named EMPLOYEE can be distinguished by their domains. The arity 4 set EMPLOYEE is called a **table** in the relational model.

In the assertion clause of the declaration there are four occurrences of 'x' that are bound by the quantifier prefix **[For some** x:EMPLOYEE]. The immediate

predecesors of the declared set EMPLOYEE of arity 4 are the sets VEMP#, VNAME, VADDRESS, VSEX, the arity 1 set EMPLOYEE, EMP#, NAME, ADDRESS, and SEX.

The use made of the existential quantifier in the assertion clause of the table EMPLOYEE can be understood in terms of a projection operator similar to the one used in EMPPROJ.EMPLOYEE. Consider the set JTE defined as follows:

JTE for
{u:VEMP#, v:VNAME, w:VADDRESS, z:VSEX, x:EMPLOYEE |
<x,u>:EMP# **and** <x,v>:NAME **and** <x,w>:ADDRESS **and** <x,z>:SEX |
the join of the attributes of EMPLOYEE}.

In the terminology of the relational algebra, JTE is the join of the relations EMP#, NAME, ADDRESS, and SEX, on the attribute EMPLOYEE. Using this declaration of JTE, a set E can be declared that has the same extension as the table EMPLOYEE:

E for {u:VEMP#, v:VNAME, w:VADDRESS, z:VSEX |
[**For some** x:EMPLOYEE] < u,v,w,z,x>:JTE | }.

This use of the existential quantifier will be frequently applied in chapter 5 where a method for designing tables from a set schema is described.

5.2. Informal Introduction to Universal Quantifiers

The meaning of the universal quantifier can be illustrated with a simple example. Consider the assertion

(4) [**For all** x:BASEBALL] x:SOCCER.

The assertion(4) is true if each baseball player also plays soccer; in particular the assertion is true if no employee is playing baseball. This assertion is false if there is a baseball player who does not play soccer. More precisely, EMPLOYEE is the type of both BASEBALL and SOCCER. If each member e of EMPLOYEE that is a member of BASEBALL is also a member of SOCCER, then (4) is true. Stating the same thing in another way, each of 'x:BASEBALL' and 'x:SOCCER' is assigned a truth value whenever 'x' is bound to an internal surrogate e that is a member of EMPLOYEE. If for each e for which 'x:BASEBALL' is assigned **true**, 'x:SOCCER' is also assigned **true**, then (4) is assigned **true** ; if there is an e for which'x:BASEBALL' is assigned **true** and 'x:SOCCER' is assigned **false**, then (4) is assigned **false**.

The possible relationships between the extensions of SOCCER and BASEBALL within the set EMPLOYEE have been illustrated in figure 5.1. In all of the situations illustrated, with the exception of (d), the assertion (4) is false; in that situation the assertion is true. The assertion is still true if the rectangle B in (d) is empty, that is if there are no employees playing baseball.

There is a simple connection between the universal quantifier and the boolean connective **and**. As before let $\underline{b}_1, \ldots, \underline{b}_k$ be a list of the internal surrogates that are members of BASEBALL, and let $x_1, \ldots, x_k$ be distinct variables of type EMPLOYEE that are bound respectively to $\underline{b}_1, \ldots, \underline{b}_k$. Then the assertion (4) has the same truth value as the assertion

(5) x_1:SOCCER **and** ... **and** x_k:SOCCER.

If every one of the employees with internal surrogates $\underline{b}_1, \ldots, \underline{b}_k$ plays soccer, then assertion (5) is true, and if one does not, then it is false.

The assertion (4) does not have the same meaning as

(6) [**For all** x:SOCCER] x:BASEBALL.

The assertion (6) is true for the situation illustrated in (d) of figure 5.1, and false for the other three.

In figure 4.4 pairs of assertions with the same truth values were listed. The corresponding figure for quantified assertions is figure 5.2.

FIGURE 5.2

not [**For some** ...](<u>assert</u>)	[**For all** ...] **not** (<u>assert</u>)
not [**For all** ...](<u>assert</u>)	[**For some** ...] **not** (<u>assert</u>)

The following is an example of the use of a declaration with a universally quantified assertion clause.

LEADSALL for {x:EMPLOYEE | [**For all** y:PROJECT] <x,y>:LEADER | }

An employee is a member of LEADSALL if the employee is leader of every project.

The employees that have been assigned to every project are in the following set.

ALLPROJ for {x:EMPLOYEE | [**For all** y:PROJECT] <x,y>:EMPPROJ | }

The following declaration of the set of departments with an employee in common with every project requires the use of both a universal and an existential quantifier:

D for{y:DEPARTMENT | [**For all** z:PROJECT] [**For some** x:EMPLOYEE] <x,y>:EMPDEPT **and** <x,z>:EMPPROJ |}.

It should be clear from the meaning of the quantifiers that they must be used with care if an assignment of a truth value to an assertion in which they appear is to be realistically determined. For example, the use of quantifier prefixes such as [**For some** x:INTEGER] or [**For all** x:INTEGER] with some scopes can result in an

assertion with a truth value that cannot practically be determined.

5.3. Quantifier Variable Declarations

The declaration for dsetnm is assumed to be
dsetnm for {trm:setnm | [**quan** qtrm:qsetnm](assert) | comment},
where [**quan** qtrm:qsetnm](assert) satisfies a type compatibility restriction about to be described, that ensures that trm:setnm and [**quan** qtrm:qsetnm](assert) satisfy the required free variable and type compatibility restrictions.

The **immediate define predecessors** of dsetnm are those of assert and qtrm:qsetnm.

Let lvr be a list of all the variables that occur in both trm and qtrm, and let lvrn be a list of distinct variables of the same length as lvr, each distinct from any variable occurring in either trm or qtrm . Let [lvrn/lvr]trm be the term obtained from trm by replacing each occurrence in trm of a variable from the list lvr with the corresponding variable from the list lvrn.

Consider now the elementary assertion
< [lvrn/lvr]trm, qtrm >: setnm x qsetnm.
It is type assigning because there is no variable with a free occurrence in both [lvrn/lvr]trm and qtrm, and each of the elementary assertions [lvrn/lvr]trm:setnm and qtrm:qsetnm are type assigning. It may, therefore, be the declaration clause of a defined set declaration, either as written or in the alternative form
(a) [lvrn/lvr]trm:setnm, qtrm:qsetnm.

The type compatibility restriction that [**quan** qtrm:qsetnm](assert) must satisfy is the following:
The assertion assert must be type compatible with the declaration (a).
This restricts the quantifier prefixes [**quan** qtrm:qsetnm] that may have scope assert. For example,
[**For some** x:PROJECT] <x,y>:EMPPROJ
is not a quantified assertion because x:PROJECT] and <x,y>:EMPPROJ are not type compatible no matter what type is assigned to 'y'.

Let var be a variable occurring in assert. An occurrence of var in assert is a **free** occurrence in [**quan** qtrm:qsetnm](assert) if var does not occur in qtrm; otherwise it is an occurrence **bound by** the quantifier prefix [**quan** qtrm:qsetnm]. For example
[**For some** y:PROJECT] <x,y>:EMPPROJ
is a quantified assertion in which the single occurrence of 'x' is a free occurrence, and in which the second occurrence of 'y' is bound by the quantifier prefix [**For some** y:PROJECT]. It is this sense of 'free' that determines whether the

declaration and assertion clauses satisfy the free variable restriction requiring that each variable with a free occurrence in [quan qtrm:qsetnm](assert) occurs in trm.

Each variable with a free occurrence in [quan qtrm:qsetnm](assert) may have a type assigned to it because it is an occurrence in assert. That trm:setnm and [quan qtrm:qsetnm](assert) are type compatible, however, follows from the fact that (a) and assert are type compatible.

5.4. Assigning Truth Values to Quantified Assertions

Consider now the assertion clause
(b) [quan qtrm:qsetnm](assert).
of the declaration of dsetnm.

Let fvr be a list of all the variables with a free occurrence in (b), and let ftpsr be a list of tuples of surrogates, each of the same type as that assigned to the corresponding variable of fvr by the declaration clause trm:setnm. To say that fvr is bound to ftpsr is to say that each variable in the list fvr is bound to the corresponding tuple of surrogates in the list ftpsr.

Let bvr be a list of all the variables with a free occurrence in assert that are bound by the quantifier prefix [quan qtrm:qsetnm] in (b). Necessarily each variable with a free occurrence in assert is in one and only one of the lists fvr and bvr. Also each variable in bvr occurs in qtrm, although variables may occur in qtrm that do not occur in the list.

Let $tpsr_1, \dots, tpsr_k$, where $k \geq 0$, be all the tuples of surrogates of the type of qsetnm which are members of qsetnm. Each $tpsr_i$ is the tuple of surrogates to which qtrm is bound when each variable occurring in qtrm is bound to a tuple of surrogates of the type assigned to the variable by qtrm:qsetnm. In particular therefore, each $tpsr_i$ determines a list $btpsr_i$ of tuples of surrogates, each of the same type as as that assigned to the corresponding variable of bvr. To say that bvr is bound to $btpsr_i$ is to say that each variable in the list bvr is bound to the corresponding tuple of surrogates in the list $btpsr_i$.

The truth value assigned to (b) when fvr is bound to ftpsr is determined as follows:

1. Let quan be **For some**.
   The assertion (b) is assigned **true** if for at least one of the lists $btpsr_i$, the assertion assert is **true** when bvr is bound to $btpsr_i$; otherwise it is assigned **false**. In particular it is assigned **false** if k=0.

2. Let quan be **For all.**
   The assertion (b) is assigned **true** if for all of the lists $btpsr_i$, the assertion assert is **true** when bvr is bound to $btpsr_i$; otherwise it is assigned **false.** In particular it is assigned **true** if k=0.

Consider, for example, the quantified assertion

[**For some** <x,y>:EMPPROJ] <x,z>:EMPDEPT.

The term qtrm is <x,y>, setnm is EMPPROJ, and assert is <x,z>:EMPDEPT. The list fvr has 'z' as its only member, and the list bvr has 'x' as its only member. The variable 'y' occurring in qtrm does not occur in assert.

Let d be a current member of DEPARTMENT. The list ftpsr of tuples of surrogates includes only d so that the truth value of the assertion will be determined when 'z' is bound to d.

Let $<e_1,p_1>, \ldots , <e_k,p_k>$ be all the current members of EMPPROJ. Then each $btpsr_i$ has e_i as the only tuple of surrogates in it.

The truth value assigned to the assertion when 'z' is bound to d, is determined as follows: The value is **true** if for some e_i the assertion <x,z>:EMPDEPT is **true** when 'x' is bound to e_i. Otherwise it is **false.**

6. NESTED DECLARATIONS

A declaration can sometimes be made more comprehensible by including within it a **nested declaration** that declares a set that is not needed outside the declaration. For example, short aliases may be declared in a nested declaration to permit the shortening of a longer name. Also the nesting of declarations permit a complicated assertion of DEFINE to be understood in stages through an understanding of the nested declarations. For example, the nested declaration

PFU for {y:PROJECT | [**For some** <x,y>:FU] <x,y>:EMPPROJ |}
where FU for {x:EMPLOYEE | <x,'FINANCE'>:NAME **or**
<x,'UNITEDFUND'>:NAME |} **end**

has included within it a declaration of FU enclosed between **where** and **end**. The declaration of FU is only needed for the purpose of declaring PFU, and is intended to make the meaning of its declaration clearer. It is not added as a member to the set schema consisting of declared sets, only the set PFU is added. Therefore there is greater freedom in the choice of name for FU; it could be any name that does not conflict with names used in the declaration of PFU.

The declarations nested within a given declaration appear as a sequence of declarations following it. A nested declaration may itself have nested declarations within it. The following syntax is used for representing a declaration $\underline{declr}_0$ in which is nested the declarations $\underline{declr}_1$, ... , $\underline{declr}_k$:

$\underline{declr}_0$ **where** $\underline{declr}_1$, ... , $\underline{declr}_k$ **end**

Assuming this declaration is made in the context of a set schema $\underline{schema}$, it has the effect of making the declarations in the reverse order. That is, the set declared in the last declaration $\underline{declr}_k$, may only have members of $\underline{schema}$ as its immediate predecessors, while the set declared in $\underline{declr}_{k-i}$ may have the sets declared in $\underline{declr}_k$, ..., $\underline{declr}_{k-i+1}$, as well as the members of $\underline{schema}$, as its immediate predecessors. Thus the set declared in $\underline{declr}_0$ may have any of the sets declared in $\underline{declr}_k$, ... , $\underline{declr}_1$, as well as the members of $\underline{schema}$, as its immediate predecessors. Declarations can be nested to any depth; that is, each of the declarations $\underline{declr}_1$, ... , $\underline{declr}_k$ may itself be a nested declaration.

Further examples of nested declarations will appear in the next section.

7. PARAMETERIZED SET and TUPLE NAMES

The declaration of a set of arity greater than 1 can be thought of as implicitly declaring many other sets. For example, assuming that 'y' is bound to an internal surrogate of a member of PROJECT, the expression

{x:EMPLOYEE | <x,y>:EMPPROJ}

can be thought of as declaring the set of employees assigned to the project to which 'y' is bound. Similarly, assuming 'x' is bound to the internal surrogate of a member of EMPLOYEE, the set of projects to which the employee is assigned can be thought of as being declared by the expression

{y:PROJECT | <x,y>:EMPPROJ}.

Proper declarations of defined sets cannot be formed from either of these expressions since a variable occurring in what would be the assertion clause is not declared in the domain+variable declaration clause: In the first 'y' is undeclared, while in the second 'x' is undeclared. In each of these the undeclared variable acts as a parameter. Whenever the variable with a free occurrence is declared bound to an internal surrogate, the declaration can be though of as the declaration of a defined set.

A **parameterized set name** is the means by which declarations of this kind are admitted without being explicitly made. The first is expressed as

{<:,y>:EMPPROJ}

and the second as

{<x,:>:EMPPROJ}.

The appearance of ':' in place of 'x' and 'y' respectively indicates that the parameterized set name represents a set abstracted using a variable of the domain+variable declaration in that location. A parameterized set name introduces a functional notation into DEFINE; the tuples of surrogates to which the parameters may be bound are the 'inputs' for the function, and the locations of ':' specify the elements of the tuple of surrogates that is its 'output'.

Parameterized set names may be created from set names of any number of arguments. For example, consider again the arity 4 set EMPLOYEE declared in section 5.1:

EMPLOYEE for { u:VEMP#, v:VNAME, w:VADDRESS, z:VSEX |
    [**For some** x:EMPLOYEE] (<x,u>:EMP# **and** <x,v>:NAME
        and <x,w>:ADDRESS **and** <x,z>:SEX)| the employee table }.

The parameterized set name

{<u,:,:,:>:EMPLOYEE}

has the same meaning as would the expression

{v:VNAME, w:VADDRESS, z:VSEX | <u,v,w,z>:EMPLOYEE}

were it to declare a set when 'u' has been assigned to a member e# of VEMP#; that is, it is the set of triples <nm,add,sex> of values of the attributes NAME, ADDRESS, and SEX, for which <e#,nm> is a member of NAME, <e#,add> is a member of ADDRESS, and <e#,sex> is a member of SEX. An 'input' to {<u,:,:,:>:EMPLOYEE} is any member e# of VEMP#, and its corresponding 'output' is <nm,add,sex>.

The seperation of the 'input' from the 'output' can be made more explicit by declaring a binary version PFE# of the arity 4 set EMPLOYEE as follows:

PFE# for {u:VE#, <v,w,z>:VNAMExVADDRESSxVSEX|
<u,v,w,z>:EMPLOYEE | a partial function of employee#}.

It is called a partial function of EMPLOYEE# since its degrees on VE# can be calculated to be <0,1>. It is a function that is only defined for those members of VE# that have been assigned as employee numbers by EMP# to members of EMPLOYEE; that is to members of the projection of EMP# on VEMP#. But for each such employee number there is a unique triple that is a member of VNAMExVADDRESSxVSEX.

Because functions arise naturally from binary sets, a simplified notation for parameterized set names obtained from binary sets is introduced. The alternative infix form for an elementary association <trm1,trm2>:setnm that was briefly mentioned in section 9 of chapter 2, is employed. The alternative form

trm1:setnm:trm2

is suitable only for binary sets setnm. For example, 'x:EMPPROJ:y' expresses that the internal surrogate to which 'x' is bound is EMPPROJ-associated with the internal surrogate to which 'y' is bound.This alternative form provides a simpler notation for parameterized set names. The first two such names introduced above can be written respectively

{:EMPPROJ:y} and {x:EMPPROJ:}.

By using the defined set PFE#, the simpler parameterized set name {u:PFE#:} can be used in place of {<u,:,:,:>:EMPLOYEE}.

In 7.1 and 7.2 the uses of parameterized set names in quantified assertions and in representing system defined functions are informally described. A variant of parameterized set names, called a **parameterized tuple name**, that is needed for declaring certain kinds of queries is described in section 7.3. Since parameterized set names are to provide a functional notation, the nesting of them as functions is permitted. This use of the notation is explained in section 7.4. Finally in section 7.5, a formal treatment of parameterized set and tuple names is provided.

7.1. Parameterized Set Names in Quantifier Prefixes

An example of a declaration using a parameterized set name is the following:

ALLMALE for {y:PROJECT | [**For all** x:{EMPPROJ:y}] x:MALE |
projects staffed only with males }

In this declaration, the variable 'y' used as a parameter in the parameterized set name {EMPPROJ:y} is declared in the declaration clause of the set declaration, rather than being declared in the quantifier prefix in which it appears. The members of ALLMALE are those projects with only male employees assigned to

them. Its membership may be any subset of PROJECT, depending upon how male employees are assigned to projects. In constrast the set

A for {y:PROJECT | [**For all** <x,y>:EMPPROJ] x:MALE |}

can have only two subsets of PROJECT as its membership; it will have every project as a member if only male employees are assigned to all projects, or will have no project as a member otherwise.

7.2. System Declared Functions

The primitive value sets INTEGER, STRING, and REAL have system defined associations declared with them. For example,

= for {x:INTEGER, y:INTEGER | system defined | identity of integers},

and

≥ for {x:INTEGER, y:INTEGER | system defined | ordering on integers}

are associations that one expects to find available. Another is

L for {x:STRING, y:INTEGER | sytem defined | the length of a string}.

L is functional, that is, an integer is determined for any string in STRING that is the length of the length of the string. Therefore the parameterized set name {x:L:} can be usefully employed in declarations of defined sets. For example, the following declaration appeared in section 4 of chapter 2:

VNAME for {x:STRING | {x:L:} ≤ 20 | a value set for names}.

This use of {x:L:} is typical of a functional notation.

A number of other associations are need for many queries. For example, the associations COUNT, SUM, AVG, MAX, and MIN are called built-in functions in the language SQL for the relational model. These are different from any associations considered so far in as much as they are associations with first argument a set. For example, COUNT would be declared

COUNT for {x:DSET, y:INTEGER | system defined| # of members in the set}.

Here DSET is the set of all currently declared sets, and COUNT associates with a member of DSET the number of its members.

With COUNT available the following set can be declared :

PROJSIZE for {x:PROJECT, y:INTEGER | {:EMPPROJ:x}:COUNT:y | }.

When 'x' is bound to a member of PROJECT, {:EMPPROJ:x} is the set of employees that are assigned to the project. Therefore, PROJSIZE associates with a project the number of employees assigned to the project.

The association PROJCOUNT associates with a department, the number of projects to which employees assigned to the department have been assigned. It is declared

PROJCOUNT for {x:DEPARTMENT, y:INTEGER |{x:DP:}:COUNT:y |}

where DP for {y:DEPARTMENT, z:PROJECT|

[**For some** x:EMPLOYEE](x:EMPDEPT:y **and** x:EMPPROJ:z |} **end**

7.3. Ordered Sets and Parameterized Tuple Names

The functions SUM and AVG must often take an ordered set, rather than just a set as their argument. Assume for example that a SALARY attribute has been declared for EMPLOYEE . To sum the salaries of all employees, SUM does not take the set
SE for {z:REAL | [**For some** x:EMPLOYEE] x:SALARY:Z | }
of salaries of employees as its 'input'. Because of the likelyhood of two employees having the same salary the sole member of
TOTALSALARY for {z:REAL | SE:SALARY:z |}
is not the desired sum. The declaration of SE as a set must be replaced by a declaration of it as an ordered set.

Recall from section 1.3 of chapter 2 that an ordered set is a set with an order specified for its members, and because it is ordered, it may have more elements than it has members. Ordered sets have been employed in the form of tuples in which the elements of the set are explicitly listed.

Rather than declaring SE as a set, it must be declared as an ordered set as follows:
SE for < z:REAL | [**For some** x:EMPLOYEE] x:SALARY:Z | >.
This declaration differs from the previous only in having '<' and '>' replace '{' and '}'. It declares SE to be a tuple of real numbers. There will be the same number of elements in SE as there are employees since SALARY is functional. With the new declaration of SE, the sole member of TOTALSALARY is the desired sum.

A more complicated example is the following. Assume that an association PROJCOST is to be declared that associates the sum of the salaries of employees assigned to a project with the project. The following declaration of PROJCOST not only employs the declaration of an ordered set, but also a **parameterized tuple name.**
PROJCOST for {x:PROJECT, z:REAL | <x:PS:>:SUM:z | } **where**
PS for <x:PROJECT, z:REAL|
[**For some** y:EMPLOYEE] (x:EMPPROJ:y **and** x:SALARY:z)|> **end**
PS must be declared as an ordered set to avoid losing pairs corresponding to employees with duplicate salaries. Similarly, the parameterized tuple name <x:PS:> must be used instead of the parameterized set name {x:PS:}.

A simpler declaration of PROJCOST is provided in the next section.

7.4. Nested Parameterized Names

The declaration of PS nested in the declaration of PROJCOST makes use of an existential quantifier in a frequently occurring manner. By allowing parameterized names to be nested in the manner of functions, the declaration of PROJCOST can be simplified to the following:

PROJCOST for {x:PROJECT, z:REAL |
<{:EMPPROJ:x}:SALARY:>:SUM:z | }

SALARY has domain EMPLOYEExREAL. The argument {:EMPPROJ:} used in <{:EMPPROJ:x}:SALARY:> is not of type EMPLOYEE, but rather of subsets of EMPLOYEE. This deliberate mismatching of types can be recognized as such and can be translated by the sytem. The term <{:EMPPROJ:x}:SALARY:> that offends the type structure can be replaced by <x:PS:>, and PS given a nested declaration as in the first declaration of PROJCOST.

This method of introducing a functional notation was described in [Gil77] for Horn clause definitions of sets, but it can be usefully employed in the more general context of DEFINE. Examples of its use will appear in later sections of this chapter, and in other chapters.

7.5. The Form and Meaning of Parameterized Set Names

To be supplied.

8. QUERIES

A query is a command to list the members of a set. The system interprets that to mean that a list of the identifiers of the set is desired. A suitable format for the command is

list setnm,

where setnm is the name of a declared set, or if a temporary declaration of the set is all that is needed, the command can take the form

list declaration.

The name of the set declared in declaration may be omitted if it is not needed elsewhere, and nested declarations may be used.

9. CONSISTENCY OF TRUTH VALUE ASSIGNMENTS

In sections 3 through 5 has been described how an assertion of DEFINE, that is type compatible with a variable+domain declaration, is assigned a truth value when each variable occurring free in it is bound to a tuple of surrogates of the type assigned to the variable by the declaration. Such an assignment of truth values is said to be **complete** if every assertion receives a truth value under a variable+domain declaration with which it is type compatible,and under any binding of tuples of surrogates of the types of its free variables to its free variables. Such an assignment is said to be **consistent** if at most one of the truth values **true** and **false** is assigned to an assertion.

In this section it will be proved that the assignment is both complete and consistent; that is that one and only one of the truth values **true** and **false** is assigned to an assertion. The proof that this is the case rests on two assumptions:

A1. Elementary assertions take the form trm:setnm where setnm is the name of a declared set, or the name of the cartesian product of declared sets.

A2. The sets referenced by a set must be declared previously to the set.

The assumption A1 will be relaxed in a later chapter. There an elementary assertion can take the form trm1:trm2, where trm1 and trm2 are terms, with the expectation that trm2 will be bound to the name of a declared set. But throughout this chapter the assumption has held.

The fact that the reference graph of a database schema is acyclic is a consequence of assumption A2. This assumption does not allow in the assertion clause a commonly used form of recursive definition, where the definition of a set is allowed to reference the set. This is the form of recursive definition used in the language PROLOG. This form of definition can be regarded, as it is in PROLOG, as a specification of a method of computation. In a later chapter it will be seen that recursively defined sets can be declared in an extended language DEFINE even though A2 is maintained, by expressing the intensions of the such sets in a manner

not suggestive of a particular method of computation.

The completion of this section will be supplied later.

10. EXTENDING DEFINE

Three ways in which the language DEFINE can be extended are:

1. Allowing trm1:trm2 as an elementary assertion, where trm1 and trm2 are terms. With this relaxation of assumption A1 of section 9, truth values may not always be assigned to assertions, and the assignment is not complete.

2. Allowing recusively defined sets to be declared.

3. Allowing primitive defined sets to be declared other than the primitive value sets. For example, allowing a primitive set to be declared that is the union of DEPARTMENT and PROJECT.

To accommodate declared sets that are possible under any one of combination of these extensions requires a reworking of the way in which truth values have been assigned to assertions. A discussion of these extensions will be left to a later chapter.

Copyright Paul C Gilmore, all rights reserved,

CHAPTER 4. INFORMATION NEEDS ANALYSIS

This chapter covers some of the finer points of information needs analysis, the naming of sets in section 1, the choice of base sets in section 2, the choice of value sets in section 3, and the role of null values in section 4. The remainder of the chapter relates domain diagrams to other commonly used diagramming techniques, to entity-relationship diagrams in section 5, data structure or Bachman diagrams in section 6, and IS_A hierarchy diagrams in section 7.

1. NAMING SETS

Like so many entities, sets are referred to by their names. But to allow them to be referred to only by their names requires that their names be unique. Although such a restriction on naming sets could be maintained, it has clearly not been maintained for the declared sets of figure 7.1 of chapter 2. The restriction has not been maintained since it requires inventing large numbers of awkward and artificial names for sets that otherwise can be given simple and intuitive ones. Consider, for example, the several uses that have been made of the set name 'NAME'. The multiple uses of 'NAME' do not generally lead to ambiguities because the name is usually used in a context that identifies the intended attribute. For example, although the assertion

x:NAME:<u>nm</u>

gives no hint as to whether 'x' is to be bound to a member of EMPLOYEE, DEPARTMENT, or PROJECT, such an assertion is rarely used without a declaration of 'x'. While it might be used in a quantified assertion such as

[**For some** x:NAME:<u>nm</u>] **true**,

which provides no hint of the type of 'x', such an assertion is unlikely to be needed. More likely is in an assertion such as

[**For some** x:EMPLOYEE] x:NAME:<u>nm</u>

where the type of 'x' is declared, and 'NAME' is no longer ambiguous.

Nevertheless, if names of sets are not to be unique it is necessary to describe some restrictions on how they are to be named, since the confusion that would arise from using a single name for all sets could not be tolerated. And once restrictions are stated, it is necessary to provide some mechanism for resolving ambiguities of names on the occasions that they arise. With a well chosen restrictions, however, those occasions should be rare.

Restrictions on the naming of sets.

1. Distinct primitive sets must be given distinct names.
2. Sets of the same type must be given distinct names.
3. Let T be a type that is not that of a value set. Let A and B be two members of the extended schema <u>schema</u>* of type T, and let C and D be any members.

Sets with domains AxC or BxD must be given distinct names, as must sets with domains CxA or DxB.

The first restirction has been followed with the naming of sets in figure 7.1 of chapter 2, since EMPLOYEE, DEPARTMENT, and PROJECT have names that are distinct from each other and from any of the primitive value sets.

The sets MALE, FEMALE, SOCCER, and BASEBALL all have EMPLOYEE as their type. Therefore the second restriction has been clearly followed as well, but it is too liberal for sets of arity greater than 1. The third restriction tightens the second for binary sets. One purpose of the restriction is to avoid ambiguities that might otherwise arise from the naming of attributes. For example, restriction 2 would not prohibit giving several of the attributes of EMPLOYEE the same name, greatly increasing the possibilities for ambiguities. However, it does allow attributes on different sets, but with the same value set, to have the same name, such as the three attributes all with the same name 'NAME'.

The definition of type ensures that once a set has been declared with a name not violating the rule for naming sets, then its name will never violate the rule as long as subsequent declarations obey the rule as well. This stability is of great importance for the integrity of a data base for an organization. The stability arises from the fact that the domain graph for a database schema is acyclic.

The rules have been designed to allow the name of a set to unambiguously identify the set in commonly occuring, but not all, contexts. Some mechanism for resolving ambiguities must therefore be provided.

Enough information is known about any declared set to unambiguously identify it, for no two sets of the same type can have the same name. A declared set can, therefore, be identified by its name and type. But since the domain of a set identifies its type, it may be sufficient to supply the domain of a set in order to resolve ambiguities. Through a prompting mechanism, the system can be expected to request the resolution of ambiguities from its users.

2. GUIDING PRINCIPLES for CHOOSING BASE SETS

The process of constructing a set schema for an enterprise is a difficult exercise requiring an ability to abstract from a large amount of information of varying quality, the essential entities of the enterprise. The process makes use of more art than science. It will never be possible to remove the art entirely from the process, but it is possible to provide guiding principles that reduce the art.

The only entities about which information can be recorded for a database are the

members of primitive sets and tuples or nested tuples of such members. Since the primitive value sets are assumed to have been declared and be available before an information needs analysis is begun, their choice is out of the hands of an analyst. But the choice of the primitive base sets is not, and the success of a SET model for an enterprise is critically dependent upon that choice.

The primitive base sets for the set schema of the XYZ corporation are EMPLOYEE, DEPARTMENT, and PROJECT. The members of these sets are entities that are not machine readable and writeable strings, but can be distinguished by a human being from other members of the same or different primitive base sets. Thus the most elementary entities encountered in SET modelling are recognized only as members of primitive sets, and each is a member of exactly one such set.

The members of the primitive defined sets INTEGER and STRING identify themselves; for example each member of STRING is a sequence of characters that can be read and written by machine. The members of the primitive base sets, on the other hand, are identified respectively by EMP# , NAME for DEPARTMENT, and NAME for PROJECT, all attributes with values from subsets of INTEGER and STRING. Although a single attribute will generally suffice as an identifier, as with these two examples, for some a pair or even a triple of attributes may be needed. Identifiers need never be provided for nonprimitive sets, however, since those that are not identifiers, such as EMPDEPT, inherit their identifiers from their immediate predecessors, while those that are do not need an identifier.

The first guiding principle to be applied in SET modelling is:

I. Primitive base sets should be chosen to be the largest sets that contain only entities of interest to the organization, and that can be easily identified and distinguished.

For example, although in the set schema for the XYZ corporation no set of customers is presently declared, in some extension of the model it may be declared. It may then be desirable to declare a set PERSON, and declare EMPLOYEE and CUSTOMER to have PERSON as their common domain. Such generalizations may not be known in advance, so that EMPLOYEE, for example, may initially be declared as primitive base. But the redeclaration of EMPLOYEE with domain PERSON after the latter set has been declared as primitive base may be the reasonable thing to do. But it would not be reasonable to declare EMPLOYEE and DEPARTMENT to have a common domain since they are unlikely to have attributes in common, and it would be difficult to declare a natural common identifier for them. It may be reasonable to declare a common domain for DEPARTMENT and PROJECT, if they have enough common attribues and if they do not in fact differ that much in character.

The second guiding principle for SET modelling is:

II. Don't declared a set as base if it can be defined in terms of previously declared sets.

As noted before, it would be a serious mistake to declare MALE as a base set. For the membership of a base set must be maintained by humans, while that of defined sets can be maintained by the system. If MALE were declared as base along with the base set SEX, the membership of both these sets would have to be maintained by humans, and furthermore they would have to be maintained so as to ensure that MALE had the membership determined by the given declaration.

The third guiding principle for SET modelling involves defined sets as well. It often occurs that the membership of a base set is restricted in some fashion. For example, the set SOCCER is a base set with domain the defined set. It would be possible to declare SOCCER directly as a base set with domain EMPLOYEE as its domain, and rely on human understanding of the restriction on soccer players to ensure that only male employees were made members of SOCCER. But again it is better to rely upon the system's capacity to maintain defined sets as stated in the third principle:

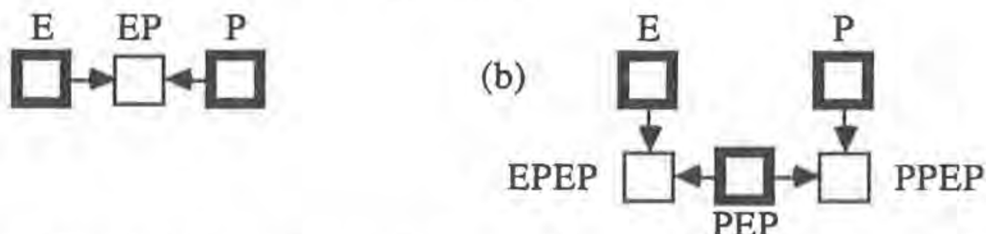
III. Whenever possible declare a defined set as the domain of a base set to enforce restrictions on the membership of the base set.

At first hand the fourth guiding principle may appear curious:

IV. Sets that can be declared with arity greater than 1 should not be declared as primitive sets of arity 1.

For example, it would be possible to declare EMPPROJ (EP), presently declared as illustrated in the domain diagram of figure 2.1 (a), as a primitive base set PEP along with two additional base sets EPEP and PPEP, as illustrated in figure 2.1 (b).

FIGURE 2



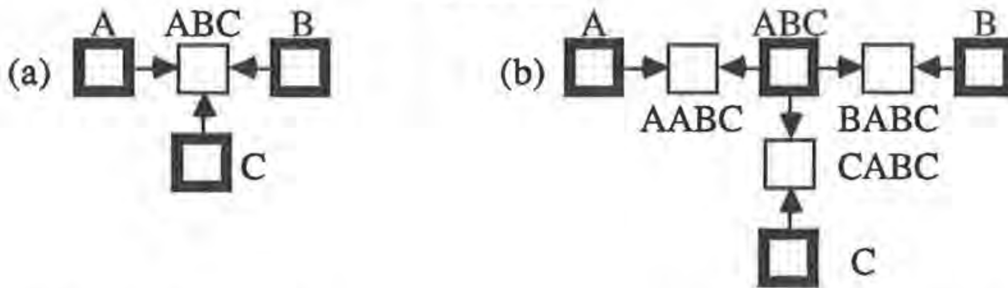
The sets EPEP and PPEP are typical fansets of a network implementation of a database, they are the "roles" of E and P in the association PEP as described by Bachman in [Bach77].

The principle states that EP should be declared, not PEP, EPEP, and PPEP. If for some reason the role sets EBEP and PBEP are needed, for example, in a network implementation of the database, they can be declared as defined sets. The principle applies equally well to sets of arity greater than two.

Although in this case the application of the principle leads to a "natural" declaration, in other cases the principle may lead to what may appear to be an "unnatural" declaration. For example, courses at universities are frequently identified by the department in which they are taught and a number. If VCOURSE# is declared as a value set the course numbers, and DEPARTMENT is the set of departments, then the principle states that COURSE should be declared as a base set with domain DEPARTMENTxVCOURSE#, rather than as a primitive base set. Were it to be declared as a base set, then it would be necessary to declare an association with domain COURSExDEPARTMENT, and an attribute with domain COURSExVCOURSE#. It would then be more difficult to declare appropriate degrees for these two associations.

The principle clearly applies to sets of arity greater than two. Consider, for example, figure 2.3.

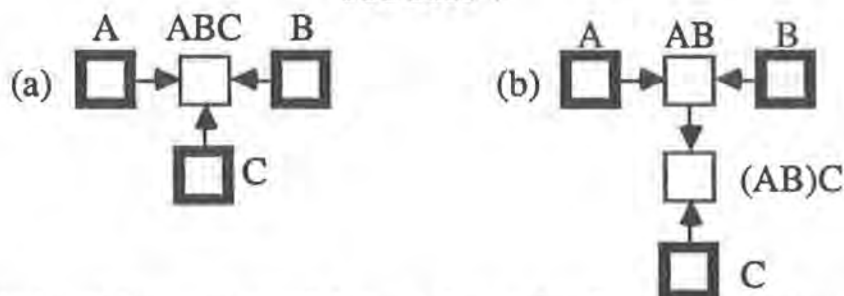
FIGURE 3



In figure 2.3(a) is illustrated a ternary association ABC with domain AxBxC, while in 2.3(b) is illustrated how it might be declared using only sets of arity 1 and 2. Again, however, the declarations illustrated in 2.3(b) are not recommended because they require declaring as base sets the associations AABC, BABC, and CABC that in 2.3(a) can be declared as defined sets.

A fifth guiding principle asserts, however, that figure 2.3(a) may not be the correct declaration for ABC either. Figure 2.4(a) repeats the illustration of the declaration of the ternary association ABC of figure 2.3(a), while 2.4(b) illustrates how it could be declared as a binary association (AB)C, once the binary association AB is first declared.

FIGURE 4



The association AB consists of those pairs <a,b> for which <a,b,c> is a member of ABC for some c, while (AB)C consists of those pairs <<a,b>,c> for which <a,b,c> is in ABC. The lower degree of (AB)C on AB is necessarily 1. Whether 2.4(a) or 2.4(b) is the correct declaration depends first upon the upper degree of (AB)C on AB. When that upper degree is 1, then 2.4(b) is clearly the correct declaration, while if it is *, then 2.4(b) is more rarely the natural declaration. The fifth and final guiding principle presented here is therefore:

V. A base set ABC of arity 3, should be declared as a base set (AB)C of arity 2 whenever its upper degree on AB is 1.

Again the principle can be generalized to sets of greater arity.

3. The CHOICE of VALUE SETS

The discussion of value sets in chapter 2 will be enlarged here. In particular a discussion of the date value sets in the EXCEL spreadsheet program will be used to illustrate the discussion. Also the question of "code or not to code" raised by Brian Mullen in his Centre course will be discussed.

4. NULL VALUES

A null value is a member of a value set of an attribute that is used to convey the fact that a "normal" value for the attribute is not available. There are two reasons why a normal value may not be available, and there are two kinds of null values used to express the two reasons, the **N/A** or **not applicable** null value, and the **D/K** or **don't know** null value.

The N/A null value is used to express that a given attribute does not, and should not, have a value for a particular entity. The purpose of introducing a N/A null value into the value set of an attribute is to extend the set for which the attribute has values. Consider the following example. Let VSOC be declared

VSOC for {x:STRING | x='YES' **or** x='NO' **or** x='N/A'}

and let SOC be the defined set:

SOC for {x:EMPLOYEE, v:VSOC | (x:MALE **and** x:SOCCER **and** x='YES') **or**
(x:MALE **and not** x:SOCCER **and** x='NO') **or**

(x:FEMALE **and** x='N/A')}

Without the availability of the N/A value in the value set VSOC, the attribute SOC could only be given proper meaning for male employees. Of course, without N/A available in VSOC, SOC could be given the value NO for female employees, but the attribute so defined would be misleading. It would suggest that the fact that no female employee is a member of the soccer team is an accident of time rather than a result of policy. With the N/A value available, there is less likely to be an incorrect assignment of a female employee to the soccer team.

The D/K null value is used to express that a given attribute should have a value for a particular entity, but that the value is not known at the present time to those responsible for maintaining the base of information. A D/K null value, therefore, does not express information about an entity, but rather information about a lack of knowledge on the part of those responsible for having knowledge. For example, the attribute SOC would be recorded as having the value D/K for a particular male employee, if it was not known whether the employee played on the soccer team.

The treatment of D/K null values is fundamentally different from the treatment of N/A null values, and requires a careful examination of what is meant by an entity.

Base sets of entities are sets whose membership can only be understood by humans. To determine whether an entity is a member of a base set, it is necessary for the people responsible for maintaining the base of information to possess information about the entity. To know whether a male employee is a member of SOCCER it is necessary to know whether the employee plays on the soccer team. Further, once an entity is known to be a member of a set, it may be necessary to know other information as well. For example, when a person is known to be a member of EMPLOYEE, it is necessary to know the person's employee number, name, address, sex, and department, since the attributes and the association EMPDEPT all have lower degree 1 on EMPLOYEE.

The fact that so much information is needed about a person that is a member of EMPLOYEE, means that there will be many occasions when all the information is not available. If a means is not provided within the information system for dealing with these occasions, then it is probable that informal files of information will be created for dealing with them, and as a consequence, the information system will be incomplete.

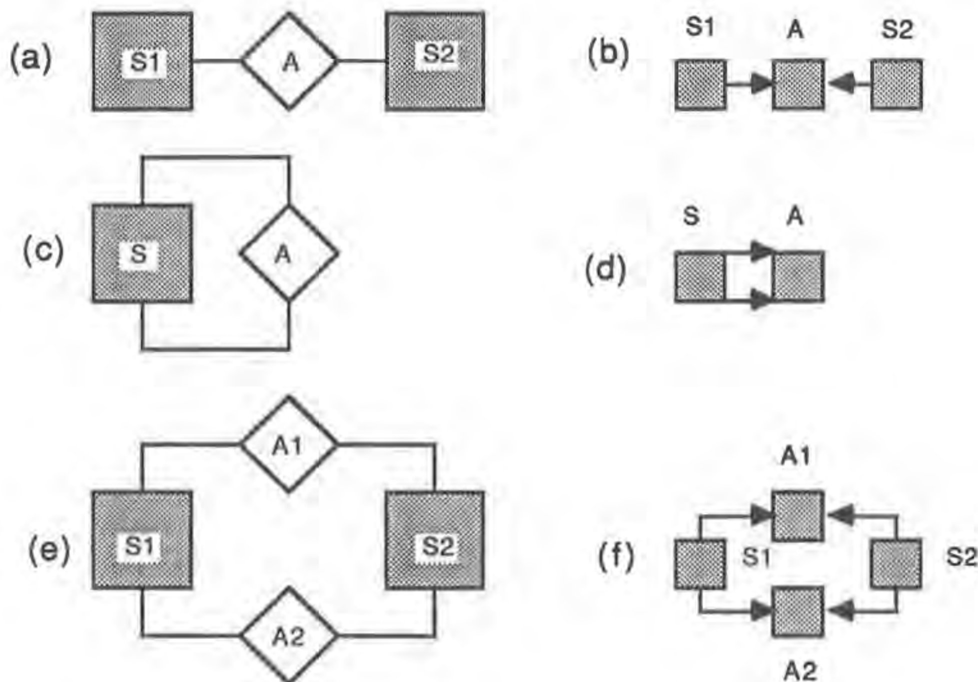
5. ENTITY-RELATIONSHIP DIAGRAMS

Entity-relationship diagrams were introduced by Chen in [Chen76] as a way of recording information obtained during an information needs analysis of an

enterprise. The domain diagrams introduced in chapter 2 to illustrate the sets declared in a set schema for an enterprise are closely related.

Figures 5.1 and 5.2 illustrate corresponding entity-relationship and domain diagrams. Figures 5.1 (a), (c), and (e) are commonly used entity-relationship diagrams, and (b), (d), and (f) illustrate the domain diagrams. The difference here is that the lines that appear in the entity-relationship diagrams are replaced with arrows in the domain diagram.

FIGURE 5.1



In figure 5.2, less commonly used entity-relationship diagrams are illustrated with their corresponding domain diagrams.

FIGURE 5.2

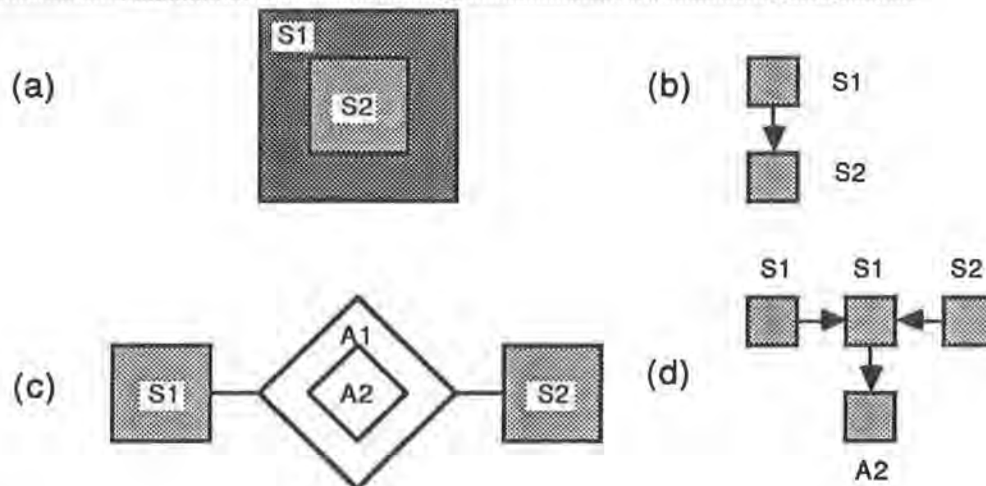
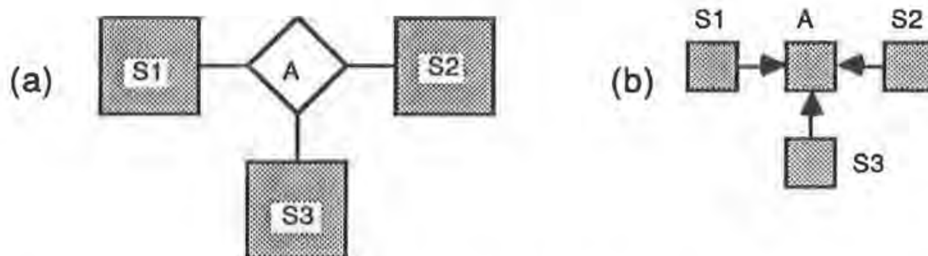


Figure 8.3 illustrates how domains that are supersets are represented in set assocation diagrams and in domain graphs. An example of the situation of figure 8.3 (a) is EMPLOYEE and MALE and an example of the situation of figure 8.3 (c) is EMPPROJ and LEADER.

The situtation illustrated in figure 5.2 (c) must be carefully distinguished from the situation illustrated in figure 5.3 (a). The latter illustrates the representation for a ternary association A with domain the cartesian product, in some order, of S1, S2, and S3. As can be seen in figures 5.2 (d) and 5.3 (b), the situations are distinguished in a domain graph by the directions of the arrows.

FIGURE 5.3

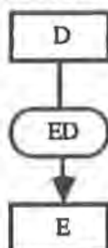


A large number of variations and extensions to the entity-relationship diagrams originally proposed by Chen have been proposed with the intent of representing all the information that can be given in the declaration of a base set. Often the diagrams are proposed with the understanding that they provide all the information needed. But as has been noted before, they are not a substitute for the declaration of sets, but only a means of illustrating the declarations.

6. DATA STRUCTURE or BACHMAN DIAGRAMS

Using a data structure diagram, often called a **Bachman diagram** after its inventor, the relationship between the sets DEPARTMENT (D), EMPLOYEE (E), and EMPDEPT (ED) can be represented in a single diagram as shown in figure 6.1.

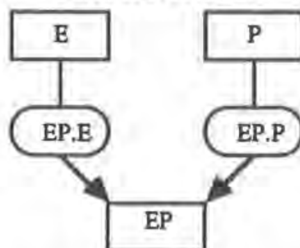
FIGURE 6.1



The arrow is directed from D through ED to E because the association ED has degrees <1,1> on E and <1,*> on E. That is, for each department there is a set of employees that are assigned to it. Actually, as discussed in chapter 6, the box labelled D represents the **record type** for DEPARTMENT, so that it contains a field for recording the value of every functional attribute on D; for the case at hand the name of the department is the only attribute that has been declared. Similarly, the box labelled E represents the record type for EMPLOYEE, so that it contains a field for recording the value of every functional attribute on E, but does not contain a field for recording the name of the department to which an employee has been assigned. The oval labelled ED represents the ED association between E and D. In these diagrams it is regarded as having a direction from D to E, indicated by the arrow.

A more complicated Bachman diagram is illustrated in figure 6.2, where as before E abbreviates EMPLOYEE, and now P abbreviates PROJECT, and EP abbreviates EMPPROJ. Note the oval boxes are now labelled with the projections of EP on E and P.

FIGURE 6.2



This diagram arises, in contrast to that of figure 6.1, because the lower degree of EP on E and on P is each 0. The two diagrams can be combined into one as illustrated in figure 6.3. In figure 6.4 a complicated diagram is illustrated in figure 6.4.

FIGURE 6.3

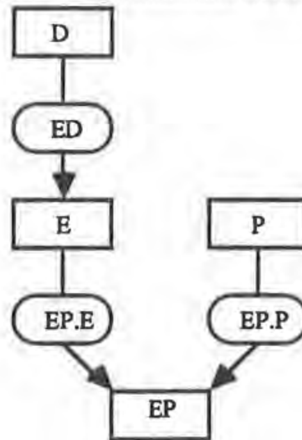
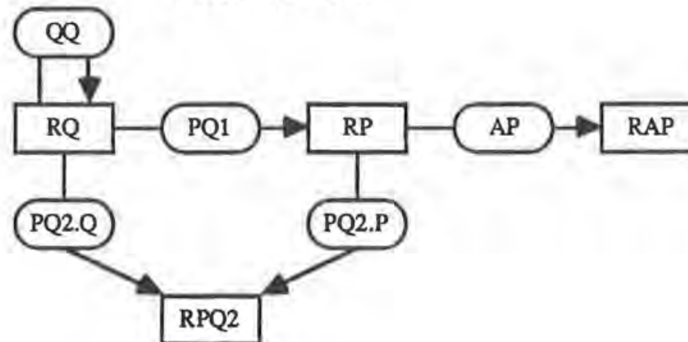


FIGURE 6.4



7. IS_A HIERARCHIES

IS_A hierarchies, sometimes called semantic nets, are widely used in artificial intelligence work to represent relationships between sets of entities. Domain diagrams are a generalization of the hierarchies in the sense that domain diagrams can illustrate the relationship between a set and its immediate domain predecessors, even when it has more than one, while IS_A hierarchies only illustrate the relationship between a set and its domain. However, IS_A hierarchies allow for individual entities to label nodes, while the nodes in a domain diagram can only be labelled with declared sets. A thorough discussion of them is given in [Sowa84].

BIBLIOGRAPHY

[Bach69] Bachman, C.W., Data structure diagrams. Data Base 1 (1969), 4-10.

[Bach77] Bachman, C.W., The role concept in data models. Proc. 3rd Int. Conf. Very Large Databases (1977).

[BCP86] Benoit, Christophe, Caseau, Yves, and Pherivong, Chantal, The LORE Approach to Object Oriented Programming Paradigms. Memo C29.0, Laboratoires de Marcoussis, Centre de Recherches de la C.G.E. (1986).

[Bern76] Bernstein, P.A., Synthesizing third normal form relations from functional dependencies. ACM Trans. Database Syst. 1 (1976), 271-298.

[Blac85] Black, Michael Julian, Naive Semantic Networks, Final Paper, Directed Study in Computer Science, Univ. of B.C., (1985).

[Chen76] Chen, Peter Pin-Shan, The Entity-Relationship model - toward a unified view of data, ACM Trans. Data Base Syst., 1 (1976), 9-36.

[Chen77] Chen, Peter Pin-Shan, The Entity-Relationship model - A basis for the enterprise view of data, AFIPS Conf. Proc., 46 (1977).

[Chen85] Chen, Peter Pin-Shan, Database Design Based on Entity and Relationship,in: Yao, S. Bing (ed.). Principles of Database Design, Volume I, Logical Organizations (Prentice-Hall, 1985) 174-210.

[Clar78] Clark, K.L., Negation as Failure, in: H. Gallaire and J. Minker (eds.), Logic and Data Bases (Plenum, New York, 1978).

[Codd70] Codd, E.F., A relational model of data for large shared data banks, Comm. ACM 13 (1970), 377-387.

[Codd72] Codd, E.F., Relational completeness of data base sublanguages, in: R. Rustin (ed.), Data Base Systems (Prentice-Hall, 1972)

[Codd79] Codd, E.F., Extending the Database Relational Model to Capture More Meaning, ACM Trans. Database Syst., 4 (1979).

[Cox86] Cox, Brad J., Object-Oriented Programming (Addison-Wesley, 1986)

[Date83] Date, C. J., An Introduction to Database Systems, Volume II

(Addison-Wesley 1983)

[DJNY83] Davis, Carl G., Jajodia, Sushil, Ng, Peter Ann-Beng, and Yeh, Raymond T., Entity-relationship approach to software engineering, (North-Holland, 1983)

[DiDa86] Dittrich, Klaus, and Dayal, Umeshwar, (Eds.) Proc. Int. Workshop Object-Oriented Database Systems, ACM and IEEE (1986) 23-26

[Flan86] Flannigan, Tim, The Consistency of Negation as Failure, J. Logic Programming, 2 (1986), 93-114.

[FuNe86] Furtado, Antonia L.and Neuhold, Erich J., Formal Techniques for Data Base Design (Springer-Verlag, 1986)

[Gil86] Gilmore, Paul C., Natural deduction based set theories: a new resolution of the old paradoxes, J. Symb. Logic, 51 (1986), 393-411.

[Gil87a] Gilmore, Paul C.,The SET Conceptual Model and the Domain Graph Method of Table Design, Tech. Report 87-7, Dept. of Comp. Sc., Un. of B.C. (March 87)

[Gil87b] Gilmore, Paul C., Justifications and Applications of the SET Conceptual Model, Tech. Report 87-9, Dept. of Comp. Sc., Un. of B.C. (April 87)

[Gil87c] Gilmore, Paul C.,Attribution by Default, Tech. Report 87-26, Dept. of Comp. Sc., Un. of B.C. (July 87)

[HaMc81] Hammer, Michael, and McLeod, Dennis, Database description with SDM: a semantic database model, ACM Trans. Database Syst., 6 (1981), 351-386.

[Kell85] Keller, Arthur M, Updating Relational Databases Through Views, Stanford Computer Science Department Tech. Report STAN-CS-85-1040 (Feb 1985)

[Kent78] Kent, William, Data and Reality (North-Holland, 1978)

[Kent83] Kent, William, Fact-based data analysis and design,in: [DJNY83] 3-54.

[LeSa83] Lenzerini, Maurizio, and Santucci, G., Semantic integrity and specifications,in: [DJNY83] 529-550.

[LyKe86] Lynback, Peter, and Kent, William, A Data Modelling Methodology for the Design and Implementation of Information Systems, in: [DiDa86] 6-17.

[Mark85] Mark, Leo. Self-describing database systems - formalization and realization. Tech. Rep. #1484, Dept. Comp. Sci. Un. Maryland (1985)

[Morr] Morrison, Roderick,Implementation Considerations for a Set Based Data Model and its Data Definition/Manipulation Language, PhD thesis, Dept. of Comp. Sc., Un. B.C. In progress.

[MW80] Mylopoulos, J., and Wong, H., Some features of the TAXIS data model, Proc. 6th Int. Conf. Very Large Databases, Montreal (1980)

[Rock81] Rock-Evans, Rosemary, Data analysis (IPC Electrical-Electronic Press Ltd, Sutton, Surrey, England 1981)

[SAAF73] Senko, M.E., Altman, E.B., Astrahan, M.M., and Fender, P.L.,Data structures and accessing in data-base systems, IBM Syst. J. 12 (1973), 30-93.

[Ship81] Shipman, David W.,The functional data model and the data language DAPLEX, ACM Trans. Database Syst., 6 (1981), 140-173.

[Sowa84] Sowa, J.F., Conceptual Structures: Information Processing in Mind and Machine (Addison-Wesley, 1984)

[TaFr76] Taylor, Robert W. and Frank, Randall L., CODASYL Data-Base Management Systems, ACM Comp. Sur.. 8 (1976), 67-103.

[TYF86] Teorey, Toby J., Yang, Dongqing, and Fry, James P., A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model, ACM Comp. Sur., 18 (1986), 197-222.

[TrLo87] Tryon, D.C. and Lloyd, D.G., Information Resource Depository: History, Current Issues, and Future Directions, Pacific Bell, A Pacific Telesis Company. Presentation to Canadian Information Processing Society, Vancouver, Canada, February 1987.

THE SET CONCEPTUAL MODEL
and the
DOMAIN GRAPH METHOD
of
TABLE DESIGN

Paul C. Gilmore

Technical Report 87-7
March 1987

ABSTRACT: A purely set-based conceptual model SET is described along with a specification/query language DEFINE. SET is intended for modelling all phases of database design and data processing. The model for an enterprise is its set schema, consisting of all the sets that are declared for it.

The domain graph method of table design translates the set schema for an enterprise into a table schema in which each table is a defined user view declared as a set in DEFINE. But for one initial step, the method can be fully automated. The method makes no use of normalization.

Two kinds of integrity constraints are supported in the SET model. Although these constraints take a simple form in the set schema for an enterprise, they can translate into referential integrity constraints for the table schema of a complexity not previously considered.

The simplicity of the constraints supported in the SET model, together with the simplicity of the domain graph table design method, suggests that a conceptual view of an enterprise provided by the SET model is superior to the lower level data presentation view provided by the relational model.

The research reported on in this paper has been supported by grants of the Natural Sciences and Engineering Research Council of Canada

1.Introduction

1.1. Why Another Model?

To master the complexity of the information processed by a typical enterprise, it is necessary to present abstract descriptions, called data models, or in their more abstract form, conceptual models of the information [Knt78, BMS84]. The earliest models, the hierarchical and network, are low level models in the sense that they are abstract descriptions of implemented data structures that store the required data. The relational model is a higher level model in that it provides abstract descriptions of the data as it is presented to the users in the form of tables. The relational model has freed users from most implementation concerns, but it forces users to be concerned with the presentation of data at the earliest stages of database design when a higher level conceptual view of information needs is more appropriate. Consequences of this insufficiently abstract view of information are evident in the unnecessarily complex normalization methods of table schema design used to ensure the absence of update anomalies, and also in the inability of the relational model to deal with referential integrity in a uniformly simple fashion.

The entity-relationship (ER) model has found a way to avoid both the excessive implementation concerns of the hierarchical and network models, and the restrictive presentation concerns of the relational model.[Chen76,77] The strength of the ER model is that it is neither implementation nor presentation oriented, but rather conceptually oriented. The weakness of the ER model, on the other hand, is that it lacks a sound foundation upon which a management system might be based. The primary motivation for the development of the purely set-based data and conceptual model SET and its specification/query language DEFINE has been to provide such a foundation. There are several reasons why this is necessary:

1. For the unified view of data proposed in [Chen76] to be fully achieved, a database is needed that is capable of recording a high level conceptual model of an enterprise and at the same time of providing the tables for a relational database schema as a defined user view in its specification/query language.

Such a management system will avoid the hand translation process that intervenes between the high level and the user table view of an enterprise [Chen77, HaMc81, Rock81, Knt83a, Chen85, TYF86]. As the enterprise's information needs evolve, such a hand translation will inevitably result in either the conceptual model or the tables being out of date, or very likely the conceptual model, with all its valuable high level information, just being discarded. The SET model integrates the two stages. In the first stage, sets are declared and recorded in a set schema that is a conceptual model for the enterprise; the tables needed for the second stage are just user views of the enterprise defined within the specification/query language DEFINE. As the enterprise's information needs evolve, the model is updated by the declaration of new sets, or by the removal of sets from the set schema. The users'

tables are then either automatically updated or if necessary redefined. As illustrated in section 3, much of the design and definition of the tables can be automated.

In as much as the SET model is object-oriented in the sense of [Ditt86], it contributes to the requirement for a high quality database design method called for there.

Queries for a relational database supported by a management system based on the SET model can be posed in DEFINE, or in a language for a relational database schema. The latter would then be translated into DEFINE queries.

2. The modelling process used in ER modelling requires a greater discipline than is now possible.

A tentative beginning is provided in section 2.13 to removing some of the art from conceptual modelling, and in providing a basis for some of the decisions that must be made while modelling. Several principles are stated that should be used in designing conceptual models. These principles cannot be understood without a rigorous foundation such as provided by the SET model.

3. A provably sound foundation is needed for databases that can reference and describe themselves.

A conceptual model for an enterprise requires the capture of large amounts of what is called meta-data that records information about the common data processed in the enterprise's daily activities [Lyke86, DKM86]. As that information is gathered, the meta-data expressing it must be recorded so that it can be queried and updated like any other data. Traditionally data dictionaries have been used for this purpose but they beg the question "where is the meta-data for a data dictionary to be stored?". To avoid an infinite sequence of data dictionaries it is necessary to begin with a database capable of recording information about itself. Although the need for such self-describing databases has been recognized before [LyKe86, Mark85, TrLo87], a concern that they raise has gone unnoticed: If such a base is not to fall prey to the paradoxes and contradictions of set theory that shook the foundations of mathematics early in this century, then it must be provided with a provably sound foundation. The necessity for this has been demonstrated again more recently; the semantic networks described in [Sowa84] are subject to the same contradictions as naive set theory [Blac85].

4. A fully unified model of an enterprise is needed that at the same time can give a conceptual view of the enterprise, a user's view of data as it is presented, a data administrator's view of data as it is stored, and a programmer's view of the processing of the data.

The construction of such a unified model of an enterprise is ambitious, but is

nevertheless necessary. For without such a unified model there will always remain the need for moving from one model to another when dealing with different aspects of information and its processing, with the probable consequence of the different models becoming inconsistent with one another. The need for such a unified model, raised in [Knt78] and repeated in [LyKe86], is argued forcefully in [TrLo87].

5. Sound foundations are needed for knowledge base systems capable of dealing with incomplete information.

Such systems require distinctions that can only be made precise in a formally described model. For example, it is necessary to distinguish between sets with intensions that can only be understood by humans, and sets with intensions expressed in a language like DEFINE that can be understood by both humans and machines. It is also necessary to distinguish between an internal surrogate for an entity maintained by a management system and a character string that is an identifier for the entity, in order to deal adequately with identity [Knt78, Codd79, KhCo86].

1.2. Summary

The concept of a set or class is one of the most fundamental of mathematics and has been incorporated into scientific and natural languages, as well as used extensively in database theory and practice. It is explicitly used in the relational and entity-relationship models, but also in the entity set [SAAF73], semantic [HaMc81], and functional data models [Ship81, LyKe86], in the system TAXIS [MyWo80], in the techniques described in [FuNe86], and implicitly used in the network data model [TaFr76]. But the concept of set used is an intuitive one and is combined with other related but independent concepts. The SET model, on the other hand, uses set and ordered pair as its only fundamental concepts, with the mathematical foundation for these concepts being provided by the provably consistent set theories of [Gil86a], while other needed concepts are defined in terms of these.

The purpose of this paper is to provide an introduction to a simple form of the SET model, and demonstrate that it supports a unified process of conceptual and presentation modelling called for in (1). A secondary purpose is to demonstrate that it also can contribute to (2). That it can contribute to the remaining three demands, and that DEFINE can be used as a query language, is demonstrated in [Gil87].

In section 2, the elementary features of the SET model and the language DEFINE are introduced through an exercise in conceptual modelling for an enterprise called Simple University, a university so simplified that it does not have any students. The exercise is designed to illustrate how the integrity constraints supported by the model can be used to formalize semantic information about an enterprise.

In section 3 a table design method is described and a table schema for Simple University obtained. Because the set model for Simple University and the presentation model in the form of the table schema are both described in the same

language, and because the tables are defined user views, it is possible to prove that the tables correctly record all required data; that is, they are provably free from anomalies. This justification of the method is stated as a theorem in 3.6, the proof of which is postponed to [Gil87]. The method, unlike [TYF86], makes no use of normalization, although it is related to the synthetic method of [Bern76], and the method of [Knt83a].

Using the Simple University example, the integrity constraints supported in the SET model are compared in section 4 with those customarily supported in the relational and network models. The conclusions of the paper are presented in section 5.

Of the cited papers [LyKe86] comes closest to presenting a model with the same intended scope as SET. There are several differences that can be noted between the IRIS model briefly presented there and SET. One superficial distinction is that IRIS is based on functions, while SET is based on sets. But the types of IRIS are sets, and functions are admitted as types. A more profound difference is in the management of declarations. The simple form of the model demands a discipline in the declarations of sets that is absent from IRIS. This may be important to conceptual modelling and to the design of databases.

A detailed description of the syntax and semantics of DEFINE is not given in the paper, but is only introduced as it is needed. It is assumed that the reader is familiar with the semantics for the boolean operators **and**, **or**, and **not**, and for the boolean quantifiers [**For some** ...] and [**For all** ...]. However, the demands on prior knowledge of mathematical logic are minimal throughout the paper.

1.3. Acknowledgements

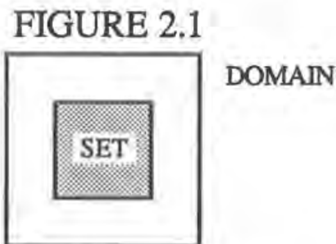
Frege's invention of the predicate logic provided a new standard of precision for mathematical theories that logicians have been exploiting for nearly a century. A demand for a computer implementation of a theory can contribute not only to that standard, but also to the usefulness of the theory. The hard-nosed views of Roderick Morrison in particular, but also of Brian Mullen and of students both undergraduate and graduate, including Michael Black, Julie Abrahamson, Georgis Tsikinis and Michael Kreykenbohm, have contributed to the SET model.

2. Elementary Features of the SET Model

A distinctly computer science view of sets is taken in the SET model. No set can be presumed to preexist; each required set must be explicitly declared. All the declared sets of an enterprise form a **set schema** for the enterprise. A set has an **intension** and an **extension**. The intension of a set is the property that an object must possess to be a member of the set; the extension of a set is the collection of objects satisfying the intension of the set. The intension of a set is to be thought of as time invariant, while the extension of a set changes as circumstances change. The intension of a set may be expressed in a natural language statement intended for human understanding only (the **base** sets), or may be expressed in a language like

DEFINE that can be understood by both humans and machines (the **defined** sets). The extension of a set is drawn from its **domain** as illustrated in figure 2.1.

FIGURE 2.1



The domain of a set S is the extension of the cartesian product of one or more previously declared sets. Each set DS occuring in the cartesian product is an **immediate domain predecessor** of S of **multiplicity** the number of times DS occurs in the product. To avoid an infinite regression of domains, it is necessary to assume that some sets are declared without domains; these are the **primitive** sets.

2.1 Simple University

To illustrate these and other elementary features of the model their application to the following example will be demonstrated: At Simple University (SU) there are departments, each identified by its name. Each employee of SU is a member of exactly one department, and is identified by a number. Each course taught at SU is identified by a course number and the department responsible for it. An academic department is one responsible for some courses. An instructor is a member of an academic department who is competent to teach some of the courses of the department to which he/she belongs; an instructor is currently teaching some of the courses he/she is competent to teach. When taught, a course is taught by exactly one instructor. Exactly one instructor of an academic department is manager of the department. Exactly one member of a nonacademic department is manager of the department. Although in a more realistic example employees, departments, and courses, would have attributes declared for them, they will be largely ignored so as to concentrate on more important features of conceptual modelling.

The set of departments of SU is a primitive base set. This set can be declared as follows:

D for {D || the current departments of SU }

The name D of the set appears to the left of 'for'. The fact that D is a primitive set is indicated by the second occurrence of D, where normally the domain of a declared set is recorded. The two vertical bars separate the domain declaration from a comment that expresses the intension of D. Since only humans can determine whether an entity satisfies the intension of D, it is a base set. In declarations of nonprimitive base sets and of defined sets, a machine interpretable expression appears between the two vertical bars.

2.2 Internal Surrogates and Identifiers

The result of the declaration of a primitive base set for a management system is the preparation of a file where internal surrogates of members of the set can be recorded. When a user of the system indicates that a new member is to be added to the set, the system generates a new distinct internal surrogate for the entity. Thus at any time the internal surrogates recorded by the system as members of the set should be in one-to-one correspondance with the entities that users perceive as members of the set.

The internal surrogate of a department must be distinguished from an identifier of the department. The former need not be known to users of the system, while the latter, as stated in the description of SU, is its name. The names of departments are particular strings of characters that can be written and read by both humans and machines. Such strings are the means by which humans and machines communicate with each other. For example, if 'ENGLISH' is the name of one of the departments of SU, then the management system will have been instructed to associate the string with the internal surrogate of a member of the set D that was created when the English Department was added to the set. By using the string 'ENGLISH' in an appropriate way, a user can convey information to the management system about the department represented by the internal surrogate, and subsequently ask for information about the department.

2.3. The Kernel Schema and Value Sets

The strings that are names of departments are members of a set STR of all strings that is supported by the management system. In programming language terminology, it is a data type. Another such data type is INT, the set of integers. Each of these is a primitive defined set. They are primitive sets since there is not a previously declared set from which their extensions can be drawn, and they are defined since the management system can determine whether an entity is or is not a member of it.

The declaration of STR is:

STR for {x:STR | x system defined | the set of strings of characters }.

The assertion 'x:STR' in the declaration expresses two things: First that the domain of STR is STR, and second that the variable 'x' is declared to be a member of STR. This is typical of the declaration of a defined set. The assertion 'x system defined' expresses that the membership of STR is determined by the system; it is assumed to be a primitive assertion of DEFINE that can be read and understood by both humans and machines. Finally the phrase 'the set of strings of characters' is an informal comment describing the membership of STR in human understandable terms only; such a comment may be added to the declaration of any defined set, but must not be confused with the intension of the set.

The declaration of INT is:

INT for{x:INT | x system defined | the set of integers }.

This declaration, along with that of STR, are among those declarations of sets needed by the system managing set schemas; they form what is called the **kernel schema** of the system. Users of the system cannot change the kernel schema in any way, although the declarations must, of course, be available to users. Kernel schemas are discussed at greater length in [Gil87].

It is possible, and at times useful, to regard the members of INT and STR as entities with internal surrogates and surface identifiers, just like the members of base sets, even though they are defined sets [Morr]. In this paper, however, the simpler view of these sets as sets of strings will be maintained.

Nonprimitive sets must also be declared in the kernel schema:

≤ for {x:INT, y:INT | <x,y> system defined | x is less than or equal to y }.

Since the domain of ≤ is the cartesian product of previously declared sets, ≤ is a nonprimitive set; INT is an immediate domain predecessor of ≤ of multiplicity 2. The declaration of ≤ makes it possible to express that an integer x is less than or equal to an integer y by the usual infix notation 'x ≤ y', although the standard form of the infix notation in DEFINE is 'x:≤:y', the colons being used to separate the two arguments from the name of the set.

Another example of a nonprimitive set declared in the kernel schema is:

L for {x:STR, y:INT | <x,y> system defined | x is a string with length y }.

L is a nonprimitive defined set with extension those pairs <x,y> admitted by the system. The declaration of L makes it possible to express that a string x has length an integer y with the assertion 'x:L:y'. The same fact can be also expressed with the assertion '<x,y>:L'. The two assertions 'x:L:y' and '<x,y>:L' have exactly the same meaning, but the infix form will be seen to be more convenient at times.

L is usually thought of as a function; that is, it takes an argument x and returns a value y. Expressed in the usual functional notation, y is L(x). The intension of L, when elaborated, describes how the value L(x) is determined from the argument x. But the extension of L is nevertheless a set of pairs <x,y>, where x is a member of STR and y is a member of INT. To avoid syntactic confusions, the usual functional notation will not be used, but rather a notation that emphasizes the fact that functions are just sets of pairs. For example, instead of 'L(x)', the notation '{x:L:}' will be used. The semantics of the notation is provided in the manner of the functional notation of [Gil77].

Not all members of STR are admitted as names of departments, but only those of length at most 10 characters. The subset of such strings is declared as follows:

VN for {x:STR | {x:L:} ≤ 10 | a value set for names of departments }.

The assertion {x:L:} ≤ 10 expresses that for x to be a member of VN, it must have length not exceeding 10.

The comment in VN refers to it as a **value set**. These sets are defined recursively as follows: Each of the primitive defined sets STR and INT is a value set, the cartesian product of value sets is a value set, and any defined set with domain a value

set is a value set. The sets ≤, L, and VN are all value sets by this definition, but unlike the first two sets, VN is not declared in the kernel schema, but is a user declared set. Generally the particular form of such sets is one of the least important considerations in the design of a set schema. Therefore good design practice dictates that the statement of the intension of value sets within DEFINE be postponed as long as possible.[Knt83a]

2.4. Degrees of Base Associations

With each member of D, a unique member of VN must be associated. The required declaration is:

DN for {D, VN | <1,1>, <0,1> | a unique name from VN is associated with each department }.

The domain of DN is the cartesian product of its immediate domain predecessors D and VN listed in that order in the declaration. The intension of DN follows the second of the two vertical bars in the form of a comment. Since the comment cannot be interpreted by machine, DN is a base set: The names to be given to the departments of SU are determined by humans. The machine interpretable expression between the two vertical bars is a **degree** declaration. It consists of two pairs of degrees, with the first pair relating to D and the second pair to VN, which is in the order specified in the domain declaration.

The first element of a pair of degrees is the **lower degree** and may be 0 or 1, while the second is the **upper degree** and may be 1 or *. The lower degree of the first pair <1,1> of degrees states that for every member of D there is at least one member of VN associated with it; that is, each department must have a name associated with it. The upper degree of the first pair states that with a member of D no more than one member of VN can be associated; that is, each department must have at most one name associated with it. The constraints expressed by the first pair <1,1> of degrees ensures, therefore, that each department has a single name associated with it.

The lower degree 0 of the second pair <0,1> of degrees expresses that not every member of VN need be associated with a member of D; that is, not every member of VN is necessarily the name of a department. The upper degree 1 of the second pair expresses that a member of VN can be associated with at most one member of D; that is, the same name is never assigned to two departments. The constraints expressed by the second pair <0,1> of degrees ensures, therefore, that if a member of VN is used as a name, then it is used as the name of a single department. The constraints expressed by the two pairs of degrees <1,1> and <0,1> ensures that each department has a single name that is unique to the department.

The need for degrees has been widely recognized. In [Knt83a, LyKe86] they are called respectively the least and maximum participation, and in [LeSa83] the minimum and maximum cardinalities. In [LyKe86] the upper degree * is denoted by m. In [TYF86] the upper degrees are referred to as the cardinalities of the connectivity of a relationship, while the lower degrees are described as optional or

mandatory connectivity. They have been called degrees in the SET model because they can be regarded as lower and upper bounds on the degrees of nodes of bipartite graphs representing the associations. For example, the bipartite graph representing the DN association has one set of nodes representing the members of D and one set of nodes representing the members of VN. The undirected edges of the bipartite graph connect each member of D to the member of VN that is DN associated with it. The degree of a node in a graph is the number of edges that connect to it.

Degrees can be declared for base sets that have any number of immediate domain predecessors. For example, a base set P with domain Q can be declared to have lower degree 0 or 1, 0 meaning that the extension of P is a proper subset of the extension of Q, and 1 meaning that they are extensionally identical. Since there is no need to declare P as a base set if it is to be extensionally identical to Q, only a lower degree of 0 is needed. Also only an upper degree of 1 makes sense, since for each member of P there is exactly one member of Q.

Degrees can also be declared for sets with more than two immediate domain predecessors. For example, if A is declared with base PxQxR, then the lower degree of A on P is the least number, for any p, of pairs <q,r> for which <p,q,r> may be a member of A. The upper degree of A on P is similarly defined.

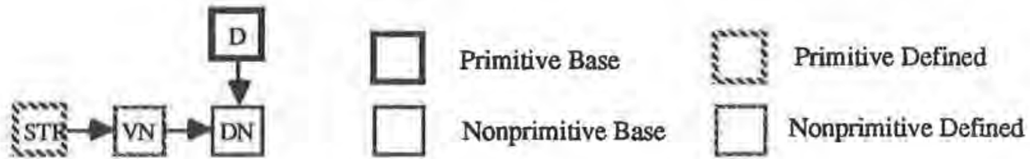
Degrees are declared only on base sets, and are **integrity constraints** expressing restrictions or constraints on the membership of the sets. They should be regarded as part of the intension of the sets; they are a machine readable part of an otherwise machine unreadable intension. The degree constraints, together with the implicit inclusion constraints expressed when the domain of a set is declared, are the only integrity constraints that can be expressed in the SET model.

Degree constraints can provide the foundation for the dependency theories of the relational model and are shown, in section 3, to be fundamental to the design of a table schema that can record data without anomalies. In [Gil87] the satisfiability and maintainance of degree constraints are discussed.

2.5. A Domain Graph

Graphical representations are an important way of providing a user with an overview of declarations that have been made. The diagrams of the ER model and the Bachman diagrams [Bach69] of the network model are examples of such representations, as are illustrations of domain graphs for the SET model, an example of which is given in figure 2.2. It must be stressed, however, that an illustration of a domain graph is not a substitute for set declarations, but only provides an overview of the domain declarations of the declared sets.

FIGURE 2.2



A **domain graph** of a set schema is a directed graph with nodes labelled with declared sets of the schema, at most one node for each set, and with directed edges <nde1, nde2> for which the tail nde1 is labelled with an immediate domain predecessor of the head nde2. In figure 2.2 each of the sets D, STR, VN, and DN labels a node, and the arrows point from the immediate domain predecessor STR of VN to VN, and from the immediate domain predecessors D and VN of DN to DN. The boxes representing the nodes of this simple domain graph are drawn with different lines simply to emphasize the four kinds of sets that have been declared.

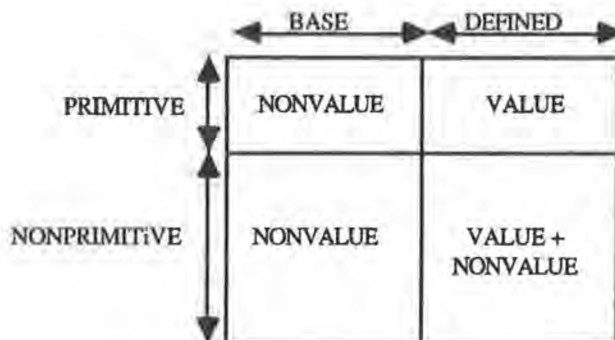
In the domain graph illustrated in figure 2.2, at most a single edge appears between any two nodes. That need not always be so. For example, in a domain graph with nodes labelled with the system declared sets ≤ and INT, there would be two edges directed from the node labelled with INT to the node labelled with ≤, since the immediate domain predecessor INT of ≤ has multiplicity 2.

Since the immediate domain predecessors of a set must be declared previously to the set, a domain graph of a set database schema is necessarily directed acyclic; that is, it is not possible to follow a path from a node back to itself by traversing edges in their specified direction. However, a domain graph may be undirected cyclic; that is, it may be possible to follow a path of edges from a node back to itself if edges are traversed in any direction. The domain graph with nodes labelled with ≤ and INT, for example, has an undirected cycle.

2.6. An Ontology of Sets

The declared sets in a set schema are either primitive or nonprimitive, and either base or defined. There are therefore four possible classes of sets. These are illustrated in the figure 2.3.

FIGURE 2.3



D is primitive base and not a value set, STR and INT are primitive defined and are value sets, DN is nonprimitive base and not a value set, and VN, $\leq$, and L are nonprimitive defined and value sets. The diagram indicates that all primitive defined sets are value sets; this is true of the sets declared so far since STR and INT are the only primitive defined sets that are value sets. The diagram illustrates the simplest form of the model that will be used throughout this paper. That form of the model is adequate for declaring the set schemas of ordinary enterprises, but the more general form of the model in which not all primitive defined sets are value sets is needed to declare the kernel schema described in [Morr] and discussed in [Gil87].

The primitive defined sets that are value sets are provided by the system with identifiers. For example, the members of INT can be regarded as strings that identify themselves since they are strings that can be read and written by both humans and machines. To refer to a member of INT, it is only necessary to state it as a string. The members of STR are also such strings; to refer to a member of STR it is only necessary to enclose it in quotes. A primitive base set, on the other hand, must have an identifier declared for it that provides a one-to-one association between its members and a subset of a value set. The set DN is an identifier for the primitive base set D. Nonprimitive sets inherit identifiers from their immediate domain predecessors. For example, if dep is the internal surrogate of the department with name 'ENGLISH', then the pair <dep, ENGLISH> is a member of DN and is identified by the pair <ENGLISH, 'ENGLISH'>, since the first element of the pair identifies dep, and the second element the member of VN that is the name of dep.

2.7. Employees and Courses

The next set to be declared for the SU schema is

E for {E ‖ the set of current employees }.

Employees are identified by employee numbers.

VE# for {x:INT | $1000 \le x \le 9999$ | }

E# for{ E, VE# | <1,1>, <0,1> | each employee has a unique employee number }

Each employee is assigned to a single department. The association between employees and departments is declared next:

ED for{E, D | <1,1>, <1,*> | associates each employee with a unique dept }

The second pair of degrees in this case indicates that a department must have at least one member and that it can have any number of members.

A course is identified by a department responsible for it and a course number. Course numbers are selected from a value set:

VC# for{x:INT | $100 \le x \le 699$ | }.

A course can therefore be regarded as an association between departments and course numbers:

C for{ D, VC# | <0*>, <0*> | a course is identified by a responsible dept and a course # }.

The lower degrees in this case mean that not every department is responsible for courses, and not every member of VC# is a course number. The upper degrees mean that a department can be responsible for any number of courses and that a course number may be the number for any number of courses.

An alternative to the given declaration of C would be to declare it as a primitive base set and then to declare an association IDC that identifies members of C through members of DxVC#. From the IDC association there then could be defined the associations between C and D and between C and VC#. The declaration chosen for C avoids the need for additional declarations, although it does so at the price of some artificiality.

If it is desired that course numbers carry additional meaning, such as the year in which a student is expected to take the course, then that meaning should be expressed as part of the intension of the C association. Such a restraint can of course only be enforced by those who assign numbers to courses.

2.8. Defined Sets and Define Predecessors

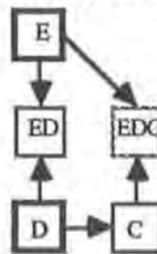
The next set to be declared requires some explanation. An academic department is one responsible for courses; an instructor is a member of an academic department who is competent to teach some of the courses of his/her department. Let ICC be the association between an instructor and a course the instructor is competent to teach. ICC is clearly a nonprimitive base set and a subset of ExC. However, ICC cannot be just any subset of ExC, for an instructor is restricted to being competent to teach courses only of his/her department. If EDC is the association between an employee and all the courses for which the employee's department is responsible, then clearly ICC has EDC as its domain. EDC can be declared as a nonprimitive defined set; it is the first example of such a set that is not a value set:

EDC for{x:E, <y,z>:C | <x,y>:ED | associates the courses for which an academic department is responsible with a member of the department}.

The declaration of a defined set such as EDC has two formal parts and one optional informal part. The optional informal part is a comment on the intension of the set. The first of the formal parts consists of the **domain assertions**, one or more elementary assertions such as the assertions x:E and <y,z>:C for EDC. These assertions accomplish two purposes. First they declare the domain of the defined set, so their order is significant; for EDC the domain is ExC. Second they declare the range of the variables or tuples or nested tuples of variables that appear in them. For EDC the range of the variable x is declared to be the set E, and the range of the pair of variables <y,z> is the set C. Although the latter declaration has the effect of declaring y to be restricted to D and z restricted to VC#, it is of course not equivalent to the domain assertions y:D and z:VC#. To ensure a proper declaration of the domain to which it is to be bound, a variable can have at most a single occurrence among the domain assertions of the declaration.

The second of the formal parts of the declaration of a defined set is an assertion of DEFINE that expresses the set's intension. For EDC the intension is the assertion <x,y>:ED. The domain assertions together with the intension of the declaration determines the extension of the set. The pairs <x,<y,z>> that are members of EDC are those for which x is a member of E, <y,z> is a member of C, and <x,y> is a member of ED. They are therefore those pairs <x,<y,z>> for which the employee x is assigned to the department y responsible for the course <y,z>. The domain graph illustrated in the figure 2.4 shows how EDC is related to E, D, and ED.

FIGURE 2.4



Notice that no degrees are declared for the defined set EDC. Since the membership of EDC is determined by the system from its intension, the degrees of EDC on E and C must follow from that intension, that is, they can be calculated. Although the calculation of the degrees of defined sets is in general an unsolvable problem, they can be quite simply determined for the defined sets usually declared for the set schema of an enterprise. The degrees for EDC, along with the degrees of other defined sets for the set schema of Simple University, are calculated in section 3.1.

The set ICC can now be properly declared:

ICC for {EDC | <0,*>, <1,*> | some employees of an academic department are competent to teach some of the courses of the department }.

The meaning of the degrees of ICC will be described in 2.10, after some required

definitions are given here and in 2.9.

The defined set EDC, like the previously declared defined sets VN and VE#, is of little direct interest. Its primary purpose is to provide a domain for ICC. Consequently a management system is unlikely to keep a list of internal surrogates of members of EDC, but rather use the definition of EDC to check that proposed members of ICC are members of EDC. EDC is therefore called a **virtual** defined set, as opposed to an **actual** defined set for which the management system is instructed to maintain a list of internal surrogates that are members. In an implementation of the SET model it is expected that a user could declare whether a defined set should be virtual or actual.

The sets E and C are immediate domain predecessors of EDC. The set ED used in the intension of EDC is an **immediate define predecessor** of EDC. Every defined set, apart from the system defined sets appearing in the kernel schema, has one or more immediate define predecessors. An **immediate predecessor** of a set is either an immediate domain predecessor or an immediate define predecessor. The immediate domain predecessors are the only immediate predecessors of a nonprimitive base set, while a nonprimitive defined set such as EDC has immediate predecessors that are immediate domain predecessors and ones that are immediate define predecessors. A **predecessor graph** for a set schema is obtained by adding edges to the domain graph corresponding to the immediate define predecessors of defined sets. Like a domain graph, a predecessor graph must be acyclic because sets must be declared before they can be used in the intension of a declared set. It is sometimes useful to display a predecessor graph in a diagram as has been done with domain graphs, although that is not done in this paper.

A **predecessor** of a set is defined recursively: It is either an immediate predecessor of the set or an immediate predecessor of a predecessor of the set. That one set is a predecessor for a second means that the extension of the second set can be dependent upon the extension of the first. For example, although ICC is a base set, it has a long list of predecessors, namely EDC, E, C, ED, D, VC#, and INT, that can affect its extension.

2.9. Arity Domains, Arity Predecessors, and Arity

The members of EDC are pairs <x,y>, with x a member of E and y a member of C; it is a **binary** set because it has two immediate domain predecessors. The members of ICC are also pairs although ICC has only the single immediate domain predecessor EDC; but that single predecessor is binary, so ICC is binary also. EDC is the **arity domain** of ICC, defined recursively as follows: The arity domain of a primitive set, or of a set with two or more immediate domain predecessors, is the set itself. The arity domain of a set with a single immediate domain predecessor is the arity domain of that predecessor. Since EDC has two immediate domain predecessors, it is its own arity domain. Since EDC is the only immediate domain predecessor of ICC, the arity domain of EDC is the arity domain of ICC.

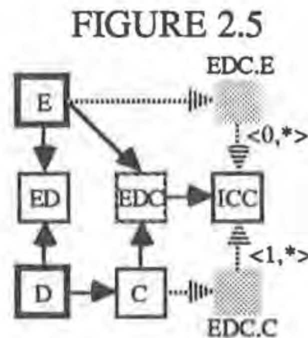
An **arity predecessor** of a nonprimitive set is any immediate domain predecessor

of the arity domain of the set. E and C are the two arity predecessors of ICC. The **multiplicity** of an arity predecessor is the multiplicity of it for the arity domain. Each of E and C have multiplicity 1 since they occur only once in the cartesian product that is the domain of EDC. The **arity** of a set is 1 if its arity domain is primitive, and is otherwise the sum of the multiplicities of its arity predecessors. The arity of ICC is the sum of the multiplicities of E and C, which is 2. An arity 2 set is said to be binary, and an arity 3 set ternary.

2.10. The Degrees of Base Sets that are not Arity Domains

Consider now the degrees declared for ICC. Since ICC is not its own arity domain, these degrees are declared relative to the projection of EDC, its immediate domain predecessor, on the immediate domain predecessors E and C of EDC. This is illustrated in figure 2.5.

FIGURE 2.5



The projections EDC.E and EDC.C are not declared, although they could be if desired. For example, EDC.C could be declared

EDC.C for { y:C | [**For some** x:D] <x,y>:EDC | the projection of EDC on C }.
EDC.E is the set of employees of academic departments and is a proper subset of E, while EDC.C is the set of courses for which departments are responsible and has therefore the same extension as C.

The graph illustrated in figure 2.5 is not a domain graph because the sets EDC.E and EDC.C have not been declared. It is called an **augmented domain graph**.

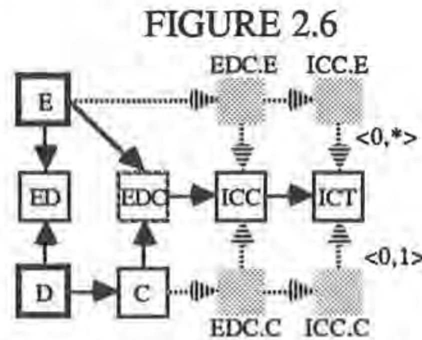
The degrees <0,*> of ICC on EDC.E mean that not all employees of academic departments are competent to teach courses, and that an employee of such a department may be competent to teach any number of courses. The degrees <1,*> on EDC.C mean that for every course there is an employee of an academic department competent to teach it, and that there may be any number of such employees.

The next set to be declared is ICT:

ICT for {ICC | <0,*>, <0,1> | associates an instructor with the courses he/she is currently teaching }
ICT also has EDC as its arity domain, since its immediate domain predecessor ICC has EDC as its arity domain. The degrees of ICT are therefore declared relative to the projections ICC.E and ICC.C as shown in the augmented domain graph

illustrated in the next figure.

FIGURE 2.6



ICC.E is the set of instructors since each member of the set is competent to teach at least one course. ICC.C has again the same extension as C since for every course there is an instructor competent to teach it.

2.11. Managers of Departments

There remains now to declare the sets needed to express the manages associations between departments and employees. Because the manager of an academic department must be an instructor of the department, and the manager of a nonacademic department must be a member of the department, it is necessary to again declare some defined sets before declaring some base sets with the defined sets as domains:

IAD for {<x,y>:ED | **[For some** v:VC#] (<y,v>:C **and** <x,<y,v>>:ICC) | associates an instructor with his/her academic dept }

MA for {IAD | <0,1>, <1,1> | an instructor of an academic dept manages it }

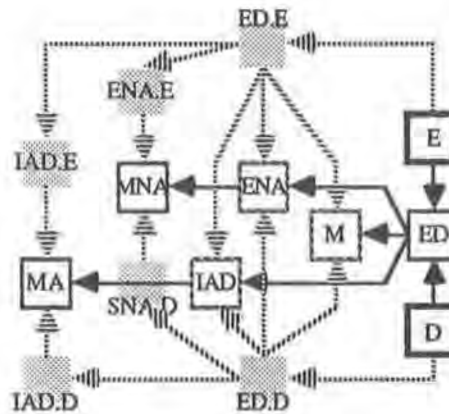
ENA for {<x,y>:ED | **not [For some** v:VC#] <y,v>:C | associates employees of nonacademic depts with their depts }

MNA for{ENA | <0,1>, <1,1> | an employee of a nonacademic department manages it }

M for { z:ED | z:MA **or** z:MNA | associates manager of a dept with the dept }

The augmented domain graph of the next figure illustrates the relationships between these declared sets and the projections that are not declared.

FIGURE 2.7



The immediate predecessors of M are the sets ED, MA, and MNA, but among its predecessors is the set ICC. This means that changes in the extension of the base set ICC could result in changes to the extension of M.

2.12. A Set Schema for Simple University

A summary of the declarations of all the sets of the schema for Simple University is provided in the next figure. As noted before, a more realistic example would have attributes declared for some of the declared sets, but they are unnecessary for the purposes of this paper.

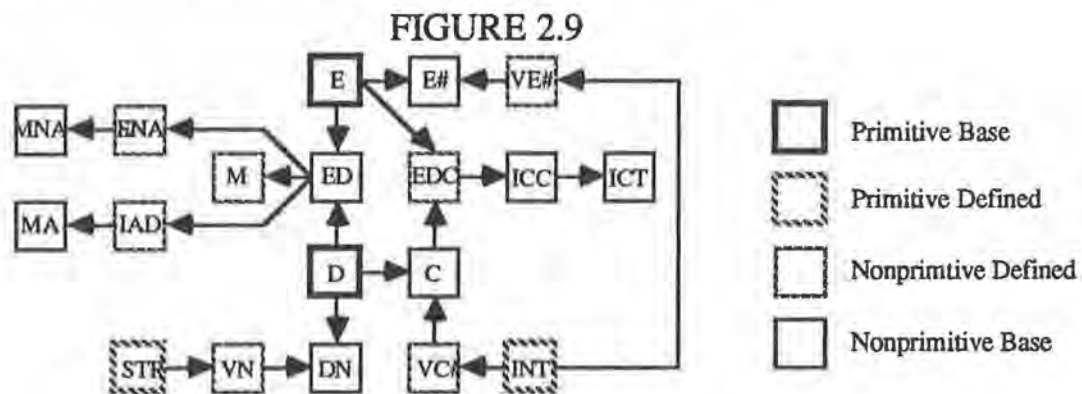
FIGURE 2.8

<table>
<tr><th>NAME</th><th>DOMAIN</th><th>INTENSION / COMMENT</th></tr>
<tr><td>STR</td><td>x:STR</td><td>x system defined | the integers</td></tr>
<tr><td>INT</td><td>x:INT</td><td>x system defined | the strings of characters</td></tr>
<tr><td>≤</td><td>x:INT, y:INT</td><td>x system defined | x is less than or equal to y</td></tr>
<tr><td>L</td><td>x:STR, y:INT</td><td><x,y> system defined | y is the length of x</td></tr>
<tr><td>D</td><td>D</td><td>| a current department</td></tr>
<tr><td>VN</td><td>v:STR</td><td>{x:LNG:} ≤ 10 | a value set for names of departments</td></tr>
<tr><td>N</td><td>D, VN</td><td><1,1>, <0,1> | identifies a dept by a name</td></tr>
<tr><td>E</td><td>E</td><td>| current employees</td></tr>
<tr><td>VE#</td><td>u:INT</td><td>1000 ≤ u ≤ 9999 | a value set for employee numbers</td></tr>
<tr><td>E#</td><td>E, VE#</td><td><1,1>, <0,1> | identifies an employee by a number</td></tr>
<tr><td>ED</td><td>E, D</td><td><1,1>, <1,*> | associates emp with a single dept</td></tr>
<tr><td>VC#</td><td>w:INT</td><td>100 ≤ w ≤ 699 | a value set for course numbers</td></tr>
<tr><td>C</td><td>D, VC#</td><td><0,*>, <0,*> | identifies course by dept and course #</td></tr>
<tr><td>EDC</td><td>x:E, <y,z>:C</td><td><x,y>:ED | an employee's dept's courses</td></tr>
<tr><td>ICC</td><td>EDC</td><td><0,*>, <1,*> | instructor competent to teach courses</td></tr>
<tr><td>ICT</td><td>ICC</td><td><0,*>, <0,1> | instructor currently teaching courses</td></tr>
<tr><td>IAD</td><td><x,y>:ED</td><td>[For some v:VC#] (<y,v>:C and <x,<y,v>>:ICC) |</td></tr>
</table>

		associates an instructor with his/her academic dept
MA	IAD	<0,1>, <1,1> | an instructor of an academic dept manages it
ENA	<x,y>:ED	**not [For some** v:VC#] <y,v>:C | associates emp of nonacademic depts with their depts
MNA	ENA	<0,1>, <1,1> | an emp of a nonacademic department manages it
M	z:ED	z:MA **or** z:MNA | associates manager of a dept with the dept

The first four declarations in this schema are part of the kernel schema. A domain graph for the schema of SU is illustrated in the next figure.

FIGURE 2.9



2.13 Principles of Set Modelling

The existence of a precise model to support the earliest stage of conceptual modelling permits the statement of principles that can be used to guide that modelling. Principles of design that have been followed in the modelling of Simple University are:

+ A primitive base set should have as members only those entities of interest to the enterprise, but should be as large as possible consistent with the provision of a simple identifier.
+ A set that can be declared to be defined should not be declared to be base.
+ Machine maintainable constraints on the membership of a base set should be expressed in the intension of a defined set that is the domain of the base set.

The primitive base set E has been declared so as to include all employees of SU, and D has been declared so as to include all departments. There are subsets of each of these sets that are important to SU, for example the instructors which are a subset of E, and the academic departments which are a subset of D. But these subsets should not be declared as primitive base since the larger sets can be declared just as easily. However, a primitive base set with extension employees and departments has

not been declared because a natural identifier for such a set would be difficult to provide.

Although the set of academic departments has not had to be explicitly declared, it could be declared as follows:

AD for { x:D | [**For some** y:VC#] <x,y>:C | the academic departments }.
It is clearly a defined set, not a base set, so that its extension can be maintained by the system. To declare AD as a base set BAD would be a serious mistake. For the extension of BAD would then have to be maintained by humans; in order to maintain the integrity of the model of SU it would be necessary for them to ensure that the extension of BAD was at all times the same as the extension of AD, an unnecessary task that can be eliminated by declaring AD rather than BAD. In [TYF86] a weak form of defined set is called a redundant relationship and the principle is recognized that redundant relationships should be eliminated.

Instructors of SU are never declared to be competent in courses of departments other than their own. The base set ICC could have been declared to have ExC as its domain, and the constraint could be maintained by users of the system. But it is better to have the system maintain the constraint to ensure that it is not violated. To accomplish this EDC was declared as a defined set and ICC declared as a base set with EDC as its domain.

Another principle that has been followed in the modelling of SU is the following:

+ A set that can be declared to be nonprimitive should not be declared to be primitive.

The set C has been declared as nonprimitive, although it could be declared as primitive. Although some artificiality results from this declaration, it is compensated for by the resulting simplification in the set schema. Nevertheless this principle is one that should be followed with care. Carried to the extreme, the principle could result in the set D being declared as a base set with domain VN, and E being declared as a base set with domain VE#.

Another principle expressed in [Gil86b], but not relevant for the SU example, concerns the declaration of sets of arity greater than two. No such base set should be declared with upper degree 1 on any of its immediate predecessors, since such a degree indicates that the set can be naturally defined in terms of sets of lesser arity. [TYF86] also recognizes the need for such a principle.

3. The Domain Graph Method of Table Design

Tables are commonly used presentation data structures. Given a set schema, it is often desirable to declare a table schema capable of correctly recording the extensions of declared sets of the set schema. In this section a method for designing such a table schema is described. The resulting table schema is a user view of the set schema in the sense that each table in the schema is declared as a defined set and becomes an additional declared set of the set schema. The method is best described in terms of operations on the augmented domain graph of the set schema. For this reason it is called the domain graph method of table design.

A summary of the steps in the method are:

1. Each edge of the augmented domain graph of the set schema is labelled with the lower and upper degrees that have been declared or calculated for it. The calculation of the degrees for defined sets requires human intervention.
2. Subgraphs of the augmented domain graph are determined by selecting edges that have been labelled with a lower degree 1, or with the degrees <1,1>. Which subgraphs to select requires design decisions as to which sets should have their extensions recorded, and what kinds of tables are acceptable. However, the latter decision can be automated if one kind of table is always acceptable. The resulting subgraphs are simplified by eliminating all nodes labelled with undeclared sets, and by replacing directed paths through such nodes with a single edge connecting nodes labelled with declared sets.
3. Each undirected cycle of a subgraph determined in 2 is broken by removing an edge of it with tail a bottom node of the cycle. The result of this step is a forest of trees.
4. Each tree obtained in 3 is extended with new nodes and edges to form its identifier extension. In the tree that is the identifer extension of a given tree, every set labelling a node in the tree has an identifier labelling a node of the tree.
5. From the identifier extension of each tree obtained in 4, a declaration of a table as a defined set is constructed.

Each set, that was selected in 2 to have its extension recorded, will label a node of exactly one of the subtrees obtained in that step. Each subtree selected will result in a single table of the table schema obtained in 5, so that the number of subtrees selected is the number of tables that will appear in the table schema. That number is the minimum possible consistent with the decision made in 2 to keep only edges of lower degree 1, or to keep only edges of degrees <1,1>, if the subgraphs selected in 2 are maximal.

The subsections 3.1 through 3.5 are devoted to these five steps. In 3.6 a justification for the method is stated and proved. It is shown that the extension of each set labelling a node of a subtree obtained in step 2 is correctly recorded through its identifers in the table constructed for the subtree. Finally in 3.7 motivations for the design decision of step 2 are discussed.

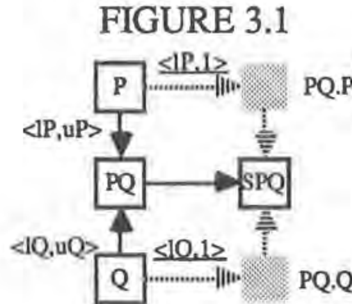
3.1. Degrees for all the Edges of an Augmented Domain Graph

Consider the edges of an augmented domain graph. Each edge with head a node labelled with a base set, has a tail that is a node labelled with an immediate domain predecessor of the set. Each such edge can therefore be assumed to have been labelled with a pair of degrees, since the degrees for a base set of arity one are always assumed to be <0,1>, while those of base sets of arity greater than one are

always declared. Edges of the augmented domain graph remaining to be labelled are therefore of two kinds, those that are directed to nodes labelled with undeclared but implicitly used sets, and those that are directed to nodes labelled with defined sets.

Consider first those edges that are directed to nodes labelled with undeclared but implicitly used sets. In figure 3.1 a typical case for such sets is illustrated.

FIGURE 3.1

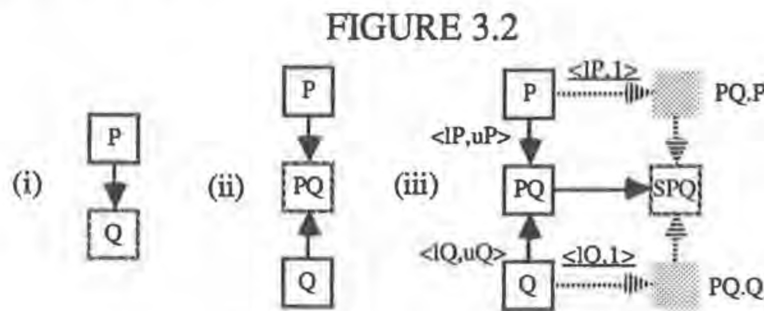


The sets P and Q may have been declared as either base or defined, or they may themselves be undeclared sets. In either case the sets PQ and SPQ will have been declared as either base or defined, with PQ the domain of SPQ. For example, in figure 2.6, P, Q, PQ, and SPQ could be respectively E, C, EDC, and ICC, or they could be respectively EDC.E, EDC.C, ICC, and ICT. In figure 2.7, they could be respectively E, D, ED and any one of IAD, ENA, or M. The degrees of PQ on P have been declared or calculated to be <lP,uP>, and those on Q to be <lQ,uQ>.

Degrees for the edge from P to the undeclared set PQ.P, and the edge from Q to the undeclared set PQ.Q, must be calculated. The upper degrees are both necessarily 1 since PQ.P is a subset of P, and PQ.Q a subset of Q. The lower degree for the edge from P to PQ.Q is necessarily lP, the lower degree of PQ on P, and the lower degree for the edge from Q to PQ.Q is necessarily lQ. In the diagram the calculated degrees have been underlined.

Consider next those edges that are directed to nodes labelled with defined sets. In figure 3.2 three typical cases are illustrated.

FIGURE 3.2



In case (i) Q is defined with domain P, in case (ii) PQ is defined with domain PxQ, and in case (iii) SPQ is defined with domain PQ. The illustration in case (iii) is a

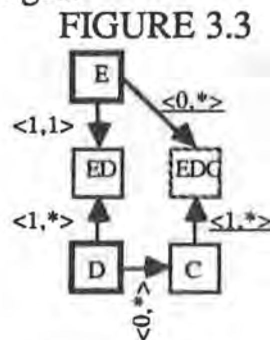
repetition of figure 3.1. An example of case (i) appears in figure 2.2 where P is STR and Q is VN. An example of case (ii) appears in figure 2.4 where P, Q, and PQ are respectively E, C, and EDC. Three examples of case (iii) appear in figure 2.7; in each case P, Q, and PQ are respectively E, D, and ED, while SPQ is IAD, ENA, or M.

It is not possible in general to calculate the degrees of a defined set, since given a sufficiently rich language for stating the intensions of defined sets, it is possible to have the extension of a defined set express solutions to unsolvable problems, such as the halting problem for Turing machines. However, for the defined sets usually declared for a set schema of an enterprise, the degrees can be calculated. This is the case for the degrees of all the defined sets in the set schema for Simple University.

VN, VE#, and VC# are the only defined sets of the set schema that fall under case (i). The degrees for each of these on their domains are clearly <0,1> since each is a proper subset of its domain. Indeed, this will in general always be the case for (i), since there is no need to declare Q if it always has the same extension as P.

EDC is the only example to fall under case (ii). To see how the degrees of EDC are calculated consider figure 3.3, in which is illustrated the domain graph of figure 2.4 with degrees labelling its edges.

FIGURE 3.3



The calculated degrees for EDC are underlined, while those that have been declared are not. Consider the undirected path from E to C via ED and D. There are two edges on this path directed in the direction of the path, the edge from E to ED, and the edge from D to C. The product of the lower degrees 1 and 0 respectively of these two edges is 0. Therefore the lower degree of EDC on E is 0. The product of the upper degrees 1 and * is *. Therefore the upper degree of EDC on C is *. Now consider the path from C to E. Since each member of C is a pair with first element a member of D, for every C there is a member of D. Further, the lower degree of the edge from D to ED is 1. Therefore the lower degree of EDC on C is 1. Every course has a single responsible department, but the upper degree of the edge from D to ED is *. Therefore the upper degree of EDC on E is *.

Consider now the case (iii) for the defined sets IAD, ENA, and M; it is helpful to refer to figure 2.7. Their degrees on the undeclared sets ED.E and ED.D can be calculated. First note that ED.E has the same extension as E and that ED.D has the same extension as D, since the degrees of ED are <1,1> on E and <1,*> on D. Therefore the degrees of IAD and ENA on ED.E are <0,1> and <0,*> on ED.D, since not every employee is an instructor and not every department is academic.

The degrees of M are more difficult to calculate. First note that the union of IAD and ENA is a subset of ED, and that M is a subset of this union. Therefore the degrees of M on ED.E must be <0,1>. However, since the degrees of MA on IAD.D and of MNA on ENA.D are <1,1>, and a department is either academic or not, the degrees of M on ED.D are <1,1> also.

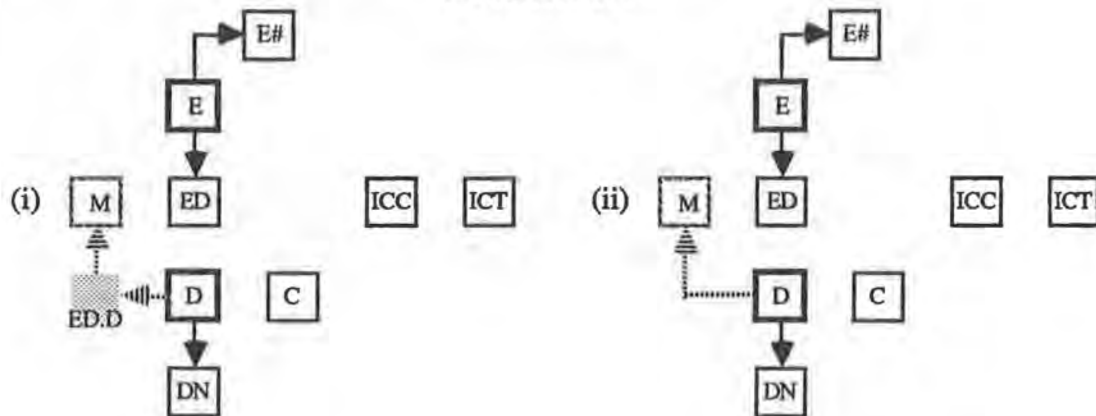
It is only in the calculation of degrees for defined sets that any significant human intervention is needed in the domain graph method of table design. The development of an algorithm for calculating the degrees of most of the defined sets declared in a typical set schema remains a research challenge.

3.2. <1,1>- and 1-subgraphs of an Augmented Domain Graph

It is now assumed that each edge of an augmented domain graph is labelled with a pair of degrees. A **<1,1>-subgraph** of an augmented domain graph is a connected subgraph with edges just those that are labelled with the degrees <1,1>. A **1-subgraph** is a connected subgraph with edges just those that are labelled with the lower degree 1. A <1,1>- or 1-subgraph is **maximal** if it cannot be enlarged by the addition of nodes that are connected by edges labelled with <1,1>, respectively the lower degree 1. It is elementary that the maximal 1-subgraphs of a domain graph provide a unique partition of its nodes, as does also the maximal <1,1>-subgraphs. Further, the nodes of each maximal 1-subgraph are partitioned by the maximal <1,1>-subgraphs. Only maximal 1- and <1,1>-subgraphs need be used in table design, although nonmaximal ones may be used.

In figure 3.4 (i) some of the <1,1>-subgraphs of the augmented domain graph for the set schema for Simple University are illustrated.

FIGURE 3.4



Examples of 1-subgraphs are obtained if the missing edge from D to ED, which is labelled with the degrees <1,*>, is replaced, or the node labelled with EDC.C is restored with its edges from C and to ICC.

The choice of whether tables should be constructed from the 1-subgraphs or the <1,1>-subgraphs is the second human intervention needed in the domain graph method of table design. Motivations for this choice will be discussed in section 3.7. In the meantime, the method will be illustrated using <1,1>-subgraphs.

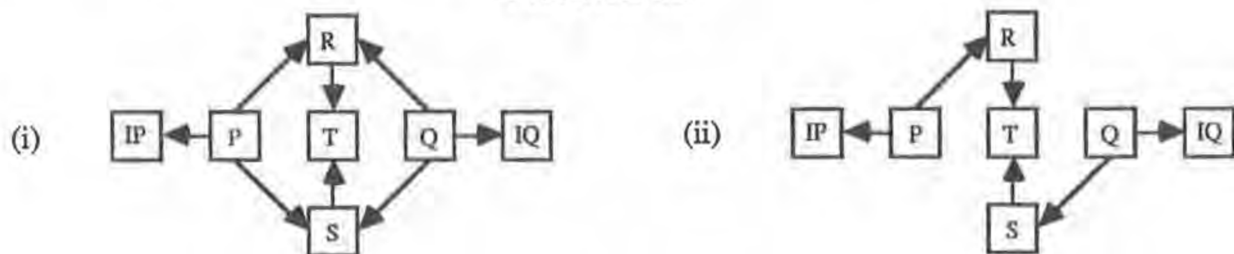
Not all <1,1>-subgraphs of the augmented domain graph are illustrated in figure 3.4 (i). The only subgraphs illustrated are those that contain a node labelled with a set whose membership is to be recorded. The membership of value sets need not be recorded, nor is it necessary to record the membership of defined sets such as EDC, ENA, and IAD, since they were needed only as domains for base sets, so the subgraphs containing nodes labelled with these sets have been dropped. Also the nodes labelled MA and MNA have been dropped because the membership of these sets can be obtained from M. They could, of course, be retained if it is desirable to have all base sets represented as tables.

In figure 3.4 (ii) one of the subgraphs of (i) has been simplified by eliminating the node labelled with the undeclared set ED.D, and replacing the directed path via the node with a single edge, as called for in step (2) of the method. The result is a graph that is no longer a subgraph of the augmented domain graph. Since each edge of the directed path from the node labelled D to the node labelled M has lower degrees <1,1>, the new edge introduced has those degrees also. It is these simplified subgraphs that will be used to determine tables.

3.3. Breaking Undirected Cycles in Subgraphs

All of the subgraphs of figure 3.4 (ii) are undirected acyclic; that is, they are trees. If one was not a tree, then it would be necessary to remove edges to make it so, as illustrated in the next figure.

FIGURE 3.5



Although the cycles of (i) could be broken by dropping any two edges that leaves the graph connected, the edges chosen to form (ii) have tails that are the bottom nodes labelled P and Q; that is, they are nodes of the cycle that are not the head of an edge of the cycle. To select other edges to break cycles results in a table with more columns than are necessary. However, any pair of edges that break the cycles, and that have tails the bottom nodes labelled P and Q, can be selected.

The trees obtained from step 2, and where necessary step 3, will be called henceforth the **selected trees**.

3.4. The Identifier Extension of a Selected Tree

A member of a value set can always be recorded in a table since it can be read by both humans and machines. A member of other sets can only be indirectly recorded in a table by recording an identifier for it. For example, a table of members of E consists of all the employee numbers that had been assigned to members of E. Similarly, a table of the names of departments in D is used to record the members of D. A table of the members of a nonprimitive set consists of the tuples of identifiers of the tuples that are members of the set; for example, a table for ED consists of the pairs <e#, dn> for which the employee with employee number e# has been ED associated with the department with name dn.

To construct a table from a selected tree it is necessary to extend the tree in order to ensure that every set that labels a node of the tree is provided with its identifier. The resulting tree is called the **identifier extension** of the selected tree. An algorithm for constructing the identifier extension for any selected tree will be

described.

Recall from 2.9 that an arity predecessor of a nonprimitive set is any immediate domain predecessor of the arity domain of the set, and that the multiplicity of an arity predecessor is the multiplicity of it for the arity domain. When edges are dropped from the augmented domain graph to form a forest of trees, a nonprimitive set labelling a node of the graph can become disconnected from one or more of its arity predecessors. For example, in figure 3.4 (ii), each of the sets E#, ED, DN, and C is its own arity domain and was disconnected from one or more of its arity predecessors, which are in these cases immediate domain predecessors: E# was disconnected from VE#, ED from D, DN from VN, and C from both D and VC#. In addition each of the sets M, ICC, and ICT, with arity domains ED, EDC, and EDC, respectively, was disconnected from one or more of its arity predecessors: M was disconnected from E, and both ICC and ICT were disconnected from both E and C.

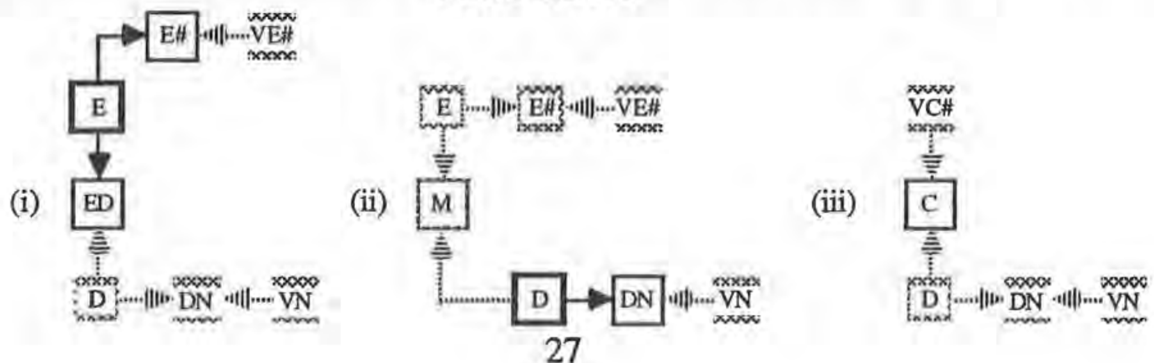
In order to ensure that every member of every set labelling a node of a selected tree can be given an identifier, it is necessary to first extend the tree to one in which each node nd is **arity predecessor complete**; that is to say, nd satisfies the following condition: Let S be a set of arity m labelling nd, and let $S_1, \ldots, S_m$ be the arity predecessors of S, with repititions appropriate for the multiplicities. Then there are nodes $nd_1, \ldots, nd_m$, labelled respectively with $S_1, \ldots, S_m$, and such that for each nd_i, <nd_i, nd> is an edge of the tree.

By beginning with a selected tree, and repeatedly adding as needed new nodes nd' and edges <nd', nd>, with nd' labelled with an arity predecessor of the set labelling nd, an extension of the tree can be obtained that is arity predecessor complete for each of its nodes.

Each bottom node nd of the resulting tree will be labelled with a primitive base set, or a value set. Nothing more need be done for a value set since it is its own identifier. For a primitive base set S, an identifier must be provided. Let S be a primitive base set labelling nd, and let IS be its identifier with value set VIS. Nodes nd' and nd" labelled with IS and VIS respectively are added to the tree together with the edges <nd, nd'> and <nd", nd'>, if such nodes do not already exist.

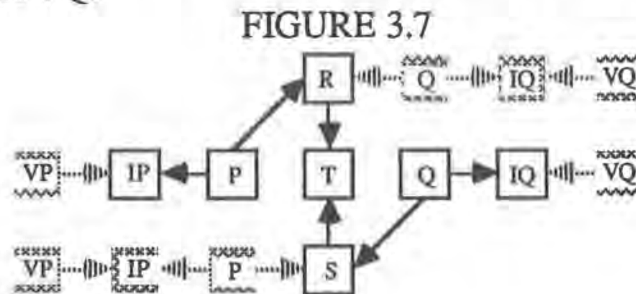
The identifier extensions of three of the selected trees illustrated in figure 3.4 (ii) are illustrated in figure 3.6. The nodes that have been added to the selected trees to form their identifier extensions are all represented by cross-hatched boxes.

FIGURE 3.6



The identifier extension of the tree (ii) of figure 3.5 is illustrated in figure 3.7. It is assumed that IP is an identifier for P with value set VP, and that IQ is an identifier for Q with value set VQ.

FIGURE 3.7



3.5. The Table for a Selected Tree

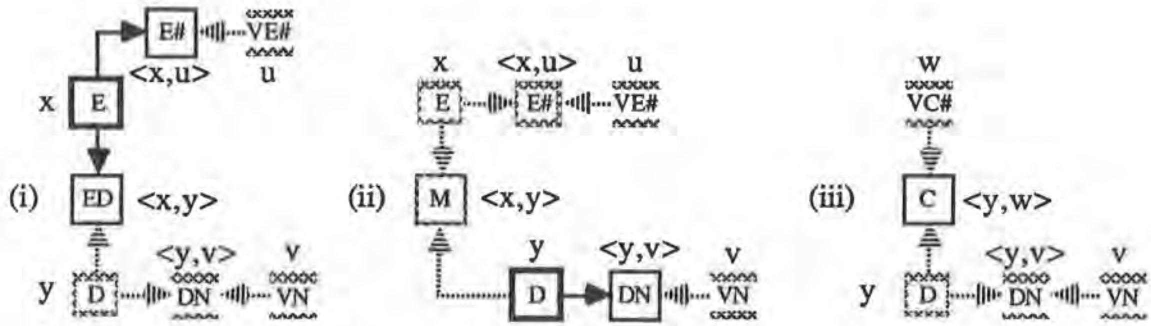
The declaration of a defined set, and therefore of a table for a selected tree, has the form of the declaration of EDC in section 2.8. The two formal parts needed for the declaration are the intension of the set expressed as an assertion of DEFINE, and the domain assertions that simultaneously declare the domain and the ranges of the variables appearing in the assertions.

The intension of the defined set that is the table for a given selected tree is obtained in two steps. A conjunction of elementary assertions, called a **join assertion** of the tree, is determined from the identifier extension of the tree. To form a join assertion, distinct variables must be assigned to each of the bottom nodes of the identifier extension, and then these variables, or nested tuples of them, are assigned to the other nodes in a manner described below. Each elementary assertion of the join assertion is then of the form tup:S, where S is a nonprimitive set labelling a node of the identifier extension that is not a bottom node, and tup has been assigned to the node. The intension of the table for the given selected tree is formed from a join assertion by prefixing it with an existential quantifier for each variable assigned to a node that is labelled with a primitive base set.

The domain assertions all take the form var:VL, where VL is a value set labelling a bottom node of the tree, and var is the variable assigned to the bottom node. The order of the domain assertions determines the order of the columns appearing in the table and is unrestricted. A convenient order is to have the identifier(s) for a set labelling a node preceed attributes of the set and associations with other sets. Since each such set labels a node of exactly one selected tree, the order can be determined automatically.

The process of assigning variables and nested tuples of variables to the nodes of the identifier extension of a selected tree is a simple one that can best be described by examples. In figure 3.8, variables and tuples have been assigned to the nodes of the trees of figure 3.6.

FIGURE 3.8



Consider, for example, the tree (ii). The nodes labelled E, VE#, D, and VN, are the bottom nodes of the tree. They have been assigned the distinct variables x, u, y, and v. The node labelled E# is assigned <x,u> because the node labelled E has been assigned x and the node labelled VE# has been assigned u, and the domain of E# is ExVE#. Similarly for the node labelled DN. The node labelled M is assigned <x,y> because the node labelled E is assigned x, the node labelled D is assigned y, and the domain of the arity domain ED of M is ExD.

Variable assignments to the tree illustrated in figure 3.7 can be given in a similar fashion. For example, let the variables ur and us be assigned to the two nodes labelled VP, the variables vr and vs to the two nodes labelled VQ, the variables xr and xs to the two nodes labelled P, and the variables yr and ys to the two nodes labelled Q, all in order from top to bottom in the diagram. Assume that the domain of R and of S is PxQ, and that the domain of T is RxS. Then the tuple assigned to the node labelled T is << xr,yr>,< xs,ys>>.

The tables corresponding to the trees in figure 3.8 are:

TE for { u:VE#, v:VN | [**For some** x:E, y:D] (<x,u>:E# **and** <x,y>:ED **and** <y,v>:DN) | },

TD for { v:VN, u:VE# | [**For some** x:E, y:D] (<x,u>:E# **and** <x,y>:M **and** <y,v>:DN) | }, and

TC for { v:VN, w:VC# | [**For some** y:D] (<w,y>:C **and** <y,v>:D) | }.

In a more realistic example the tables would have additional columns for the values of attributes of declared sets. However, the table TE would not have columns for the attributes of D, since such attributes would never label nodes of the selected subgraph containing the original node labelled with E. Similarly the attributes of D would appear only in TD, and the attributes of C in TC. The tables obtained from the other two selected trees of figure 3.4 are declared:

TICC for { u:VE#, v:VN, w:VC# | [**For some** x:E, y:D] (<x,u>:E# **and** <y,v>:DN **and** <x,<y,w>>:ICC) | }, and

TICT for { u:VE#, v:VN, w:VC# | [**For some** x:E, y:D] (<x,u>:E# **and** <y,v>:DN **and** <x,<y,w>>:ICT) | }.

The table obtained for the tree of figure 3.7 under the variable assignment given earlier is

TPQRST for { ur:VP, vr:VQ, us:VP, vs:VQ | [**For some** xr:P, yr:Q, xs:P, ys:Q] (<xr,ur>:IP **and** <xs,us>:IP **and** <yr,vr>:IQ **and** <ys,vs>:IQ **and** <xr,yr>:R **and** <xs,ys>:S **and** <<xr,yr>,<xs,ys>>:T) | }.

3.6. Justification for the Method

Consider any set schema Sch. Let S be a set declared in Sch of interest to users. S labels exactly one node of the domain graph of Sch, and therefore exactly one node nde of the augmented domain graph. Let Tdg be the single tree obtained in step 3 of 3.1 of which nde is a node. Let Tr be the identifier extension of Tdg obtained in step 4, and let T(Tr) be the table obtained in step 5. If the domain graph method is correct, then it should be possible for a user to determine the membership of S from the table T(Tr).

Consider, for example, the table TE. E and ED are the only two sets of interest to users of TE that label nodes of the selected tree of figure 3.4 (ii) for which the tree (i) of 3.6 is the identifier extension. From knowledge of E# and DN, a user can determine from TE the membership of two related sets EU and EDU declared as follows:

EU for {x:E | [**For some** u:VE#,v:VN] (<u,v>:TE **and** <x,u>:E#) | }, and
EDU for {<x,y>:ExD | [**For some** u:VE#,v:VN] (<u,v>:TE **and** <x,u>:E# **and** <y,v>:DN) | }.

As far as the user is concerned, the table TE is correct for E if EU has the same extension as E, and EDU has the same extension as ED. Similarly TC is correct for C if the set

CU for {<y,w>:DxVC# | [**For some** v:VN] (<v,w>:TC **and** <y,v>:DN) | }

has the same extension as C.

In [Gil87] a definition is given of the user form SU of S, of which EU, EDU, and CU, are special cases, and the following theorem is proved:

Theorem: Assume that all declared and defined degree constraints labelling edges of an augmented domain graph are satisfied by the membership of the declared sets. Let S be a declared set labelling a node of a 1-connected, not necessarily maximal, subtree of the domain graph, and let SU be the user form of S declared for the table obtained from the subtree. Then S and SU have the same extension.

The statement and proof of a simple sufficient condition on the satisfiability of the degree constaints, is also given there.

3.7. 1-Connected or <1, 1>-Connected Selected Trees?

Since tables obtained from 1-connected or <1,1>-connected subgraphs are equally correct, the decision as to which tables to use depends upon other considerations, sometimes upon taste.

When the tables are declared as defined actual, and correspond to flat file data structures that record them, duplication of data is a concern. Using 1-connected subgraphs can result in tables with unnecessary duplication of data, and it is therefore prudent to consider <1,1>-connected subgraphs. Because the <1,1>-connected subgraphs partition the nodes of the 1-connected subgraphs, the tables obtained from the <1,1>-subgraphs split the tables otained from the 1-connected subgraphs. Therefore using <1,1>-connected subgraphs results in the

construction of more than the minimum number of tables, and therefore in the unnecessary duplication of data. A simple calculation will determine which kind of duplication is the least demanding of space.

When the tables are used purely as presentation data structures, the choice is more fully a matter of taste. Taste need not be restricted to the unnecessary, and often artificial first normal form. Tables that are not in that form can be easily defined, and the additional formatting provided for data can make them more comprehensible to users. The restriction to first normal form demanded in the relational model results, after all, from a burdening of a presentation view of data with implementation concerns. Other evidence of that burdening is the necessity to declare keys for tables in addition to the identifiers declared for the primitive base sets upon which the tables are based.

It is interesting to note that the synthetic method of table design described in [Bern76], which also found a minimum number of tables, is dependent upon functional dependencies for attributes that can be defined in terms of the pairs <1,1> of degrees. The domain graph method is also related to the method described in [Knt83a]. Its relationship to the method described in [TYF86] is less clear because of the use of normalization in that method.

4. Integrity Constraints

Only two kinds of integrity constraints are expressible in the SET model, the constraints implicit when one set is declared to be the domain of another, and the degree constraints declared for base sets. As explained in 2.10, the two interact. For example, the effect of the degree constraints of ICT are described in terms of its arity domain EDC and its arity predecessors E and C. In 4.1 the effect of these constraints on the tables declared in 3.5 is described, while in 4.2 the possibility of using the fanset data structure of the network model to maintain the constraints is discussed.

4.1. The Relational Model

The domain graph method of designing tables from a set schema for an enterprise results in a relational schema for the enterprise. Since the tables are defined sets, the membership of them is automatically maintained by a management system supporting the set schema. From the definitions of the tables can be determined constraints that will be automatically maintained by the management system, but that will have to be declared for a relational database.

Consider, for example, the constraints that are determined by the definitions of the tables TD, TE, TC, TICC, and TICT given in 3.5:

1. the VE# values of TD, TICC, and TICT, must appear in TE;
2. the VN values of TE, TC, TICC, and TICT, must appear in TD;
3. the VC# values of TICC and TICT must appear in TC;
4. the <VN, VE#> values of TD must appear reversed in TE;
5. the <VE#, VN> values of TICC must appear in TE;
6. the <VN, VC#> values of TICC must appear in TC;

7. for those VN values appearing in TICC, the <VN, VE#> values of TD must appear in TICC; and
8. the <VE#, VN, VC#> values of TICT must appear in TICC.

The constraints 1-4 can be seen to be determined from the identifier extensions for the trees from which the definitions of the tables were obtained. The identifier extensions from which TE, TD, and TC were obtained are illustrated in figure 3.8. Note that the node labelled D in (i) was added while forming the identifier extension (i) of a selected subtree of the domain graph. Similarly for the nodes labelled E in (ii) and D in (iii). There is only one set E and one set D in the set schema for SU. Consequently any value of VE# appearing in TD must be among the values of VE# appearing in the table TE. Similarly any value of VN appearing in TE or TC must be among the values of VN appearing in the table TD. But also the <VN, VE#> values of TD must appear reversed in TE, since every pair that is a member of ED.D must be a member of ED. The constraints involving TICC and TICT are obtained in a similar fashion.

The given constraints are all examples of what has been called referential integrity constraints [Date83], although they are much more complicated than those discussed in the literature. Nevertheless, were the tables TE, TD, TC, TICC, and TICT, to be declared for a relational database, the management system for the database would have to maintain these constraints.

The fact that the constraints do not have to be explicitly recognized in the set schema for SU, but that they follow implicitly from the declarations of the tables as defined sets, indicates one advantage a conceptually oriented model such as SET has over a presentation oriented model such as the relational. Although the domain and degree constraints of the SET model express simple real world constraints in a simple fashion, the form that these constraints take in the defined tables is much less transparent. In more complex databases, such as those supporting a kernel schema or knowledge bases, the number of such inclusion constraints that have to be maintained overwhelms any benefits that can be expected from the presentation oriented relational database model.

4.2. Fansets and Referential Integrity

In [Date83] the use of the fanset data structure in the maintenance of referential integrity constraints is discussed. With an appropriate pointer implementation, the constraints 1-4 can be maintained using fansets, although the reversal in the fourth adds a complication. The constraints 5 and 6 present much greater difficultes, while 7 is of a kind not customarily implemented using fansets. The difficulties presented by 5 and 6 arise from the fact that the two pairs <VE#, VN> and <VN, VC#> have VN in common; although a single pointer can maintain either one of these constraints, a pair of pointers will not maintain the pair. It is therefore not clear how the tables would be implemented in the Network Model. On the other hand, a fanset implementation of a set schema may be feasible if defined sets can be maintained virtually.

5. Conclusions

The presentation orientation of the relational model provided a more abstract view of data than could be provided by the hierarchical or network models. The model did not, however, completely free a user from implementation concerns. Its emphasis on first normal form and the need for keys are consequences of its treatment of tables as flat file storage structures, regardless of whether they are actually to be so used. More importantly, the concentration of the relational model on data, rather than on the reality the data is intended to describe, has resulted in unnecessarily complicated table design methods and integrity constraints. Through the use of the specification/query language DEFINE of the SET model, the design of presentation data structures such as tables can be almost fully automated, declared as defined sets, and proved to be correct. As a consequence integrity constraints for presentation data structures need not be stated, but are maintained by any system that maintains the domain and degree constraints of the SET model.

BIBLIOGRAPHY

[Bach69] BACHMAN, C.W. Data structure diagrams. *Data Base* 1,2 (Summer 1969), 4-10.

[Bern76] BERNSTEIN, P.A. Synthesizing third normal form relations from functional dependencies. *ACM Trans. Database Syst.*, 1, 4 (March 1976), 271-298.

[Blac85] BLACK, MICHAEL JULIAN. Naive Semantic Networks. Final Paper, Directed Study in Computer Science. Dept of Comp. Sci., Univ. of B.C. Jan 22, 1985.

[BMS84] BRODIE, MICHAEL L., MYLOPOULOS, JOHN, AND SCHMIDT, JOACHIM W. On conceptual modelling, Springer-Verlag, 1984.

[Chen76] CHEN, PETER PIN-SHAN. The Entity-Relationship model - toward a unified view of data. *ACM Trans. Data Base Syst.*, 1, 1 (March 1976), 9-36.

[Chen77] CHEN, PETER PIN-SHAN. The Entity-Relationship model - A basis for the enterprise view of data. AFIPS Conference Proceedings, Vol. 46, 1977 NCC.

[Chen85] CHEN, PETER P.S. Database Design Based on Entity and Relationship. [Yao85], 174-210.

[Codd79] CODD, E.F. Extending the Database Relational Model to Capture More Meaning. *ACM Trans. Database Syst.*, 4, 4 (Dec 1979)

[Date83] DATE, C.J. An Introduction to Database Systems, Vol.II. Addison-Wesley, 1983.

[DJNY83] DAVIS, CARL G., JAJODIA, SUSHIL, NG, PETER ANN-BENG, AND YEH, RAYMOND T. Entity-relationship approach to software engineering, North-Holland, 1983.

[DKM86] DE TROYER, O., KEUSTERMANS, J., AND MEERSMAN, R. How Helpful is an Object-Oriented Database Model?. [DiDa86]. 124-132.

[Ditt86] DITTRICH, KLAUS R. Object-oriented Database Systems: The Notion and the Issues. (extended abstract) 1986 International Workshop on Object-Oriented Database Systems. 2-4.

[DiDa86] DITTRICH, KLAUS, AND DAYAL, UMESHWAR. (Eds) Proc. International Workshop on Object-Oriented Database Systems. ACM and IEEE. Sept 23-26, 1986.

[FuNe86] FURTADO, ANTONIO L. AND NEUHOLD, ERICH J. Formal Techniques for Data Base Design. Springer-Verlag. 1986.

[Gil77] GILMORE, PAUL C. Defining and computing many-valued functions. Parallel Computers - Parallel Mathematics. FEILMEIER, M. (ed.), North-Holland (1977), 18-23.

[Gil86a] GILMORE, PAUL C. Natural deduction based set theories: a new resolution of the old paradoxes. *J. Symb. Logic*, 51, 2 (June 1986), 393-411.

[Gil86b] GILMORE, PAUL C. Class notes for CPSC 404. Dept of Computer Science, Un. of B.C. August 11, 1986.

[Gil87] GILMORE, PAUL C. Justifications and Applications of the SET Conceptual Model.Dept of Computer Science Tech. Report 87-9, Un. of B.C. April 1987.

[HaMc81] HAMMER, MICHAEL, AND McLEOD, DENNIS. Database description with SDM: a semantic database model. *ACM Trans. Database Syst.*, 6, 3 (Sept 1981), 351-386.

[Knt78] KENT, WILLIAM. Data and Reality, North-Holland, 1978.

[Knt81] KENT, WILLIAM. Consequences of Assuming a Universal Relation. *ACM Trans. Database Syst.*, 6, 4 (Dec 1981), 539-556.

[Knt83a] KENT, WILLIAM. Fact-based data analysis and design, [DJNY83], 3-54.

[Knt83b] KENT, WILLIAM. The Universal Relation Revisited. *ACM Trans. Database Syst.*, 8, 4 (Dec 1983), 644-648.

[KhCo86] KHOSHAFIAN, SETRAG N. AND COPELAND, GEORGE P. Object Identity. [Meyr86]. 406-416.

[LeSa83] LENZERINI, MAURIZIO, AND SANTUCCI, G. Semantic integrity and specifications, [DJNY83], 529-550.

[LuKl86] LUK, W.S. AND KLOSTER, STEVE. ELFS: English Language for SQL. *ACM Trans. Database Syst.*, 11, 4 (Dec 1986), 447-472.

[LyKe86] LYNGBACK, PETER, AND KENT, WILLIAM. A Data Modelling Methodology for the Design and Implementation of Information Systems. [DiDa86]. 6-17.

[Morr] MORRISON, RODERICK. Implementating a Set Based Data Model and its Data Definition/Manipulation Language. PhD thesis, Department of Computer Science, Un. British Columbia. *In progress.*

[MyWo80] MYLOUPOLOS, J., AND WONG, H. Some features of the TAXIS

data model. *Proc. 6th Int. Conf. Very Large Databases*, Montreal (1980).

[Rock81] ROCK-EVANS, ROSEMARY. Data analysis. IPC Electrical-Electronic Press Ltd, Sutton, Surrey, England (1981).

[SAAF73] SENKO, M.E., ALTMAN, E.B., ASTRAHAN, M.M., AND FENDER, P.L. Data structures and accessing in data-base systems. *IBM Syst. J.* 12, 1 (1973), 30-93.

[Ship81] SHIPMAN, DAVID W. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst.*, 6, 1 (March 1981), 140-173.

[Sowa84] SOWA, J.F. Conceptual Structures: Information Processing in Mind and Machine. Addison-Wesley, 1984.

[TYF86] TEOREY, TOBY J., YANG, DONGQING, AND FRY, JAMES P. A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model. ACM Computing Surveys, 18, 2 (June 1986). 197-222.

[TaFr76] TAYLOR, ROBERT W.; AND FRANK, RANDALL L. CODASYL Data-Base Management Systems. *ACM Comp. Surveys*. 8,1 (March 1976), 67-103.

[TrLo87] TRYON, D.C.; AND LOYD, D.G. Information Resource Depository: History, Current Issues, and Future Directions. Pacific Bell, A Pacific Telesis Company. Presentation to Canadian Information Processing Society, Vancouver, Canada, February 1987.

[Yao85] YAO, S. BING (editor). Principles of Database Design, Volume I, Logical Organizations. Prentice-Hall, 1985.

JUSTIFICATION
for
DOMAIN GRAPH METHOD OF TABLE DESIGN
A Supplement to Tech Report 87-7

Paul C Gilmore

1. Introduction

In the paper [Gil87b] it was argued that the conceptual orientation of the entity-relationship (ER) model [Chen76,77] permits it to avoid both the excessive implementation concerns of the hierarchical and network models, and the restrictive presentation concerns of the relational model. The weakness of the ER model, on the other hand, is its lack of a sound foundation upon which a management system might be based. The primary motivation for the development of the purely set-based data and conceptual model SET and its specification/query language DEFINE described in [Gil87b] was to provide such a foundation. Five reasons were offered as to why this is necessary:

1. For the unified view of data proposed in [Chen76] to be fully achieved, a database is needed that is capable of recording a high level conceptual model of an enterprise and at the same time of providing the tables for a relational database schema as a defined user view in its specification/query language.
2. The ER modelling process requires a greater discipline than is now possible.
3. A provably sound foundation is needed for databases that can reference and describe themselves.
4. A fully unified model of an enterprise is needed that at the same time can give a conceptual view of the enterprise, a user's view of data as it is presented, a data administrator's view of data as it is stored, and a programmer's view of the processing of the data.
5. Sound foundations are needed for knowledge base systems capable of dealing with incomplete information.

The domain graph method of table design described in [Gil87b] translates the set schema obtained from the modelling of an enterprise using SET, into a table schema in which each table is a defined user view declared as a set in DEFINE. But for one initial step, the method is fully automated. A theorem stated in the paper asserts that the method will always result in correct tables; that is, tables that are free from any anomalies. But no proof was provided for the theorem. The first purpose of this paper is to remedy that deficiency, so that (1) can be offered as an advantage of the SET model. That proof is provided in section 2.

A tentative beginning was made in [Gil87b] in providing a basis for some of the decisions that must be made while modelling, so that (2) can be offered as an advantage of the SET model as well. Another purpose of this paper is to demonstrate that DEFINE can be used as a query language for the SET model, and that the model also satisfies the demands (3)-(5). While doing this, applications and extensions of the SET model will be described in section 3. The final purpose of this paper, accomplished in section 4, is to sketch the basis for the consistency of the model and its integrity constraints.

Familiarity with the paper [Gil87] is presumed.

2. Correctness of the Domain Graph Method of Table Design

The domain graph method of table design described in [Gil87b] translates the set schema obtained from the modelling of an enterprise using SET, into a table schema in which each table is a defined user view declared as a set in DEFINE. The method was described in terms of operations on the augmented domain graph of the set schema. The steps of the method are:

1. Each edge of the augmented domain graph of the set schema is labelled with the lower and upper degrees that have been declared or calculated for it.

2. 1-connected subgraphs of the augmented domain graph are determined by selecting only edges that have been labelled with the lower degree 1. The resulting subgraphs are simplified by eliminating all nodes labelled with undeclared sets, and by replacing directed paths through such nodes with a single edge connecting nodes labelled with declared sets.
3. Each undirected cycle of a subgraph determined in 2 is broken by removing an edge with tail a bottom node of the cycle. The result of this step is a forest of trees.
4. Each tree obtained in 3 is extended with new nodes and edges to form its identifier extension.
5. From the identifier extension of each tree obtained in 4, a declaration of a table as a defined set is constructed.

The construction in (4) of the identifier extension of a tree needed in (3) was described in 3.4 of [Gil87]. The following lemma expresses a fundamental property of identifier extensions:

Lemma 1: Let Tdg be any tree obtained in step 3, and let Tr be its identifier extension obtained in step 4. Let $nd_0, nd_1, \ldots, nd_p$, be an undirected path of Tr for which nd_0 is a node of Tdg. Let the edge from nd_i, to nd_{i+1}, have lower degree 0. Then the edge has head nd_i and tail nd_{i+1}.

Proof of lemma 1: An edge of lower degree 0 is not an edge of Tdg, but has been added in making a node arity predecessor complete or in adding a pair of nodes labelled with an identifier for a primitive base set and with a value set for the identifier. The former must point towards the node that is arity predecessor incomplete without it, while the latter must point towards the node labelled with the identifier.

End of proof of lemma 1

Consider now any set schema Sch. Let S be a set declared in Sch of interest to a user. S labels exactly one node of the domain graph of Sch, and therefore exactly one node nde of the augmented domain graph. Let Tdg be the single tree obtained in step 3 of which nde is a node. Let Tr be the identifier extension of Tdg obtained in step 4, and let T(Tr) be the table obtained in step 5. If the domain graph method is correct, then it should be possible for a user to determine the membership of S from the table T(Tr).

The bottom nodes of Tr are labelled with value sets or primitive base sets only, while all other nodes are labelled with nonprimitive sets. The declaration of T(Tr) makes use of an assignment of variables to the bottom nodes of Tr, with a distinct variable assigned to each node. Every other node nd of Tr is then assigned a nested tuple tp of the variables assigned to the bottom nodes; tp is a tuple of the tuples assigned to the nodes that are immediate predecessors of nd. Associated with each node of Tr is therefore an assertion tp:SS, called the **assertion of the node**, where tp is the variable or tuple assigned to the node, and SS is the set that labels the node. Join(Tr) is an assertion of DEFINE consisting of the conjunction of all such assertions for nodes that are not bottom nodes of Tr. The declaration of T(Tr) is then:

T(Tr) for { $v_1:V_1, \ldots, v_n:V_n$ | [**For some** $bv_1:BS_1, \ldots, bv_m:BS_m$] Join(Tr) | }.

Here $V_1, \ldots, V_n$ are all the value sets that label bottom nodes of Tr in some order with repititions if necessary, and $v_1, \ldots, v_n$ are the variables assigned to those nodes; $BS_1, \ldots, BS_m$ are all the primitive base sets that label bottom nodes of Tr in some order with repititions if necessary, and $bv_1, \ldots, bv_m$ are the variables assigned to those nodes.

Consider now how a user determines the membership of S from T(Tr). First the columns of T(Tr) that identify members of S must be known to the user. These columns are determined as follows: Let tup be the tuple assigned to the node nde that S labels. The variables occurring in tup are among the variables $v_1, \ldots, v_n$ and $bv_1, \ldots, bv_m$, since these are all the variables assigned to bottom nodes of Tr. By reordering the columns of T(Tr) it can be assumed that $v_1, \ldots, v_j$ are the variables among $v_1, \ldots, v_n$ that occur in tup. Without loss of generality it may be assumed that $bv_1, \ldots, bv_b$ are those variables among $bv_1, \ldots, bv_m$ that occur in tup. Join(Tr) necessarily includes a conjunction

$bv_1:IBS_1:vbv_1$ **and** ... **and** $bv_m:IBS_m:vbv_m$,

where $IBS_1, \ldots, IBS_m$ are identifiers for $BS_1, \ldots, BS_m$, and $vbv_1, \ldots, vbv_m$ are among the variables $v_1, \ldots, v_n$. None of the variables $vbv_1, \ldots, vbv_b$ occur among $v_1, \ldots, v_j$ since Tr is a tree. Therefore by a futher reordering of the columns they may be assumed to be $v_{j+1}, \ldots, v_{j+b}$. The columns of T(Tr) that are used to identify members of S are therefore those for the variables $v_1, \ldots, v_j, v_{j+1}, \ldots, v_{j+b}$.

With knowledge of the columns of T(Tr) that identify members of S, a user determines the members of S as follows: From a selected row of T(Tr), a user can determine a member of $V_1 x \ldots x V_{j+b}$. From knowledge of the identifiers $IBS_1, \ldots, IBS_b$ a user can therefore determine a j+b tuple that is a member of $V_1 x \ldots x V_j x BS_1 x \ldots x BS_b$. This j+b tuple is a flattened version of a member of a S, provided that the table T(Tr) is correct. No matter whether T(Tr) is correct or not, it is a flattened version of a member of a set SU that will be declared after one more definition is given to make more precise what is meant by "flattened".

The **bottom domain** of a set labelling a node of Tr is defined recursively as follows: The bottom domain of the value set or primitive base set that labels a bottom node of Tr is the set itself. Let SS label a node nd that is not a bottom node, and let $SS_1 x \ldots x SS_k$ be the domain of the arity domain of SS. Necessarily there are exactly k nodes that are tails of edges with head nd and these nodes are labelled with $SS_1, \ldots, SS_k$. Let $BDSS_1, \ldots, BDSS_k$ be the bottom domains of $SS_1, \ldots, SS_k$, respectively. Then the bottom domain of SS is $BDSS_1 x \ldots x BDSS_k$.

Let BDS be the bottom domain of S. Then the set SU, called the **user form** of S, is declared:
SU for {tup:BDS | [**For some** $v_{j+1}:V_{j+1}, \ldots, v_n:V_n$] ($<v_1, \ldots, v_n>$:T(Tr) **and**
$<bv_1,v_{j+1}>:IBS_1$ **and** ... **and** $<bv_b,v_{j+b}>:IBS_b$) | }.

The following theorem justifies the domain graph method of table design.

Theorem: Assume that all declared and defined degree constraints labelling edges of an augmented domain graph are satisfied by the membership of the declared sets. Let S be a declared set labelling a node of a 1-connected, not necessarily maximal, subtree of the domain graph, and let SU be the user form of S declared for the table obtained from the subtree. Then S and SU have the same extension.

Proof of theorem: Let Sch, S, nde, Tdg, Tr, and, T(Tr), be as described. To avoid clashes of bound variables, the variables $bv_1, \ldots, bv_b$ in the declaration of T(Tr) will be replaced below by the distinct variables $bv'_1, \ldots, bv'_b$ that are distinct from any variables assigned to nodes of Tr. Join(Tr) is the join assertion for Tr for the given variable assignment. Join'(Tr) results from Join(Tr) by replacing occurrences of $bv_1, \ldots, bv_b$ by $bv'_1, \ldots, bv'_b$ respectively.

To prove the theorem it is sufficient to prove that the assertions

1. [**For all** s:SU] s:S, and
2. [**For all** s:S] s:SU,

are assigned **true** whenever all declared and defined degree constraints labelling edges of the augmented domain graph are satisfied by the membership of the declared sets.

By a **variable binding** BVar is meant a binding of some or all of the variables $v_1, \ldots, v_n, bv_1, \ldots, bv_m$, and $bv'_1, \ldots, bv'_b$ to members, or internal surrogates of members, of the sets that label the bottom nodes of Tr. A variable v_i is bound to a member of V_i, and a variable bv_i or bv_i', is bound to an internal surrogate of a member of BS_i. A variable binding BVar' is an **extension** of BVar if the variables bound by BVar are all bound to the same entities in BVar'.

Consider assertion (1). Should the node nde that S labels be a bottom node, then that (1) is assigned **true** follows immediately from the definition of SU. It may be assumed therefore that nde is not a bottom node and that S is nonprimitive.

Let s be bound to the internal surrogate of a member of SU. Necessarily that internal surrogate takes the form of tup with its variables bound by a variable binding BVar, for which the assertion

3. tup:SU

is assigned **true.** It is sufficient to show that the assertion

4. tup:S

is also assigned **true** under BVar.

From (3) and the declaration of SU it follows that the assertion

5. tup:BDS **and [For some** $v_{j+1}:V_{j+1}, \dots, v_n:V_n$**]**
   ($\langle v_1, \dots, v_n \rangle$:T(Tr) **and** $\langle bv_1, v_{j+1} \rangle$:$IBS_1$ **and** ... **and** $\langle bv_b, v_{j+b} \rangle$:$IBS_b$)

is also assigned **true** under BVar. Therefore the following assertion is also assigned **true**:

6. **[For some** $v_{j+1}:V_{j+1}, \dots, v_n:V_n$**]**
   ($\langle v_1, \dots, v_n \rangle$:T(Tr) **and** $\langle bv_1, v_{j+1} \rangle$:$IBS_1$ **and** ... **and** $\langle bv_b, v_{j+b} \rangle$:$IBS_b$)

Necessarily there is an extension BVar' of BVar, in which the variables $v_{j+1}, \dots, v_n$ are bound to members of $V_{j+1}, \dots, V_n$, under which the following assertion is also assigned **true**:

$v_{j+1}:V_{j+1}$**and** ... **and** $v_n:V_n$ **and**
$\langle v_1, \dots, v_n \rangle$:T(Tr) **and** $\langle bv_1, v_{j+1} \rangle$:$IBS_1$ **and** ... **and** $\langle bv_b, v_{j+b} \rangle$:$IBS_b$.

From the declaration of T(Tr) the following assertion must also be assigned **true**:

$v_{j+1}:V_{j+1}$**and** ... **and** $v_n:V_n$ **and**
[For some $bv'_1:BS_1, \dots, bv'_b:BS_b, bv_{b+1}:BS_{b+1}, \dots, bv_m:BS_m$**]** Join'(Tr) **and**
$\langle bv_1, v_{j+1} \rangle$:$IBS_1$ **and** ... **and** $\langle bv_b, v_{j+b} \rangle$:$IBS_b$.

Therefore there is an extension BVar" of BVar', in which the variables $bv'_1, \dots, bv'_b, bv_{b+1}, \dots, bv_m$ are assigned to internal surrogates of members of $BS_1, \dots, BS_b, BS_{b+1}, \dots, BS_m$, under which the following assertion is assigned **true**.

$v_{j+1}:V_{j+1}$**and** ... **and** $v_n:V_n$ **and**
$bv'_1:BS_1$**and** ... **and** $bv'_b:BS_b$**and** $bv_{b+1}:BS_{b+1}$**and** ... **and** $bv_m:BS_m$ **and**
Join'(Tr) **and** $\langle bv_1, v_{j+1} \rangle$:$IBS_1$ **and** ... **and** $\langle bv_b, v_{j+b} \rangle$:$IBS_b$.

Since the assertion Join'(Tr) includes a conjunction

$\langle bv'_1, v_{j+1} \rangle$:$IBS_1$ **and** ... **and** $\langle bv'_b, v_{j+b} \rangle$:$IBS_b$,

and since the degree constraints are assumed to be satisfied, the internal surrogates to which $bv'_1, \dots, bv'_b$ have been bound are the same as the internal surrogates to which $bv_1, \dots, bv_b$ are bound. This follows from the fact that each of $IBS_1, \dots, IBS_b$ has upper degree 1 on its value set. Necessarily, therefore, the assertion

7. $v_{j+1}:V_{j+1}$**and** ... **and** $v_n:V_n$ **and**
   $bv_1:BS_1$**and** ... **and** $bv_b:BS_b$**and** $bv_{b+1}:BS_{b+1}$**and** ... **and** $bv_m:BS_m$ **and**
   Join(Tr) **and** $\langle bv_1, v_{j+1} \rangle$:$IBS_1$ **and** ... **and** $\langle bv_b, v_{j+b} \rangle$:$IBS_b$

is also assigned **true** under BVar". Since Join(Tr) includes (4) as one of its conjuncts, (4) is also assigned **true** under BVar", and therefore under BVar as well.

Note that in this half of the proof the only use made of the assumption on the satisfiability of the degree constraints concerned the upper degrees of $IBS_1, \dots, IBS_b$ on their value sets. That these are all 1 are the only degree assumptions necessary to show that (1) is assigned **true**.

Now consider the assertion (2). Let s be bound to the internal surrogate of a member of S. Again that internal surrogate takes the form of tup with its variables bound by a variable binding BVar. Under BVar the assertion (4) is assigned **true**. It is sufficient to prove that (3) is assigned **true** also. The following lemma is required for the proof.

Lemma 2: There is an extension BVar" of BVar for which (7) is assigned **true**.

Proof of lemma 2: Recall that the assertion of a node nd of Tr is the assertion tp:SS, where SS labels nd and tp is assigned to nd. A node of Tr will be said to be true for a binding of variables, if the assertion of the node is assigned **true** for the binding. Each conjunct of (7) is the assertion of a node of Tr. Therefore to prove the lemma, it is sufficient to prove that there is an extension BVar" of BVar for which every node of Tr is true.

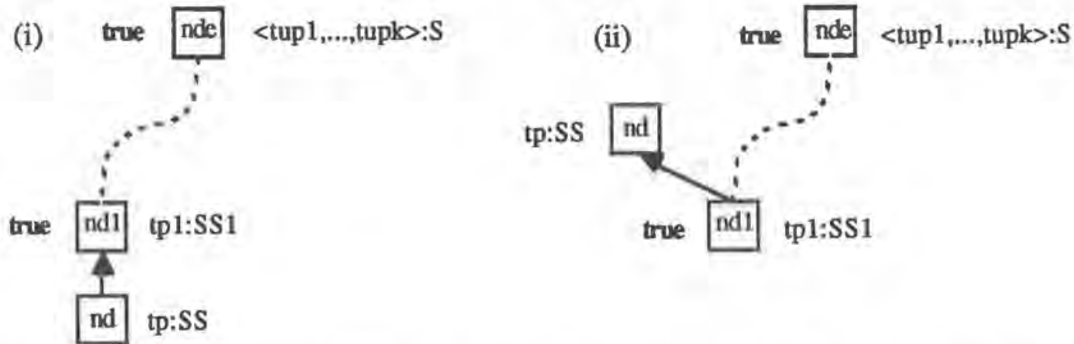
(4) is the assertion of the node nde of Tdg. Since Tr is a connected tree, there is a unique undirected path from nde to any other node of Tr. It will be proved by induction on the length of such paths that there is an extension BVar" of BVar for which the end node of any path beginning

at nde is true. Since the node nde is true for BVar, the result is true for paths of length 0.

Consider now a path nde, ... , nd_1, nd, where nd_1 may be nde. Assume that there is an extension BVar' of BVar for which all of the nodes nde, ... , nd_1 are true. It is sufficient to show that there is an extension BVar" of BVar' for which nd is also true.

Let tp:SS be the assertion of nd, and tp_1:SS_1 the assertion of nd_1. The latter is assigned **true** under the variable assignment BVar' since nd_1 is true by the induction assumption. The edge of the path connecting nd_1 and nd may have head nd_1 or head nd. The two possibilities are illustrated in figure 2.1.

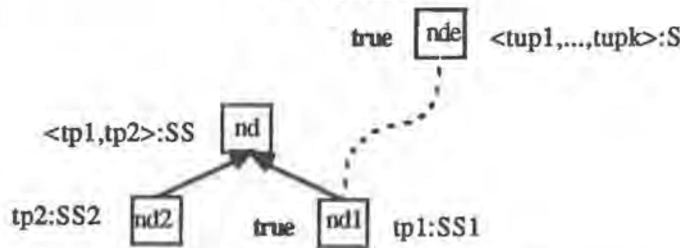
FIGURE 2.1



Consider (i) first. In this case SS is an immediate predecessor of SS_1, or of the arity domain of SS_1. Therefore tp_1 must either be tp, or of the form <... , tp, ...>. Since tp_1:SS_1 is assigned **true,** tp_1 must be in the domain of SS_1, so that tp:SS is assigned **true** also.

Now consider (ii). By lemma 1, the edge <nde_1,nd> necessarily has lower degree 1. Let nd_1, ... , nd_d, $d \geq 1$, be the nodes of Tr for which <nd_i, nd> is an edge of Tr, and let tp_i:SS_i be the assertion of nd_i. Therefore tp is tp_1 if d=1, and is otherwise <tp_1, ... , tp_d>. The situation is illustrated in figure 2.2 for d=2.

FIGURE 2.2



Since the edge <nd_1, nd> has lower degree 1, SS has lower degree 1 on SS_1. The theorem being proved assumes that all degree constraints are satisfied by the memberships of the declared sets. Therefore there must be an extension BVar"of BVar', binding all the variables occurring in tp, under which the assertion of nd is assigned **true.**

End of proof of lemma 2

In the first part of the proof of the theorem an argument was given that concluded that if (3) is assigned **true** under a variable binding BVar, then there exists an extension BVar" under which (7) is assigned **true** also. In this second part, the lemma establishes that if (4) is assigned **true** under a variable binding BVar, then there is an extension BVar" of BVar under which the assertion (7) is assigned **true.** Now consider the reverse of the argument used in the first part of the theorem. It is elementary to establish that (6) is necessarily assigned **true** under BVar", and therefore under BVar. Further, tup:BDS is assigned **true** under BVar, since BDS is the bottom domain of S and

tup:S is assigned **true**. Therefore (5) is assigned **true** under BVar, and hence (3) also.
End of proof of theorem.

In the first half of the proof of the theorem, the only degrees that were required to be satisfied are the upper degrees of 1 of $IBS_1, \ldots, IBS_b$ on their value sets. It is possible, therefore, to relax the definition of identifier by requiring it to have only the degrees <1,*> on the set it identifies, rather than <1,1>. But efficiency considerations often dictate that there should be a single identifying string for each member of a primitive base set.

The more interesting conclusion from this observation is, however, that SU is a subset of S, no matter what node S labels, or no matter from what subtree of the domain graph Tr was constructed. The worst that can happen is that SU loses members of S. The theorem establishes sufficient conditions that SU does not lose members. Therefore, rather than saying T(Tr) is correct for S when SU has the same extension as S, the tradition of the relational model could be followed and T(Tr) could be said to be **lossless** for S.

The assumption of the theorem concerning the satisfiability of the degree constraints will be discussed in 4.1.

3.2. A Kernel Schema

The SET model can provide a provably sound foundation for databases that can reference and describe themselves. Such a base will be sketched here, while the basis for its justification as "provably sound" will be sketched in 4.1.

In figure 2.9 of [Gil87] a summary of the declarations of all the sets of the schema for Simple University was given in a table. That table should be defined from a set schema, called a **kernel schema**, in the same manner that the tables for Simple University were defined from its set schema. The kernel schema can be thought of as the schema that will support the enterprise of set schema design. It must be capable, therefore, of storing the information contained in any set schema, including itself. The design of such a kernel schema, as a set schema that is at the same time a schema for its own "data dictionary". was one of the orginal motivations for developing the SET model. An incomplete kernel schema has been found to be useful in the teaching of the entity-relationship approach in undergraduate and graduate courses in database design since 1979 [Gil86b]. The equivalent of such a schema for relations is described in [Mark85].

The rudiments of a kernel schema are evident from figure 2.9 of [Gil87]. First the set of declared sets must be declared:

DSET for { DSET || all declared sets }.

DSET is a primitive base set that has as its members all declared sets. Since DSET is itself a declared set, it will be a member of itself. The first four sets declared for Simple University, namely STR, INT, L, and ≤, are also in the kernel schema, and are members of DSET.

The identifier for DSET is the declaration attribute DEC that associates the declaration of a set with the set. The declaration of the value set VDEC for the attribute requires the definition of the full syntax of assertions of DEFINE as well as of declarations, and is beyond the scope of this paper Assuming the declaration of VDEC, however, the declaration of DEC can be given:

DEC for { DSET, VDEC | <1,1>, <0,1> | the declaration attribute }.

Although the declaration of a set is its ultimate identifier, some abbreviation of the declaration is necessary for a reasonable syntax, and the name of a set, which is part of its declaration, is so used. In the set schema for SU, each set has a unique name so that the name attribute of sets could be used as an identifier in that model. But for practical modelling it is essential that some duplication of names be permitted. For example, there are likely to be several attributes on different sets all with the same value set VN, and it should be possible to call them all NAME.

When duplicate names are allowed, the name of a set appearing in isolation need not uniquely identify the set. In the context of an assertion of DEFINE, however, the name of a set should identify the set, or at least narrow its identification sufficiently to permit the system to request clarification from the user, or make intelligent guesses. The following rules for naming sets, using definitions given in 2.10 of [Gil87], appear to satisfy this requirement:

1. Each primitive set must have a name distinct from the name of any other set.
2. Sets of the same arity, with a common arity predecessor that is not a value set, must have distinct names.

The need for the first rule is transparent. The second rule reflects the fact that members of value

sets are generally not entities about which information is recorded, but are used in human-machine communication to record and retrieve information about other sets. Thus a variety of attributes all with the same value set VN, but on different sets, may have the same name NAME. But different attributes of the same set, whether inherited from the arity domain of the set or not, must have different names. If in a kernel schema it is necessary to declare attributes on sets of strings, for example on the set VDEC, then distinct names can be given to the attributes although by rule 2 they are not required to be distinct.

From the DEC attribute it is possible to declare the immediate domain predecessor, the immediate define predecessor, and the immediate predecessor associations as defined sets with domain DSETxDSET. The names given to them are assumed to be IDMP, IDFP, and IP, respectively. The set of primitive sets can then be declared as a defined set:

PSET for { x:DSET | **not** [**For some** y:DSET](x:IDMP:y **or** x:IDFP:y) | }.

The need for the assertion x:IDFP:y will be apparent shortly.

The ontology of sets, described in 2.7 of [Gil87] for a simple form of the SET model, admitted as primitive defined sets only the primitive value sets such as STR and INT. Other primitive defined sets, however, are needed for a kernel schema. They are not members of PSET because, although they do not have an immediate domain predecessor, they do have an immediate define predecessor. The most important of these is the set of entities that are members of members of PSET:

UV for { x:UV | [**For some** y:PSET] x:y | }.

UV is a defined set since its intension is stated in DEFINE, but it is a primitive set since there is not a previously declared set from which its extension can be drawn, since its members are drawn from the extensions of all the primitive sets. The members of UV form the basic universe of the model for the enterprise described in a set schema. All other entities are nested tuples of members of UV.

One of the important ways in which SET differs from the similarly motivated models of [LyKe86, BCP86] is that UV is a primitive defined set in SET, while in the other models it is a primitive base set. It is possible to declare UV as a primitive base set in SET, although that would offend the first principle of conceptual modelling stated in 2.14 of [Gil87], since an identifier for UV cannot be declared apart from the identifiers for the primitive sets.

The effect on the language DEFINE of admitting sets such as UV is profound. In the simple form of the model each entity that was a member of a set could be uniquely "typed" as a member of the arity domain of the set. The language DEFINE is monomorphic without such sets, and polymorphic with them, since a member of UV is also a member of some other primitive set [CaWe85, Ing86].

A second related effect concerns the arity of sets. Each set that can be declared in the simple form of the SET model has as members only tuples of length the arity of the set. That is no longer the case in the extended model. For example, the union of any two sets of different arity can be declared as a primitive defined set. By definition the set will have arity 1, although its members are not tuples of length 1, or even tuples all of the same length.

3.3. Recursively Defined Sets

It is necessary to declare as sets of the kernel schema some of the associations that were defined informally in section 2 of [Gil87]. For example, domain predecessor, the transitive closure of immediate domain predecessor, must be declared as a recursively defined set:

DMP for { DSET, DSET | [**For all** x:IDMP] x:DMP **and**
[**For all** u,v,w:DSET] **if** (<u,v>:DMP **and** <v,w>:DMP) **then** <u,w>:DMP |
the domain predecessor association }.

Although DMP is a defined set, its domain declaration does not declare variables. The declaration can be recognized as recursive from the fact that the machine readable portion of it between the two vertical bars is not a degree declaration, but rather an assertion of DEFINE.

The declaration appears to offend the requirement that the predecessor association be acyclic, since DMP is used in its own declaration. But when the declaration is properly interpreted, this is not so. The meaning of such a definition is that DMP is the smallest transitively closed subset of DSETxDSET that includes IDMP. A formal expression of this meaning requires a second order set theory of the kind introduced in [Gil86a]. For example, using second order variables, DMP can be declared:

DMP for { z:DSETxDSET | **[For all** X⊆DSETxDSET]
if (**[For all** y:IDMP] y:X **and**
[For all u,v,w:DSET] **if** (<u,v>:X **and** <v,w>:X) **then** <u,w>:X)
then z:X | }.

Note the quantifier for X ranges over subsets of DSETxDSET. The fact that this range is not a previously declared set, but rather all possible subsets of a previously declared set, makes it a second order variable that cannot be replaced by a first order variable without explicitly declaring every possible subset.

The form of the DMP declaration is one that has been widely used in logic for defining recursive sets. The assertions used in the **if ... then** clauses take the form of pure Horn clauses, that now form the basis for the programming language PROLOG. Restricting the intension of recursively defined sets to the use of pure Horn clauses avoids the complications that are necessary when PROLOG is extended to include nonHorn clauses [ABW86]. Given the ability to use any assertion of DEFINE in the intensions of nonrecursively defined sets, no loss of expressive power results.

The set TUP of all tuples of members of the set UV can be defined recursively in the usual way. With that set available, COUNT can also be defined recursively with domain TUPxINT, as can also SUM.

3.4. Updates and Data Processing

Once a set schema has been declared, a user must be able to add members to any declared set, and a command must be available for doing this. The form of a suitable command is:

Add tup **to** S **where** assert

where the variables occurring in tup occur unbound in the assertion assert; the latter provides an appropriate description of the entity to be added to the declared set S. A companion command removes an entity from a set:

Drop tup **from** S **where** assert.

In the most elementary form of the **Add** and **Drop** commands, S must be restricted to being base, and the meaning of the commands when S is defined must be reduced to the elementary form. For humans are responsible only for the membership of base sets, while the system is responsible only for the membership of defined sets. A command to add or drop a member of a defined set must, therefore, be interpretable as one or more commands to alter the membership of base sets. But how such commands are to be interpreted requires more research; the equivalent problem for the relational model is the updating of virtual relations, or views [Date83, Kell85].

In a kernel schema, the declaration of a set is equivalent to adding the set to the primitive base set DSET. The sets declared in the kernel schema itself are assumed predeclared and therefore have members determined by the schema. But when a member is added to DSET by a user, the declaration attribute DEC must be updated. During such a transaction, the system must ensure that cycles are not introduced into the immediate predecessor association; it is sufficient for the system to ensure that the immediate predecessors of a set are declared before the set can be declared.

In 1.1 of [Gil87] it was argued that a fully unified model of an enterprise is needed that at the same time can give a conceptual view of the enterprise, a user's view of data as it is presented, a data administrator's view of data as it is stored, and a programmer's view of the processing of the data. A commonly encountered perception of the latter is that the dynamic nature of data processing prevents the representation of a programmer's view in a "static" model such as SET.

In as much as the commands **Add** and **Drop** are extensions to DEFINE, the perception is soundly based. For example, it is not possible to represent the addition of a new employee to the primitive base set E as an association in the same sense that the assignment of an employee to a department is represented by ED: First, the new employee, before being added to the set E, is not a member of any set, so that a domain for **Add** is not available; and second, the effect of adding the employee to E is not to change its intension, but only its extension. When an entity that is a member of the domain of a nonprimitive base set is added to the set, a change of extension is the result. Such an application of **Add** can be regarded as an association between two states of the extensions of the declared sets. But that requires treating the extensions of all the declared sets as a tuple of tuples, something theoretically possible but often impractical.

A recognition of the need for **Add** and **Drop**, and the form of them given above, is one of the

contributions of [Morr].

On the other hand, each defined set can be regarded as a "program" that determines the membership of the set from the membership of its immediate predecessors. In this sense, therefore, the perception of the SET model as only supporting static descriptions is not soundly based. Typical data processing requires in part the use of the **Add** and **Drop** commands, sometimes imbedded in assertions, but also a substantial number of defined sets.

The two cornerstones of object-oriented programming, encapsulation and inheritence [Cox86], are central features of the SET model. These are the features also of the object-oriented data modelling methodology for information systems described in [LyKe86]. The objects of the SET model are the members of declared sets. The **list**, **Add**, and **Drop** commands provide the procedural element, as the comparable commands do in [LyKe86]. In the LORE approach to object oriented programming[BCP86], the procedural element is introduced through message passing, traditionally a part of object-oriented programming.

4. Consistency and Integrity

In the last section a sketch was given of a kernel schema that can reference and describe itself. It remains to be shown however that such a schema is a provably sound foundation for databases. A basis for a proof of this is given in 4.1. In 4.2 the satisfaction and maintenance of the integrity constraints of SET are discussed.

4.1. Consistency

"Consistency" is used here in the sense employed in logic and mathematics: A theory is consistent if it is not possible to conclude that both a sentence and its negation are true for the theory. The meaning given in [Date83] is included in the meaning of "integrity" used here.

In a consistent database in which all integrity constraints have been satisfied, it is not possible to conclude that a sentence and its negation are both true. But a consequence of the high level of abstraction tolerated in kernel schemas, is that such schemas might satisfy very strong integrity constraints but neverthless not be consistent, unless care is exercised in its implementation. Consider, for example, the following declared set:

R for{ x:DSET | **not** x:x | };

the members of R are those declared sets that are not members of themselves. The set E for example, is a member of R, while the set DSET is not.

There is nothing inherently wrong with the declaration of R, and it is even conceivable that it might prove useful for some purposes, but the set has a notorious past: It is the basis for the Russell paradox that shook the foundations of mathematics at the turn of the century. The paradox arises when one asks whether R is a member of itself. Using naive reasoning it can be concluded that R is a member of itself, and that it is also not a member of itself. Thus a database system that permits the declaration of R, or some similar set, and that permits naive reasoning will be inconsistent, even though it enforces very strong integrity constraints. This is what [Blac85] demonstrated for the semantic networks described in [Sowa84].

Since the discovery of the Russell and other paradoxes, a number of set theories have been invented that maintain consistency by restricting the declaration of sets in a variety of ad hoc fashions. In [Gil86a] it is argued that it is the reliance on naive reasoning, rather than the definition of R, that is the source of the paradoxes.

An assumption employed in naive reasoning is that every sentence is either true or false. A careful examination of this assumption shows, however, that it can only be made for atomic sentences, or what is called in theorem proving, ground sentences. The truth value for a complex sentence must be reduced to the truth values of simpler sentences, and be ultimately expressed in terms of the truth values of atomic sentences. If it is not possible to ground a truth value for a complex sentence in atomic sentences, then the sentence receives no truth value. It is this provably sound resolution of the paradoxes that is used to maintain the consistency of the SET model.

This simple resolution of the paradoxes has an important consequence. Some sentences, such as R:R, cannot be grounded in atomic sentences and therefore have no truth value assigned to them. On the basis of the semantics, the correct answer is "no" to a query as to whether R is a member of itself, since R:R is not true; it is also the correct answer to a query as to whether R is not a member of itself, since R:R is not false. Negation cannot therefore be understood in the sense of "negation

as failure" as suggested in [Clar78] and criticized in [Flan86]; the negation of an assertion is assigned a truth value if and only if the assertion has been assigned a truth value, and it then receives the opposite truth value. In this sense a negated assertion derives its meaning from the assertion that can be obtained from it by driving negations down to the level of assertions of membership in base sets. All assertions expressing membership in defined sets must be replaced with their intensions, with membership in recursive sets requiring "recursive" treatment. Once a query has been grounded in this sense, a truth value can be determined for it, but not before. Therefore a management system can respond to a query as to whether R is a member of itself, with a report of failure to ground the query.

4.2. Integrity in the SET Model

Two kinds of integrity constraints are expressed in the domain and degree declarations. The maintenance of these constraints presents different problems to a management system.

The maintenance of domain constraints is a relatively simple matter. There are no constraints on adding new members to primitive base sets such as E or D. Domain constraints affect users actions only when a new member is to be added to a nonprimitive base set such as ED; then the new member must be selected from the domain of the set.

The degree constraints on base sets present problems of a different character. There is first the question as to whether the degree constraints can be satisfied at all. Consider, for example, a base association R with domain PxQ and degrees <1,1> and <1,1>, where

P for { x:INT | x=1 }, and Q={ x=INT | x=1 **or** x=2 | }.

Clearly the degrees can never be satisfied since they require that P and Q have the same number of members. The satisfaction problem for degree constraints is unsolvable when a sufficiently rich form of DEFINE is available for defining sets; for example, the halting problem for a Turing machine can be expressed as the problem of determining whether two defined sets have the same number of members. But degree constraints satisfying one simple condition can always be satisfied.

A **membership specifying path** in a domain graph is a directed path in which the first node is labelled with a primitive defined value set, and every edge <nde1, nde2> for which nde2 is labelled with a base set has lower degree 1. For example, the nodes labelled with INT, P, and R form a membership specifying path, as do the nodes labelled with INT, Q, and R.

Theorem: Consider a set schema for which no base set labels a node of a membership specifying path. Then the degrees of the set schema are satisfied if every set that does not label a node of a membership specifying path is empty.

Proof: By induction on the maximum length of directed paths in the domain graph. If that length is 0, then every set that does not label a node of a membership specifying path is necessarily primitive base and can therefore be empty. Consider now a schema in which the maximum length of directed paths is greater than 1, and let S be a nonprimitive set labelling a node that is not in a membership specifying path. Should S be defined, then each immediate domain predecessor of S may be assumed to be empty, so that S may be assumed to be empty also. Should S be base, then each immediate domain predecessor of S that does label a node on a memberships specifying path, may be assumed to be empty. The lower degree of any edge from a node labelled with an immediate domain predecessor of S to the node labelled with S is necessarily 0 if the immediate domain predecessor labels a node of a membership specifying path. Therefore S may in this case also be assumed to be empty.

End of proof

Assuming that the degree constraints of a set schema can be satisfied, they nevertheless present special problems to a management system that must maintain them. The lower degree constraints are the most difficult since lower degrees of 1 may require that new members be added to an association such as ED in response to the addition of a new member to E or a new member to D. Combined with upper degree constraints, lower degree constraints may even compel the addition of a new member to a primitive base set. For example, adding a new member to D will compel at the very least a changing of the membership of ED, and could compel the adding of a member to E. Since there is no limit on the size of 1-connected subgraphs, there is no limit on the number of **Add** and **Drop** commands that must be executed in a transaction that transforms a database state in which all degree constraints are satisfied, to another such state. However, the existence of these

subgraphs permits transactions to be checked for degree preservation before being committed. The provision of assistance to users in the management of degree preserving transactions is an interesting research problem. Aspects of the problem are addressed in [ApPu87].

BIBLIOGRAPHY

[ABW86] APT, K., BLAIR, H., AND WALKER, A. Towards a Theory of Declarative Knowledge. *Proc. Workshop on Foundations of Deductive Databases and Logic Programming*. Washington, D.C. 546-629. 1986.

[ApPu87] APT, KRYSZTOF 9. AND PUGIN,JEAN-MARC. Maintenance of Stratified Databases Viewed as a Belief Revision System. *Proc. Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. 136-145. March 1987.

[BCP86] BENOIT, CHRISTOPHE, CASEAU, YVES, AND PHERIVONG, CHANTAL. The LORE Approach to Object Oriented Programming Paradigms. Memo C29.0, Laboratoires de Marcoussis, Centre de Recherches de la C.G.E. April 15, 1986.

[Blac85] BLACK, MICHAEL JULIAN. Naive Semantic Networks. Final Paper, Directed STudy in Computer Science. Dept of Comp. Sci., Univ. of B.C. Jan 22, 1985.

[CaWe85] CARDELLI, LUCA, AND WEGNER, PETER. On understanding types, data abstraction, and polymorphism. *ACM Comp. Surveys* 17, 4 (Dec. 1985), 471-522.

[Chen76] CHEN, PETER PIN-SHAN. The Entity-Relationship model - toward a unified view of data. *ACM Trans. Data Base Syst.*, 1, 1 (March 1976), 9-36.

[Chen77] CHEN, PETER PIN-SHAN. The Entity-Relationship model - A basis for the enterprise view of data. AFIPS Conference Proceedings, Vol. 46, 1977 NCC.

[Clar78] CLARK, K.L. Negation as Failure. H. Gallaire and J. Minker (eds.), Logic and Data Bases, Plenum, New York, 1978.

[Cox86] COX, BRAD J. Object-Oriented Programming. Addison-Wesley, 1986.

[Date81] DATE, C.J. An Introduction to Database Systems, Vol.I, 3rd ed. Addison-Wesley, 1981.

[Date83] DATE, C.J. An Introduction to Database Systems, Vol.II. Addison-Wesley, 1983.

[DKM86] DE TROYER, O., KEUSTERMANS, J., AND MEERSMAN, R. How Helpful is an Object-Oriented Database Model?. [DiDa86]. 124-132.

[DiDa86] DITTRICH, KLAUS, AND DAYAL, UMESHWAR. (Eds) Proc. International Workshop on Object-Oriented Database Systems. ACM and IEEE. Sept 23-26, 1986.

[Ethn86] ETHERINGTON, DAVID WILLIAM. Reasoning with Incomplete Information: Investigations of Non-Monotonic Reasoning. PhD Thesis, Dept. Comp. Sci., Univ. of B.C. April 1986.

[Flan86] FLANNAGAN, TIM. The Consistency of Negation as Failure. *Journal of Logic Programming*, 2 (1986), 93-114.

[Gil77] GILMORE, PAUL C. Defining and computing many-valued functions. Parallel Computers - Parallel Mathematics. FEILMEIER, M. (ed.), North-Holland (1977), 18-23.

[Gil86a] GILMORE, PAUL C. Natural deduction based set theories: a new resolution of the old paradoxes. *J. Symb. Logic*, 51, 2 (June 1986), 393-411.

[Gil86b] GILMORE, PAUL C. Class notes for CPSC 404. Dept of Computer Science, Un. of B.C. August 11, 1986.

[Gil87] GILMORE, PAUL C. The SET Conceptual Model and the Domain Graph Method of Table Design. Dept of Computer Science Tech. Report 87-7, Un. of B.C. March 1987.

[Kell85] KELLER, ARTHUR M. Updating Relational Databases Through Views. Stanford Computer Science Department Tech. Report STAN-CS-85-1040. Feb 1985.

[Knt81] KENT, WILLIAM. Consequences of Assuming a Universal Relation. *ACM Trans. Database Syst.*, 6, 4 (Dec 1981), 539-556.

[Knt83] KENT, WILLIAM. The Universal Relation Revisited. *ACM Trans. Database Syst.*, 8, 4 (Dec 1983), 644-648.

[KhCo86] KHOSHAFIAN, SETRAG N. AND COPELAND, GEORGE P. Object Identity. [Meyr86]. 406-416.

[Krey87] KREYKENBOHM, MICHAEL. Optimizing DEFINE Queries. Term Project, Dept of

Computer Science, Un of B.C. March 1987.
[LuKl86] LUK, W.S. AND KLOSTER, STEVE. ELFS: English Language for SQL. *ACM Trans. Database Syst.*, 11, 4 (Dec 1986), 447-472.
[LyKt86] LYNGBACK, PETER, AND KENT, WILLIAM. A Data Modelling Methodology for the Design and Implementation of Information Systems. [DiDa86]. 6-17.
[Mark85] MARK, LEO. Self-describing database systems - formalization and realization. Technical Report - #1484. Dept. Comp. Sci. Un. Maryland. April, 1985.
[McC80] Circumscription - A Form of Non-monotonic Reasoning. *Artificial Intelligence* 13, 295-323. 1980.
[Meyr86] MEYROWITZ, NORMAN. (ed.) Proc. Object-Oriented Programming Systems, Language and Applications. ACM Sigplan Notices, 21, 11 (Nov 86).
[Morr] MORRISON, RODERICK. Implementating a Set Based Data Model and its Data Definition/Manipulation Language. PhD thesis, Department of Computer Science, Un. British Columbia. *In progress.*
[Ship81] SHIPMAN, DAVID W. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst.*, 6, 1 (March 1981), 140-173.
[Sowa84] SOWA, J.F. Conceptual Structures: Information Processing in Mind and Machine. Addison-Wesley, 1984.
[TrLo87] TRYON, D.C.; AND LOYD, D.G. Information Resource Depository: History, Current Issues, and Future Directions. Pacific Bell, A Pacific Telesis Company. Presentation to Canadian Information Processing Society, Vancouver, Canada, February 1987.
[Ull80] ULLMAN, JEFFREY D. Principles of Database Systems. Computer Science Press, 1980.
[Ull82] ULLMAN, J.D. The U.R. Strikes Back. *Proc. ACM Symp. Principles of Databse Systems.* 1982, 10-23.
[Ull83] ULLMAN, J.D. On Kent's "Consequences of Assuming a Universal Relation. *ACM Trans. Database Syst.*, 8, 4 (Dec 1983), 637-643.
[VaTo87] VAN GELDER, ALLEN AND TOPOR, RODNEY W. Safety and Correct Translation of Relational Calculus Formulas. *Proc. Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* 313-327. March 1987.