THE UNIVERSITY *of* EDINBURGH

EVALUATION OF FUNCTIONAL DATA MODELS FOR DATABASE DESIGN AND USE

by

KRISHNARAO GURURAO KULKARNI

Ph. D.

University of Edinburgh

1983

To my wife, Rama

and

daughter, Bhuvana

Acknowledgements

I am deeply indebted to my thesis supervisor, Dr. Malcolm Atkinson, for his constant guidance, help, and encouragement, for his highly incisive comments on the writing of this thesis, and for a host of other reasons. It is indeed a great pleasure to acknowledge his contribution. I am also greatly indebted to Dr. David Rees, who supervised me during the initial six months. His help and guidance during that time was invaluable.

I am thankful to many of my friends in the Department for their kind advice from time to time. In particular, I would like to thank Paul Cockshott, Ken Chisholm, Pedro Hepp, Segun Owoso, George Ross, and Rob Procter for many useful discussions. I am also thankful to Moira Norrie for many useful comments on the content of this thesis.

I am also thankful to many of the database researchers outside Edinburgh for their advice and comments. In particular, I would like to thank Dr. Peter Gray from Aberdeen, Dr. Peter Stocker from East Anglia, Dr. Ron Morrison from St. Andrews, and Dr. Jochim Schmidt from Hamburg.

My thanks go also to the Department of Computer Science at the University of Edinburgh for providing the necessary resources to conduct this research, to the Commonwealth Scholarship Commission in the United Kingdom for awarding me a scholarship which provided the necessary financial support during the entire study period, to the Ministry of Education, Government of India, for selecting me for the award and to the authorities of Indian Telephone Industries Ltd., Bangalore, for granting the study leave.

This research was greatly helped by the PS-algol project and my thanks go to all involved in it. PS-algol project was partly funded by SERC grants GRA86541, GRC21977 and GRC21960 and a grant from ICL.

Abstract

The problems of design, operation, and maintenance of databases using the three most popular database management systems (Hierarchical, CODASYL/DBTG, and Relational) are well known. Users wishing to use these systems have to make conscious and often complex mappings between the real-world structures and the data structuring options (*data models*) provided by these systems. In addition, much of the semantics associated with the data either does not get expressed at all or gets embedded procedurally in application programs in an ad-hoc way.

In recent years, a large number of data models (called *semantic data models*) have been proposed with the aim of simplifying database design and use. However, the lack of usable implementations of these proposals has so far inhibited the widespread use of these concepts. The present work reports on an effort to evaluate and extend one such semantic model by means of an implementation. It is based on the functional data model proposed earlier by Shipman[SHIP81]. We call this 'Extended Functional Data Model' (EFDM).

EFDM, like Shipman's proposals, is a marriage of three of the advanced modelling concepts found in both database and artificial intelligence research: the concept of **entity** to represent an object in the real world, the concept of **type** hierarchy among entity types, and the concept of **derived data** for modelling procedural knowledge. The functional notation of the model lends itself to high level data manipulation languages. The data selection in these languages is expressed simply as function application. Further, the functional approach makes it possible to incorporate general purpose computation facilities in the data languages without having to embed them in procedural languages. In addition to providing the usual database facilities, the implementation also provides a mechanism to specify multiple user views of the database.

Table of Contents

List of Figures

List of Tables

Chapter One

Introduction

This thesis is concerned with the evaluation of **functional data models** for database design and use and in particular with the implementation of one variant of the functional data model, proposed earlier by Shipman [Shipman 81]. The implementation termed ***Extended Functional Data Model*** (EFDM) system, can be used as a tool for organising small database systems or as a prototyping tool for designing large database systems. The data model underlying EFDM incorporates many modifications which attempt to correct as well as improve some aspects of the data model proposed by Shipman. The implementation of EFDM is done using a novel programming language called *PS-algol* [Atkinson 81a], which has data persistence built into it as an orthogonal property of the data.

EFDM provides for an interactive language interface which is again based on the DAPLEX language proposed by Shipman. This interface allows for interactive creation, retrieval, and modification of both structure and contents of the databases modelled using this data model. It also provides tools to check for consistency of the schemas so designed. A notable feature of the implementation is the way it handles general-purpose computation. In contrast to the current approach of embedding query languages in a compile-and-run procedural languages, we have chosen to extend the DAPLEX language itself in an applicative way. A number of database applications have been developed using this implementation. As far as we know, this is the only complete and interactive implementation of the functional data model.

1.1 Motivation

Most of the data-intensive applications benefit immensely from a database management system (DBMS) to manage the associated collection of data, i.e., the database. Until a few years ago, the large computing costs associated with the database technology had made it beyond the reach of many users. However, the situation is changing as the computing costs are rapidly going down. It is no longer the case that the volume of data has to be massive before justifying the use of a database. This has in turn resulted in increased demands on the database management systems. In addition to the tasks related to storing and retrieving data, a DBMS is also expected to perform other functions which are presently done by system modules other than DBMS. Such functions include transaction management, exception handling and checking for consistency of the data as it is updated.

However, the three most popular types of database management systems, i.e., hierarchical, CODASYL/DBTG, and relational, are proving to be inadequate for meeting these increased expectations. It is difficult as well as time consuming to implement database systems using these DBMSs. Current state-of-the-art methods for database design are essentially trial-and-error, supported by neither a scientific foundation nor an engineering discipline [Yao 78a]. These DBMSs lack tools for the users to express their requirements in as *natural* a way as possible, to translate those requirements into an effective design, and to adapt the design to new and/or changing requirements [Fry 78]. Users wishing to use these systems have to make conscious and often complex mappings between the real world structures and the data structuring options (*data models*) provided by these systems.

In addition, the data models provided by these systems are essentially syntactic, i.e., the data model structures do not explicitly contain any real-world meaning. Much of the semantics associated with the data either remains unexpressed or gets embedded procedurally in application programs in an ad-hoc, distributed and often repeated way. Consequently, it is difficult for the users to interpret the information contained in such databases. The less explicit the meaning of data, the greater the likelihood that its significance will be misunderstood.

Using such systems poses another set of difficult problems. Even simple applications demand the services of expert programmers and the programs are invariably complex, difficult to design, implement, and maintain. Changes in requirements of the enterprise often cause massive re-programming. In addition, current database management systems assume little

responsibility for maintaining the consistency of the data they manage. The fact that data integrity is required to be enforced by measures outside the DBMS leads to an immense software maintenance/management task whose cost increases much faster than the number of applications.

These deficiencies have triggered intense research work in the data modelling area. Data modelling has been one of the major themes of database research for more than ten years [Kerschberg 76, Tsichritzis 82]. Various researchers have proposed either extensions to existing models or a number of new *semantic data models*. Basically, these models aim to capture, in a more or less formal way, the meaning of the data so that database design can become systematic and the database itself can behave intelligently. They do this by modelling the information in terms of irreducible objects or *entities* as well as by providing constructs to show more of the relationships between data objects and the permissible operations upon them. Consequently, semantic models have the inherent ability to support high level query languages as well as to respond to queries and other transactions in a more intelligent manner [Codd 79].

However, most of the new data models proposed in the literature are incomplete in the sense that they are concerned primarily with modelling database structure while significantly less attention is paid to the operators or manipulation aspects of the data they attempt to describe. To be usable, it is most critical for a data model to have a consistent, complete, and simple set of operations on the data it models. The quality of the data model cannot be assessed independently of this set of operations.

In addition, most of these data model proposals are not followed up with actual implementations and consequently work on the tools for designing or maintaining the databases based on these models is almost non-existent. It is difficult to determine the efficacy of a data model without using it in anger over a range of applications. An implementation, by allowing the users to actually experiment with the concepts, can draw attention to the power of the model much more effectively.

It is in this context that the implementation and evaluation of a semantic data model assumes significance. The work reported here is such an attempt. The principal reasons for choosing the *functional data model* proposed earlier by Shipman [Shipman 81] for this task are its powerful data structuring facilities, the set of operations it provides, and its simplicity.

The data structuring facilities of this model are based on three of the advanced modelling

concepts found in both database and artificial intelligence research, the concepts being: **object orientation**, **type hierarchy**, and **derived** or **virtual data**. By ***object orientation*** we mean the ability to distinguish objects from their external names. By ***type hierarchy*** we mean the ability to describe objects at different levels of abstraction. By ***derived data*** we mean the ability to treat procedural knowledge or rules about the application domain as part of the data.

This data model also meets another of our important criteria, i.e., a complete set of operations on the data it models. The functional orientation makes it possible to express data selection in a most *natural* way as function applications. The object orientation provides *natural* units for update and because of this, the referential constraints [Date 81] are automatically supported.

The most attractive feature of this model, however, is its simplicity and the uniformity of its notation. It provides a semantically rich modelling power with just two concepts, i.e., **entity** and **function**.

Hence, the major goal of this research was to examine the usefulness of Shipman's proposals by an actual implementation and suggest a list of explicit modifications required to make the model and the language usable by a wide variety of users like database designers, database administrators, and end-users.

1.2 Overview of the Thesis

Chapter 2 represents the author's own perception of the published research material on conceptual modelling of databases. It starts with the discussion of the elements considered important for conceptual modelling of real world applications in the context of databases. It also lists a set of criteria for assessing a particular data model proposal. This chapter also considers the *classical* data models in the light of criteria described earlier and points out a number of problems which make them inadequate for the purpose of conceptual modelling.

Chapter 3 describes, albeit very briefly, twenty one of the semantic data model proposals and provides a rough assessment of these based on the set of criteria mentioned above.

Chapter 4 discusses the functional data model as proposed by Shipman and shows how it meets the above set of criteria.

Chapter 5 describes the underlying model of our implementation, EFDM, concentrating on the modifications and extensions we have made to Shipman's proposals.

Chapter 6 briefly discusses the persistent algorithmic language PS-algol and the issues connected with the actual PS-algol structures used for implementing functional model constructs.

Chapter 7 discusses a few applications of the EFDM. In particular, we show how it provides a flexible DBMS environment by means of an example. We demonstrate how to design, set up, operate, and maintain a personal DBMS using the facilities offered by EFDM. We also show how it can be used as a prototyping tool for designing large database systems.

Chapter 8 discusses some of the topics for further research suggested by our work. We also discuss a few of the weaknesses of our system and suggest a broad line of attack for solving some of these problems.

Finally, chapter 9 summarises the important lessons learned from this research.

Chapter Two

Background

This chapter establishes the broad context in which the work reported in this thesis fits, i.e., the *conceptual data modelling*. That is, to capture and identify the relationships between the data in the database and the corresponding objects and behaviour in the real world. For database technology to be used effectively, there is a need for understanding and visualising the data and the information they represent. Conceptual data modelling is concerned with this need.

We start this chapter with some preliminary definitions and the research leading to the three schema architecture proposal by ANSI SPARC. We then give a putative definition of the *data model* which acts as a tool to design the conceptual model of a database. We provide a detailed discussion of the data model elements. We identify a set of requirements which a data model should satisfy in order to be considered for the purpose of conceptual modelling. Against this set of requirements, we examine the data models underlying the contemporary database management systems.

2.1 Preliminary Definitions

An *information system* is a means of supplying the information needed by an organisation. An information system receives the information, stores it, processes it, and provides access to it at the request of the users. When information is to be stored and processed, it needs to be coded into some descriptive form. Such coded information is called *data*. A collection of data stored on a physical media is termed *database*. A *database system* is an information system involving four major components: database, hardware, software, and users. Users interact with the data in the database through a number of *user interfaces*.

Data as stored in a database have a certain *physical organisation* on physical storage media and a certain *logical organisation* as seen at the user interface. It is important to insulate the users from the physical aspects so that they are not distracted by the details of physical storage and are not inconvenienced if it is changed. A *database management system* (DBMS) is a general-purpose tool that accommodates the logical structuring, physical storage, and control of data, and provides a number of user interfaces.

An *application system* or an *application* is a part of the database system that generates the information required to serve a specific component of an organisation, e.g., accounting. A *view* is a part of the database as seen by a processing activity of the application system to perform a specific function, e.g., accounts payable, accounts receivable, etc.

Databases are primarily concerned with the *structured* or *formatted* data, i.e., many *instances* of data possess sufficient similarity to classify them into a *class* or *category*. This makes it possible to separate the description of the data from the actual data. The rules that the instances of a class are expected to obey are specified once, in a *schema*. Hence, the schema contains the description of the data.

2.2 Development of the Conceptual Model

The evolution of database systems from primitive file systems organised sequentially on magnetic tapes to sophisticated systems using direct access storage technology has been well documented elsewhere [Fry 76, Atkinson 79]. It was characteristic of the early database management systems that the user had to view and manipulate the data structures in the configuration in which they are physically stored. Thus the user was forced to deal with aspects of physical data organisation, pointers, index tables etc., which, although important for machine efficiency, are irrelevant to the user's understanding of the data. Any alteration of that organisation in order to improve efficiency necessarily involved considerable reprogramming effort.

An important landmark in the development of DBMS was the attempt by the CODASYL Database Task Group (DBTG) to identify a system based on a network model of data [CODASYL 71]. It recognised the value of explicit data description and suggested a DBMS structure which separated the logical structure from the physical structure of the data. It also recognised the fact that a database system needs to support many different views of the same database. However, the structure proposed by the CODASYL committee did not make a complete distinction between the logical and physical aspects.

It was the relational model of data proposed by Codd [Codd 70] which allowed a description to be made exclusively in terms of logical aspects of the data. Further research in this area has culminated in the ANSI SPARC report [ANSI 75, ANSI 78], which introduced a new component into the model of the databases called the *conceptual schema* or the *conceptual model*, and their model of a database system is known as a three-level model. The three levels in the ANSI SPARC report refer to:

1. *external level* which deals with the model of the real world as seen by an application;

2. *conceptual level* which deals with the model of the real world maintained for all applications;

3. *internal level* which deals with the physical data maintained for the representation of the conceptual model.

A fundamental concern behind this proposal is the notion of *data independence*. Because of the separation between the conceptual and internal levels, effects due to changes in physical organisation can be minimised (*physical data independence*). Similarly, because of the separation between external and conceptual levels, an application's view is insulated from changes in the schema of a database (*logical data independence*). However, ANSI proposal makes no attempt to define the precise formalisms for the description of the conceptual model, the description of mappings from one level to another, or the required transforms to translate the actions at one level to those at another.

2.3 Roles of the Conceptual Model

As seen above, it is the conceptual model of a database described in logical terms which represents the information content of that database. It records as precisely, clearly and unambiguously as possible the intended semantics of the data. Bubenko [Bubenko 80a] lists the following two main uses of the conceptual model:

1. To act as a basis for discussions and negotiations about how to abstract reality and which assumptions and rules to build into the model. This includes the problem of 'integrating' the requirements and 'views' of various information consumers.

2. To constitute part of a basis for the design of an efficient computer based information system including structuring of its database.

2.4 Data Models

A data model is the primary tool for designing the conceptual model of a database. The basic components of such a data model include a set of rules to describe the structure and meaning of data in a database and the atomic operations that may be performed on the data in that database. Thus, a *data model* **M** can be defined as consisting of two parts: a set of generating rules, **G**, and a set of operations, **O** [Tsichritzis 82].

G defines the allowable *structures* for the data as a set of schemas **S**. The set of generating rules **G** can be partitioned into two parts: the structure specification G_s and the rule specification G_r. The generators G_s generate the categories and structures of a schema and the generators G_r generate the inferences and the constraints associated with a schema. A schema S therefore consists of two parts; a structure part S_s and a rule part S_r. The rule part S_r further consists of an inference part S_i and a constraint part S_c. S_i is a list of inference rules that allow one to deduce or compute facts from others. S_c is a list of *explicit* constraints that should not be violated.

As well as explicit constraints, a data model can also provide *inherent* constraints. Inherent constraints can be associated with a data model by incorporating them in the structure part S_s, i.e., the structure by its own definition can disallow certain structure instances.

There are many different databases **D** in terms of occurrences which can correspond to the schema S. A *database state DBS* corresponds to a particular database occurrence D. The set of operations of a data model **O**, called a *data language*, defines the allowable actions that can be performed on a database occurrence D_i to arrive at another database occurrence D_j, i.e., each operation O maps one database state to another database state O: DBS_1 -> DBS_2.

2.5 Structures

In discussing information structuring, we find that there is still no mathematically accurate theory with well defined terms and concepts. Notable work in this area is reported by Langefors [Langefors 66], Senko [Senko 73], Sundgren [Sundgren 74], Abrial [Abrial 74], Smith and Smith [Smith 80a], Bubenko [Bubenko 80b], and more recently by the International Organization for Standardization (ISO) [ISO 82]. Unfortunately, all of them have adopted their own terminology. Here, we shall employ popular informal terms like "entity", "entity type", "attribute", "relationship" etc. to describe the structure and the semantics of information.

2.5.1 Entities

An *entity* can be loosely defined as a thing that exists and is distinguishable [Senko 73, Hall 76a]. Examples of entities are **person** *named* **John**, **department** *named* **Computer Science**, **university** *named* **University of Edinburgh** etc.

Some entities correspond to real world objects while some entities are used as names for something else. The latter are distinguished by having a lexicographic representation, i.e., each such entity is represented by a printable token for itself. For example, strings, integers etc. We call these *lexical entities*.

2.5.2 Designation of Entities

When an entity is entering the "perception field" of the model it must identify itself as a new entity or an already known entity. An easy way to do it is to give a name to any entity which identifies it without ambiguity. For example, a social security number for a person or a number for a car. This approach is found to have many disadvantages [Kent 78]:

1. An entity may not have any unique name and still be distinguishable.

2. An entity may have more than one unique name.

3. An entity may change its name over a period of time.

Hence, the unique designation of entities should be the responsibility of the conceptual model. Abrial proposes that each newly created object be assigned an internal name that is guaranteed to be unique [Abrial 74]. Hall et al. [Hall 76a] use the term *surrogates* for such internal names. Such surrogates are characterised by the fact that they are not accessible to the users. Entities can, of course, have other user-controlled unique names.

2.5.3 Entity types

Certain objects in the real-world invariably share some common properties. For example, all persons have the properties like name, address, age etc. An *entity type* or a *category* is a conceptual representation that corresponds to a categorisation of real world objects based on some common characteristics shared by them. As Senko, et al. [Senko 73] note, recognising and taking advantage of the distinction between entity types (such as sets of objects) and entities (such as individual objects) offers great power in building database

systems. Since information that is common to all instances of a collection is placed into type descriptions, the concept of entity type provides a powerful method of organising, simplifying, and condensing the information about groups of objects.

A strict categorization of objects is not always possible in practice [Kent 78, Kent 79]. For example, the same person may be an employee, a stock holder, a customer etc. Hence, an entity can be a member of more than one entity type, i.e., the extensions of different entity types can overlap. This corresponds to the idea of *roles* proposed in the context of semantic networks [Hayes 77]. A similar idea of roles was introduced by Bachman and Daya in data models [Bachman 77].

2.5.4 Type Hierarchy

Often, certain objects may share only some properties, while having some properties unique only to them. For instance, in a personnel information system, some information requirements may concern all employees and others may concern subsets of employee set, such as managers, researchers, female members etc. Thus, some entity types are, by definition, subtypes of others, making a member of one entity type automatically a member of another. This kind of hierarchical organisation of entity types was first introduced in the artificial intelligence field as *ISA* hierarchies in semantic networks [Quillian 68] with the precise inheritance rules for related types within the hierarchy. A similar notion of type hierarchy was introduced in the database field by Roussopoulos et al. [Roussopoulos 75] and by Smith and Smith [Smith 77]. Smith and Smith call this as *generalisation* hierarchies. Accommodation of type hierarchy is now accepted as an important element in conceptual modelling.

It is to be noted that a strict hierarchical organisation of entity types is also not possible in practice. For example, in a database concerning customers, a customer may be a company, government agency, or a person. Hence, an entity type may be a subtype of more than one entity type. This idea of a network of entity types is termed *alternative generalisation* by Codd [Codd 79].

2.5.5 Attributes and Domains

Attributes of an entity type refer to those properties of objects which can be, in a general sense, given values [Brown 75]. For example, the *person* entity can possess attributes like name, height, date of birth etc. Values are essentially lexical entities like integers, strings etc.

A *Domain* is a set of values of similar type [Codd 79]. Domains serve to define sets of values from which properties of other objects can take values over time. For example, six-digit numbers form a domain from which salaries of employees can take values. Since values are lexical entities, domains can also be considered as sets of lexical entities.

2.5.6 Relationships

Relationships represent association between several objects [Kent 78]. Examples of relationships are **person** named John *works in* the **university** named University of Edinburgh, **person** named Peter *belongs to* the **department** named Computer Science etc. A *relationship type* corresponds to a collection of similar relationships or an aggregation of two or more entity types. For example, *enrolment* can be represented as an aggregation of the entity types *student* and *course*.

Relationships can be **one-to-one** (departments and managers), **one-to-many** (departments and employees), **many-to-one** (employees and managers), and **many-to-many** (students and courses). There can be many relationships between the same set of objects and the objects participating in a relationship may or may not all belong to same entity type. Relationships can have a single, neutral name such as *enrolment* or two names, such as *course-of* in one direction and *students-of* in the other direction.

2.5.7 Entity Types, Attributes, Relationship Types

Much discussion has centred around the concepts of entity types, attributes and relationship types and their representation and materialisation [Kent 78]. There is no absolute distinction between entity types and attributes or between attributes and relationship types or between entity types and relationship types. Sometimes an attribute can exist only as related to an entity type. In a different context, it can be an entity type in its own right. For example, to a car manufacturing company, a colour is merely an attribute of one of its products; to the company that made the paint, a colour may well be an entity type [Hall 76a].

Similarly, there is no clear distinction between an entity type and a relationship type. In the above example of relationship between students and courses, it can either be considered as a relationship between *student* and *course* types with *grade* as a relationship between *student*, *course* and *integer* entity types or as an entity type *enrolment* with *grade* as the property of this new entity type.

There is no clear distinction between attributes and relationships either. For example, *salary* can be treated either as an attribute of *employee* or as a relationship between *employee* and *integer* entity types.

2.5.8 Time

Attributes and relationships in the real world do frequently change. So do the set of participating relevant entities. It is necessary that these properties are given explicit consideration in the conceptual model. The concept of *time* is therefore fundamental in the realm of conceptual models [Langefors 77].

However, time is perhaps the most cumbersome aspect of data modelling. To begin with, real time implies a certain synchronisation between phenomena which is unrealistic. Also, we are often more interested in the relative time of phenomena (i.e., one phenomena occurs before another phenomena) than we are in the real time of occurrence. This aspect of data can be adequately captured by ordering phenomena rather than recording their real time. Therefore, the notion of time is often replaced either by other kinds of explicit properties or by orderings among objects or by the convention that all data relates only to the last meaningful datum (phenomenon, fact or instance). For instance, the *salary* attribute of an employee does not capture the knowledge of all the salaries he or she has ever received, only the current one. If there is interest in past salaries, this is encoded using a different property (e.g., salary history). Some data models, however, do treat time as one of the modelling constructs [Bubenko 77, Breutmann 79, Schiel 83].

2.5.9 Conceptual Data Modelling and Knowledge Representation

There are a number of similarities between the information modelling for establishing database systems and the *knowledge representation* problem in artificial intelligence (AI) area [Roussopoulos 75]. Both databases and AI systems must represent and process knowledge about the real world. In particular, AI research also assumes that knowledge consists of

objects and relationships among them. A knowledge base may be viewed as consisting of a network of objects (nodes) connected by relations (directed edges). The directed edges are labelled with the type of each relation. These networks are called *semantic networks* [Quillian 68]. Some of the concepts in new data model proposals owe their origin to the knowledge representation research. For example, the concept of subtypes.

However, the goal of these networks is the representation and organisation of general knowledge of the world as opposed to the information structured in a manner oriented towards database applications. Specifically, as instances of types appear in semantic network definitions, schema and data are unified. One implication of this unification is that large amounts of data cannot be handled conveniently, as the network grows with the addition of new data (even when no new types or attributes are added).

2.6 Operations

As defined earlier, the operations of a data model, called a data language, transform a database state DBS_i to another database state DBS_j (or undefined). When we perform operations on a database, it is a natural restriction to focus them on one small part of the database. This focusing is important both for user convenience and for the ability to concentrate on a few narrowly defined tasks at one time. Focusing on a certain part of the database implies a *selection* [Earnest 75]. Regardless of the operation that is to be performed, this selection needs to be specified. For example, the selected database part may be retrieved or updated, new data may be inserted into it or old data deleted from it.

The operations usually follow a pattern of specifying an action and a selection. The *action* specifies what is to be done. The *selection* selects the part of the database to which the action is to be applied. An action may correspond to any one of, or a combination of, the following operations:

1. Retrieve - access the data in the database.
2. Insert - add new data to the database.
3. Delete - remove data from the database.
4. Modify - modify existing data in the database.

2.7 Rules

Rules correspond to the set of inference mechanisms to provide the derived data (see below) and the set of constraints that limit the number of different facts which can be part of the model. Rules extend the semantics of the data captured by the structures above. Bubenko [Bubenko 80a] calls the information captured by such rules as *abstract knowledge* as opposed to the *concrete knowledge* captured by the structures.

The specification of rules is an important part of conceptual modelling as most of the facts in the real world are mostly derived rather than pure data. As an example, consider the chart of accounts for a firm. The only pure data needed are original journal entries; all other facts are derived by manipulating this data.

The proposals of the CODASYL Data Base Task Group include the concepts of *database procedures* and *virtual* data items which are essentially the mechanisms to capture such rules [CODASYL 71]. A number of other authors are also investigating the deduction mechanisms for relational databases [Chang 78, Minker 78]. Many recent data model proposals attach a great importance to the concept of derived data [Hammer 78, Shipman 81].

The concept of rules or inference mechanisms forms an important component of the *expert systems* in the artificial intelligence field [Nau 83] and is the basic notation used in the logic languages such as PROLOG [Kowalski 74].

2.7.1 Derived Data

The *derived data* or the *virtual information* can be defined as the information which is accessible through combinations of algorithms and stored data, but which is not physically stored in the database. For example, given an employee's department and the manager of a department, we can infer the employee's manager or given the selling price and buying price of a product, we can infer the profit one can make.

2.7.1.1 Classes of Derived Data

Two major classes of derived data deal with the issues of extracting factual content: **inferred data**, and **computed data** [Folinus 74].

Inferred data: Consider the two relationships, one between *student* and *tutorial* entities and the other between *tutorial* and *staff* entities. It is then a straight-forward matter to infer the relationship between *student* and *staff* entities.

When inferring facts this way, an important point should be noted: not all such inferred relationships may be meaningful or even correct. For example, if a part is available from a certain warehouse, and a warehouse is serviced by that supplier, we cannot infer that the part is stocked in that warehouse. Even if it was stocked in that warehouse, it might be a different supplier who supplied that part to that warehouse. This is the familiar *connection trap* [Codd 70] whereby an erroneous inference may be drawn from the *join* of two relationships on a common domain. Hence, the need to explicitly identify which inferences can be made.

Computed data: Whereas inferred data is developed merely by accessing facts available in the database system, computed data is derived by processing algorithms. For example, given an entity such as a *room* whose attributes are *length* and *width*, its area could be defined by the algorithm *product(length, width)*.

2.7.1.2 Applications of Derived Data

Some of the important applications of derived data in the database context are listed below:

1. Derived data as a modelling tool: Derived data mechanisms are useful to accommodate complex, procedural inter-relationships among data items. For example, the age of a person, if expressed as a stored data item, requires continuous updating to reflect the accurate value. On the other hand, derived data mechanism allows it to be expressed as a formula (*current date* - *date of birth*).

2. Derived data as a means of capturing data semantics: A large part of the application specific *knowledge* exists not as stored data but as a set of programs. By modelling these rules or inference mechanisms as part of the database, the burden on the users can be much reduced.

3. Derived data as a basis for multiple views: A major application of derived data is to provide multiple user views of the schema. As no single model of reality may be appropriate for all users, it is important that the information is structured according to the properties which are considered relevant by each user. Derived data makes it possible to provide this logical restructuring of the same stored data.

4. Derived data as a means of hiding database evolution: In order to provide an illusion of stability in the face of constant changes, it is important to ensure that only those users concerned with the changes are affected and the rest are not. Derived data makes this possible by re-establishing the 'old' view by suitable transformations.

5. Derived data and the distributed database management systems (DDBMS) and database servers: Sharing of information between components in the DDBMS and database servers will be made easier with the presence of derived data. Components may easily build upon each others activities without having to tediously generate the required information from the stored data.

2.7.2 Constraints

There are usually many properties of data that cannot be captured in the form of structures. These properties essentially serve as additional restrictions on the values of the data and/or how the data may be related (structured). For example, there may be a restriction that a department's head must belong to the same department or a student may be required to attend a minimum of 3 and a maximum of 5 courses in a term. Such restrictions cannot be expressed in terms of structures, but must be captured by some additional mechanism. These logical restrictions on data are called *constraints*.

A constraint is a property which is either true or false. The constraints are expected to be true always. Constraints are required for semantic and integrity reasons. In terms of semantics, they permit schemas to reflect more accurately the real-world situation. In terms of integrity, they permit the DBMS to restrict the possible database states that can be generated from a given schema to those that meet the constraints.

2.8 Requirements of Data Models

In order to be useful to design conceptual models of database systems, data models must meet the following requirements:

1) Object orientation

By object orientation, we mean that a data model should make an explicit distinction between an object and the name(s) used to identify it. User-defined names of objects should neither act as unique designators of objects nor should they participate in defining relationships. Failure to do so results in users being forced to be aware of the mapping from real-world objects to values which act as tokens for them in the stored data and in some cases, to invent and maintain such tokens. On the other hand, models which make such distinction provide many advantages, viz.,

1. Objects can have different names in different applications or may not have any name at all.

2. Names identifying objects can be changed freely.

3. Semantics of both data retrieval and data manipulation operations can be clearly understood.

4. General rules to guarantee the consistency of the data as it is updated can be specified easily since the changes in the application environment can be accurately modelled as operations on the corresponding structures.

2) Semantic expressiveness

A data model should provide sufficient mechanisms to allow a database schema to describe the meaning of a database. Hammer and Mcleod call this aspect *semantic expressiveness* [Hammer 81]. There is, as yet, no proper definition of this concept, nor any yardstick by which it can be measured. Roughly, it could be considered as the ability to name and manipulate objects at different levels of abstraction. As we discussed in section 2.5, a data model should provide constructs to organise entities into entity types, to organise entity types into subtype–supertype hierarchy, and to establish relationships between two or more entities. Corresponding to these, Smith and Smith [Smith 80a] list three abstraction mechanisms: *classification*, *generalisation*, and *aggregation*. Classification collects instances to form 'types'. Generalisation refers to supertype–subtype hierarchy among types. Aggregation refers to relationship between two or more entities. In addition to the above three, some more abstraction mechanisms have been suggested by Hammer and Mcleod [Hammer 78], and Codd [Codd 79].

3) Ease of design

It should be easy to arrive at a schema, i.e., users should not be forced to make complex mappings from the real–world structures to the data structures supported by the data model.

4) Neutrality

As discussed in 2.5.7, a data model should allow a fact to be interpreted either as an entity, an attribute, or a relationship.

5) Operations

A data model must act as a basis for the development of families of very high level data languages for data retrieval and data manipulation. A data language should specify the data selection purely on the basis of logical relationships and not by the physical position of the data. Additionally, in case many instances of an entity type are to be selected, there should

be no necessity to know the path to go from one instance to another (*navigation*). Explicit navigation often complicates the selection request.

A data language should provide update operations in terms of objects. It is important to ensure that objects participating in a new relationship already exist in the database and whenever an object is deleted, the relationships from the deleted object to all other objects are removed.

6) **Facilities to specify constraints**

A data language should provide constructs to specify explicit constraints on the data. As a general principle, the specification of constraints should be migrated out of the application programs both to reduce application programming costs and to ensure quality in data and its use [Nijssen 80].

7) **Facilities to specify derived data**

A large part of the information exists not as data but as programs. A data model should allow incorporation of such derived data as part of the schema. To accomodate this, a data model should treat data and programs in an unified way.

8) **Freedom from physical considerations**

The structures and operations of a data model should not refer to any physical storage aspects.

9) **Ease of evolution**

It must also provide a structure to which the progressive details associated with further steps in the design can be conveniently attached. This means that a database can be incrementally designed, which is important for a complex system like a database.

10) **Complexity**

A data model must not be too complex. The languages it supports must also be easily usable. Simplicity is attractive to the user in moderating the effort of learning and comprehension needed, and to the implementor in limiting the size of his task.

2.9 Classical Data Models

By classical data models we mean the three most common and popular data models used in contemporary database management systems, *viz.*, hierarchical, network, and relational data models. The best known implemented system using the hierarchical model [Tsichritzis 76] is IBM's IMS (Information Management System) [IBM 75]. The most comprehensive specification of a network data model is in the CODASYL DBTG report [CODASYL 71, Taylor 76]. There are many systems based on these proposals, such as IDS [Honeywell 72], IDMS [Cullinane 75], etc. Examples of systems based on the relational data model [Codd 70] include INGRES [Stonebraker 76, Stonebraker 80] and System R [Astrahan 76, Blasgen 77].

Briefly, the hierarchical and the network data models incorporate the concept of a *record* as a collection of named fields to represent each individual object in the application environment. In addition, the hierarchical model allows a tree-like set of one-to-many relationships in which each record occurs at a single specified level of the hierarchy. The CODASYL model provides the *set* mechanism to establish one-to-many association between any *owner* record and a number of *member* records, thus allowing a network of relationships. The relational model accommodates only record types and not explicit links. It is based on the concept of mathematical relations. Logically, a relation is a collection of instances (*tuples*) of a record type in which the sequencing of the instances and the sequencing of the fields within the record type are unimportant. The relationships between relations are not explicitly specified in the schema. Logically, relations describe entities and inter-relationships among them. Inter-relationships which are not represented as tuples can be dynamically established at access time using the relational data manipulation facilities (*joins*). Thus, unlike other two models, the user is not restricted to the pre-defined relationships.

The retrieval and update languages for systems based on the hierarchical and network models, called *data manipulation languages* (DML) tend to be *navigational*, in the sense that a user must access a database by explicit traversal through a tree or network, rather than by stating the properties of the data of interest. As a result, users are required to write complex programs in procedural fashion to carry out retrieval and update functions. In addition, the exact form of the statements available to the programmer to conduct this navigation depends on the way the data is stored, which adds to the complexity of these languages.

Data manipulation languages for the relational data model are typically derivatives of *relational calculus* or *relational algebra* [Codd 70]. These languages are essentially *non-procedural*, i.e., operations in these languages are specified in terms of names and values only and hence do not require the knowledge of the physical representation of data.

Detailed descriptions of the significant features of these models are well covered in many books and surveys [Date 83, Tsichritzis 77, Ullman 82].

2.9.1 Assessment of Classical Data Models

To assess these models, we follow a framework based on the requirements discussed in 2.8.

1) Object orientation

All three classical models lack the object orientation, i.e., these models model the objects in the real-world through the names or values (*keys*) associated with those objects.

2) Semantic expressiveness

All three models fail to capture much of the semantics associated with the data. The problem with the relational model is that it uses a single mechanism (the relation) to model a collection of entities, to express an association among entities and so forth. This semantic overloading of the relation makes it difficult for a user to determine the meaning and purpose of a relation, and obscures the meaning of a database as a whole [Schmid 75]. In addition, as relationships between relations are not specified in the schema, there are no mechanisms to guide the user in interpretation of the data. Lack of structure in the relational model also allows meaningless relationships (joins) to be formed. In this respect, the hierarchical and CODASYL models are better than the relational model as they provide for explicit capturing of at least some of the relationships in the form of trees or networks.

In addition, all three models do not provide adequate mechanisms to specify entity types or the type hierarchy. Not all records or tuples in these models represent objects [Kent 78, Kent 79] nor is it possible to capture the different roles played by an object. For example, a record in CODASYL model is an instance of one record type and a tuple in the relational model belongs to a single relation.

3) **Ease of design**

It is not easy to arrive at a schema for a database using the constructs of any of these models. Since a data structure in these models (record or relation) may not always correspond to a single object, these models require complex normalisation procedures [Codd 72] to be carried out in order to ensure that there are no undesirable side-effects of update, so-called 'update anomalies'. The result is that the database schemas are often difficult to design. In addition, severe inherent structural constraints in both hierarchical and CODASYL data models limit the data modelling capability and may force unnatural organisation of data.

Another shortcoming of all the three models is that they force modelling apparently similar type of inter-object associations in a number of different ways [Kent 78, Kent 79]. For example, a one-to-one association may be represented as fields of a record representing one object or the other but not both. A one-to-many association may be represented as a repeating group, a hierarchy, a CODASYL set or a relation, depending on the model. A many-to-many association may be represented by creating a separate intersection record or relation. These multiple ways often cause problems for the users both while accessing and while updating, requiring them to write different DML code in each case.

4) **Neutrality**

All three models force a single global perspective of data, i.e., the data is arranged in a rigid, inflexible structure and it is not easy to mould a fact into a variety of semantic interpretations.

5) **Operations**

All three models provide data languages. However, the problem with the data languages for the hierarchical and CODASYL models is that they tend to be oriented towards record-at-a-time access and explicit navigation from one record to another. This complicates the data selection requests considerably. The relational model is better in this respect as most of the data selection and modification languages developed for the relational systems are non-procedural. (It should, however, be pointed out that it is possible to provide navigation-free data languages for hierarchical and CODASYL models as well, as was demonstrated by the ASTRID system [Gray 81] for the CODASYL model).

A major problem with all the three models as regards update is that the units of update in these models (record or relation) do not constitute atomic semantic units. As a result, these

models require extensive additional constraints to maintain semantic integrity of the database. For example, an explicit constraint is necessary to ensure that an object identified by a name in a relation or record really exists in the database, i.e., the concept of *referential integrity* [Codd 79, Date 81].

6) Facilities to specify constraints

Most hierarchical data models do not provide any explicit constraints. An indirect explicit constraint mechanism is provided in IMS via the definition of logical relationships [IBM 75]. By means of logical relationships it is possible to assure the consistency of certain data by constraining the data to be identical in two different definitional trees. The CODASYL model does propose facilities for specifying constraints via database procedures, but no system has implemented these. Relational systems provide facilities to specify the constraints using the constructs of associated query languages [Eswaran 75, Chamberlin 76, Stonebraker 75].

7) Facilities to specify derived data

The hierarchical model provides virtually no facilities to incorporate derived data in the schema. The CODASYL model provides for incorporation of derived data through the use of database procedures, but again no system incorporates these. The relational model also provides facilities to define derived relations (called 'views' in both System R [Astrahan 76] and INGRES [Stonebraker 76]) but these are rather limited in that such derived data corresponds only to inferred facts and not to computed facts. There are relations, logically derivable from others, which cannot be expressed by means of the relational algebra. For example, relations expressed as the result of user operations such as arithmetic operations, statistical operations etc.

8) Freedom from physical considerations

Both the hierarchical and CODASYL models are heavily biased towards representational aspects of the data rather than the information content of the data they manage. Inter-record relationships in these models actually correspond to physical access paths. Because of this, the information that can be retrieved from these databases is tightly constrained by the pre-defined relationships. This is not the case with the relational model as it is based on the concept of mathematical relations.

9) **Ease of evolution**

Because of the presence of a large number of physical constructs, accommodating changes in the logical structure is a difficult task for databases designed using both hierarchical and CODASYL data models. Most relational systems provide facilities to add or delete relations as well as add or drop attributes.

10) **Complexity**

Both the hierarchical and CODASYL models incorporate many complex features, which make them difficult to understand. Also, as the data structures in these models are biased towards a limited class of applications, specifying new applications requires a high degree of computer expertise. Though the relational model is very simple to understand, the fact that inter-relationships among objects are not captured in the schema but are formulated at execution time implies that the users are required to perform a number of explicit relation joins at run-time. Requiring the users to perform such unnecessary joins complicates their task considerably.

2.10 Conclusions

In this chapter, we have presented a detailed discussion on the relationship of the database to the world it represents (Conceptual Modelling) and the tools to design the logical structure of the database (Data Models). We identified a set of concepts that are useful to represent the information and a set of criteria to assess the suitability of data models. Against this set of criteria, we examined the three classical (hierarchical, CODASYL, and relational) data models.

Chapter Three

Semantic Data Models

In this chapter, we give a brief description of twenty-one data model proposals that have appeared in the literature over past ten years. There are possibly many other data model proposals which are worthy of attention, but the lack of space prevents their inclusion here. There is considerable variation among these models, but all of them have one common goal: to reflect more of the semantics associated with the application. Hence, we call all of them *semantic data models*.

For the purpose of this discussion, we classify these models into two broad categories: those that were proposed as extensions to classical models and those that are significantly different from the classical models. We then examine these models against the set of criteria established in section 2.8 and explain why we consider the functional data models to be most attractive for the purpose of conceptual data modelling. We conclude this chapter with the assessment of functional data models that are proposed in the literature.

It is to be noted that the discussion on these models is neither exhaustive nor complete. References given at the end should be consulted for further details.

3.1 Extensions of Classical Data Models

We include the following data models under this category:

(1) Role Model
(2) Basic Semantic Model
(3) Entity-Relationship Model
(4) Structural Model
(5) Abstraction and Generalisation
(6) Extended Relational Model

3.1.1 Role Model

The role model [Bachman 77] has been developed as an extension of the network data model. This model introduces the concept of role-segment type to model the different roles a real-world entity can play, e.g., persons can play the role of customers, employees etc. Within a data description, constructed according to the role model, the concept of a record description is augmented by the concept of a role-segment description such that a record occurrence is a vehicle for one or more role-segment occurrences, each of a different role-segment type.

The set declarations used within the role model represent a change from those used within the network model. As in the network model, set relationships may be established with owner and member declarations with the constraint that only one role-segment description may be declared as the owner of a set description and only one role-segment description may be declared as the member of a set description. Unlike the network model, the role model explicitly recognises that there may be two or more record occurrences concerning the same entity and possibly even denoting the same role.

3.1.2 Basic Semantic Model

In this model, proposed by Schmid and Swenson [Schmid 75], the real world is considered as consisting of *objects* and *associations*. An object can either be *independent* or *dependent*; the difference is that a dependent object must be "existence-dependent" on some independent object, whereas independent objects exist in their own right. For example, in a personnel database, an *employee* might be independent object, whereas an *employee_child* might be dependent object (an *employee_child* can exist in the database only if the corresponding *employee* also exists in the database.) Both dependent and independent objects can have properties (*characteristics*). An association is a relationship between independent objects.

Both objects and associations are represented as relations. In fact, relations in this model are classified into five different types, according to the type of information they represent. Associated with this categorisation is a set of integrity rules; for example, an association can be created only if all associated objects already exist. Conversely, an object cannot be deleted if it currently participates in any associations.

3.1.3 Entity-Relationship Model

The details of this model, proposed by Chen [Chen 76], are almost identical to those of the Basic Semantic Model, except in terminology. Chen uses the terms *regular entity*, *weak entity*, and *relationship* in place of *independent object*, *dependent object* and *association*, respectively. Again, both objects and associations are represented by suitably classified relations. Like Schmid and Swenson, Chen defines a set of integrity rules; one difference of detail is that deletion of an entity should cascade to deletion of corresponding relationships, instead of prohibiting the original deletion if any such relationship exists.

3.1.4 Structural model

In this model, proposed by Weiderhold and El-Masri [Wiederhold 79], entities and attributes of entities are again represented by relations. Relations are specialised as *entity relations* if they define a set of independent objects, as *lexicons* if they represent one-to-one correspondence between names, or as *associations* if they represent many-to-many relationships among objects. This model also specifies rules for the enforcement of integrity during the insertion and deletion of tuples.

3.1.5 Abstraction and Generalisation

The proposals of this model, proposed by Smith and Smith [Smith 77], are also based on representing each object by a relation. Here, inter-object relationships are also considered as objects in their own right. A type/subtype notion is used to classify objects. Detailed discussion of this model is not provided here as the Semantic Hierarchy Model by the same authors, a direct descendant of these proposals, is discussed in detail later.

3.1.6 Extended Relational Model

Extended Relational Model, RM/T, proposed by Codd [Codd 79], represents a synthesis of many of the extensions to the basic relational model proposed earlier by other researchers. The basic assumption underlying RM/T, like all other extensions, is that the real world can be modelled in terms of *entities*. However, RM/T also has the concept of *E-attributes* for the unique designation of entities. An E-attribute uniquely identifies an entity (as opposed to a tuple) within the entire database. Relations can still have user-controlled and user-defined keys, but this is no longer required. E-attributes are created and deleted by the system as

a result of user operations on the database and can be used in operations such as *joins*. However, they are not controlled or seen by the user. E-attributes take their values from a special domain (*E-domain*).

Like other extensions, RM/T also classifies relations according to what they represent. Basically, relations can represent both entity types and relationship types. Entity types are represented by *E-relations* and *P-relations*, both of which are specialised forms of the general n-ary relation. An E-relation contains a single column that specifies the surrogate for every instance of that type. A P-relation represents attributes of an entity type by associating entity surrogate values with property values.

A variety of relationships can exist among entities - for example, two or more entities may be linked together in an association, or a given entity type may be a *subtype* of some other type. A variety of integrity constraints that are implied by the existence of such relationships are also given. A number of high-level operators are provided to facilitate the manipulation of the various RM/T objects (E-relations, P-relations and so on).

3.2 Other Data Models

The significant contributions in this category are:

(1) Data Semantics
(2) Data Independent Accessing Model DIAM II
(3) Binary Relational Models
(4) Object-Role Model
(5) Functional Data Models
(6) Semantic Data Model
(7) Semantic Hierarchy Model
(8) Semantic Network Models

3.2.1 Data Semantics

This model, proposed by Abrial [Abrial 74], organises various objects in a database into different *categories*. Binary relations define atomic links between pairs of objects belonging to certain categories. The two directions of a binary relation are named uniquely. Each name, called an *access function*, corresponds to a binary relation followed in one direction. (It is to be noted that access functions in this model are not really functions in the mathematical sense of the word, since their application can yield more than one value.) Relationship types between more than two categories are represented by generating new categories.

Abrial's model provides for logical access to data by means of *elementary operations* on

access functions via programs. It also provides operations for manipulating objects in a category (e.g., introducing a new object) and for manipulating connections between objects (e.g., relating an object in one category to an object in another category). In addition, one can define operations that take a more general form and affect many objects in many categories according to a program.

3.2.2 Data Independent Accessing Model

Data Independent Accessing Model (DIAM), proposed by Senko [Senko 75], spans five different levels: the *end-user level*, the *information level*, the *string level*, the *encoding level*, and the *physical device level*. The end-user level, corresponding to the ANSI/SPARC external level, consists of *entity*, *property*, *fact*, *entity set* and *enterprise*. At the information level, names of things are dealt with, hence the terms used are *attribute name*, *attribute value*, *identifier name*, *identifier value* and *fact representation*.

A fact representation is a pair of attribute names, also called an *association pair*, linking identifiers. The association pair is symmetric, i.e., one attribute name uses one identifier as the subject and the other identifier as the value, while the other attribute name does just the opposite. N-ary and many-to many associations are treated by creating artificial entities. The language associated with DIAM II is called FORAL.

3.2.3 Binary Relational Models

The binary relational models, proposed by Brachhi et al. [Brachhi 76], Sharman [Sharman 77], and Munz [Munz 78] use only two kinds of representational concepts: entities and labelled binary relations. This implies that relationships of a higher order than two will be considered as entities. Properties of entities are modelled as binary relations between sets of entities and property values. The domains of these binary relations are all fundamental domains, i.e., they are not themselves binary relations. This means that nested binary relations are not allowed.

3.2.4 Object-Role model

This model, proposed by [Falkenberg 76], models facts from a particular universe of discourse by means of *objects*, *roles* and *associations*. Objects are atomic, discrete elements in nature; the only information represented by them is their existence. Facts concerning an object correspond to its association with one or more objects. An object performs a role in every association of which it is a part. Thus associations, which are n-ary in general, are composed of object-role pairs. An association may be treated as an object in its own right and may perform a role in another association. The latter is termed *a nested association*.

Objects are pooled into *object types* such that objects under one type have at least one role in common. Associations are also classified into *association types* which refer to all associations with identical object-role pairs. Objects and roles may be provided with *significations*. An object type named *person* may be signified by the person's first name. A number of significations may be provided, e.g., first name, last name, date of birth for the object type *person* to make the signification unique. Semantic rules can also be specified as a means of providing additional constraints for consistency and integrity among data instances.

3.2.5 Functional Data Models

The idea of viewing an information system as a collection of functions was introduced by Folinus *et. al.* [Folinus 74]. There are five different functional data model proposals that have appeared in the literature [Kerschberg 75, Buneman 79, Housel 79, Shipman 81, Katz 83]. All five data model proposals are based on the fundamental concept of function to model relationships among objects in the real world. All five models also support simple data manipulation languages using the function and set operators, and one of them, FQL [Buneman 79], goes even further by integrating data manipulation and general purpose computation in one language.

3.2.5.1 Functional Database Model

In this model, proposed by Kerschberg and Pacheco [Kerschberg 75], a database schema is considered as a graph whose vertices represent sets and whose arcs are total functions. The sets may correspond to either *entity sets* that model real-world entities or *abstract sets* representing associations among entity sets.

A set is described by its name and the functions defined on it. These functions have as

ranges other sets, including a distinguished set of data values. In this model, a *key* for a set S consists of a non-empty collection of functions of S such that the mapping from S to the Cartesian product of the function's range sets is one-to-one.

Logical access to the complete functional data model is achieved by means of a high-level declarative language. Query specification is done by walking (or navigating) through the graph.

3.2.5.2 Functional Dependency Model

In this model, proposed by Housel et.al [Housel 79], a database is viewed as consisting of sets of values with functions between them. Sets can be either *simple* or *tuple* sets. Simple sets correspond to integers, strings etc. Tuple sets correspond to many-to-many relationships among sets. Because the domain and range of functions may be tuple sets, relationships between relationships is supported. This model also supports the concept of "generalisation" [Smith 77] by treating one value set as the subset of another.

The model provides operators for retrieval, update, insertion and deletion of occurrences in functions and sets and iterative (FOR loops) and conditional (IF-THEN-ELSE) control constructs. There are three retrieval primitives: RANGE, which given an occurrence of a function's domain, returns the corresponding value of its range; DOMAIN, which given an occurrence of a function's range, returns the ordered set (possibly null) of domain values; and ENTRY, which given a set name returns the set of values from the named set. There are also four data manipulation primitives: CREATE, which adds a value to a set; LINK, which adds a pair of values to the domain and range of a function; UNLINK, which performs the inverse; and DELETE, which deletes a value from a set.

3.2.5.3 Functional Data Model (Buneman and Frankel)

Buneman and Frankel have proposed a functional data model built into the functional query language (FQL) [Buneman 79, Buneman 82]. In this work, a database is viewed as a collection of functions over various data types. The model also provides five operators on functions: *compose*, *tuple*, *extend*, *restrict*, and *generate* which are used to combine functions to form new functions. FQL has built-in functions for arithmetic and boolean operations for use in query formulation. Finally, the notation used in FQL is derived from the functional programming notation advocated by Backus [Backus 78].

3.2.5.4 Functional Data Model (Shipman)

In this model, proposed by Shipman [Shipman 81], the database is modelled as a set of functions mapping entities to entities. Functions in this model can have zero, one, or more arguments. Functions with no arguments are used to define entity types, and functions with arguments are used to define attributes of and relationships among entities. Functions can also be specified as *single-valued* or *multi-valued* (i.e., yielding a set as a result). This is useful to capture one of the most common type of real-world constraints as part of the structure itself.

An important concept in this model is the distinction between an entity and its external names. In addition, entity types in this model are arranged in a type hierarchy with automatic inheritance of attributes and relationships (i.e., functions) from a supertype to all of its subtypes.

This data model proposal also incorporates a high level, integrated data definition and data manipulation language, DAPLEX. In this language, function applications are used to express queries. In addition, the language includes a FOR EACH statement to iterate through sets, and a set of update statements. There is also a special operation in DAPLEX, DEFINE, for incorporating user-defined functions into the schema. Update operations on derived functions may be specified by the user.

A detailed discussion of this model can be found in the next chapter.

3.2.5.5 Functional Data Model (Katz and Wong)

This model, proposed by Katz and Wong [Wong 79, Katz 83], a database is viewed as *object sets* and *functions* between them. Object sets are either *value sets* or *entity sets*. The primary difference is that value sets never appear in the domain of a function. For example, the set of employees is an entity set, while the set of employee names is a value set.

Multi-valued (many-to-many) relationships are represented by confluent hierarchies, that is, by an explicit "relationship" set and functions to map it into the participating entity sets.

3.2.6 Semantic Data Model

Semantic Data Model (SDM), proposed by Hammer and McLeod [Hammer 78, Hammer 81], views a database as a collection of disjoint *classes* of objects. SDM distinguishes between non-atomic abstract objects (called *entities*) and identifiers (*names*). Database entities and classes have *attributes* that describe their characteristics and relate them to other database entities. Each attribute has a *value* which is either an entity in the database (a member of some class) or a collection of such entities. The value of an attribute is selected from its underlying *value class*, which contains the permissible values of the attribute.

SDM also supports a notion of IS-A hierarchy among objects and a notion of grouping types: a grouping type is formed by considering instances of a type to be subtypes, rather than individual objects. SDM also provides an extensive predicate language for specifying derived information. This mechanism allows a significant amount of data that would normally have to be derived at access time to be made a permanent part of the schema.

In SDM, subtypes may be derived from properties of parent types or by stating that a subtype is an arbitrary, user-chosen group of values. Extensions of subtypes can also overlap. Attributes may be derived in terms of other attributes, e.g., attributes may be *inverses* of each other or the value of an attribute may be declared to be directly assignable by the user.

In addition, classes can be constrained to have members with non-duplicate (unique) attribute values. The attributes of any class can be constrained to be single- or multi-valued or to have a non-null value or have values which are not changeable or are exhaustive of its class. It is also possible to place a constraint on the size of a multi-valued attribute or have non-overlapping values for different entities of a multi-valued attribute.

Thus SDM provides a rich set of constructs to capture the semantics associated with the data. However, the complexity of the model makes it difficult to see how it can be used effectively.

3.2.7 Semantic Hierarchy Model

In this model, proposed by Smith and Smith [Smith 80a], real world information is treated as consisting of relative structures among all the objects of interest. Three classes of abstractions are used to determine these structures *viz.*, *classification*, *generalisation* and *aggregation*. Classification collects instances to form a new type. Generalisation forms a new

type by merging existing types (called *categories*). These categories can be considered as subtypes. Aggregation forms an object as a relationship among other objects (called as *components*). In general, each object is a type, an aggregate object and a generic object. An object will be called *primitive* when it has no instances, components or categories of interest. When classification, generalisation and aggregation are repeatedly applied to objects, hierarchies of objects are formed.

To specify such structures a simple type declaration language is used. It lists for each type in the structure its component types and categories. The attributes of a subtype are inherited from its parent types, and the subtype may introduce additional attributes as well. Subtypes of a given type are assumed to be non-overlapping; thus, subtypes partition the instances of the parent type.

The primitive operations of *create*, *destroy* and *modify* are provided to specify the behaviour of objects. A first order predicate language is provided for naming individuals in terms of their structure - their attributes, categories and types. Using these primitive operations, naming capabilities and simple control structures such as the if-then-else construct of standard programming languages, functions and procedures can be constructed that describe the behaviour of the enterprise being modelled.

3.2.8 Semantic Network Data Models

Semantic network data models have been mainly developed by people working in various branches of artificial intelligence. The basic structures of these data models consist of nodes and arcs forming a network much as in a binary or network data model. These data models distinguish between a type and a token and introduce the idea of hierarchy of types. Hierarchy inheritance deals not only with the inheritance of attributes and their values, but also with the inheritance of permitted relationship types among types.

We discuss here the following two semantic network data models:

3.2.8.1 TAXIS

TAXIS is a programming language, proposed by Mylopoulos *et. al.* [Mylopoulos 80], based on the concept of semantic network for data and procedure modelling. Here, a database is modelled in terms of three types of objects: *tokens* which represent constants; *classes* which describe collections of tokens; and *metaclasses* which describe collections of classes. Classes

and tokens have properties through which they can be related to other classes and tokens. Classes and metaclasses are defined by specifying their name and their properties. TAXIS provides for organisation of the classes and metaclasses into a hierarchy by defining IS-A (generalisation) relationship over classes and metaclasses.

TAXIS provides three primitive database state transformation functions, *insert-object*, *remove-object*, and *update-object* and the *procedural attachment* which associate directly with each class the database transformation functions that affect its extension. TAXIS also provides *transactions* which are treated as classes except that their body is given in terms of zero or more prerequisite, action and result properties. *Exceptions* are raised when a prerequisite or result expression evaluates to a value other than true. Exception classes are defined and organised into an IS-A hierarchy, like all other classes.

3.2.8.2 Semantic Binary Relationship Model

Semantic Binary Relationship Model (SBRM), proposed by Azmoodeh and Lavington [Azmoodeh 82], is a type of semantic network with the nodes representing entities and the arcs representing relationships between entities. The entities are categorized broadly into *information*, *meta information*, and *meta meta information* according to their levels of abstraction.

The information consists of entities and binary relationships between them. The entities may be grouped together into different classes. Relationships may be categorised in the same manner. Meta information consists of classes of similar entities or relationships at information level. The entities at meta meta level denote these classes at meta level. The meta meta information and its semantics is static and built into the system and invisible to users. Users only manipulate information and meta-information.

The class membership is achieved via the system defined relationship ISA. (Note that this use of ISA is different from that usual in semantic networks). The classes may be overlapping. The interdependency between the classes is represented by means of a *subclass* operation. The IS SUB relationship is used to express the subclass operation.

3.3 Assessment of Semantic Data Models

In general, most of the semantic data models describe the real world in terms of units that are close to the concept of an entity and a few of them carry it through to make a distinction between an entity and its names, i.e., what we call as object orientation. Most of the models organise the modelling units into sets or types and a few of them carry it through to provide hierarchical type structure. Some models make a distinction between attributes and relationships while some models treat both as relationships. Some models make a distinction between entities and relationships while some models force all relationships to be treated as entities in their own right. Some models limit themselves to binary functional (one-to-one and many-to-one) relationships forcing creation of "excess" entity sets for modelling n-ary or non-functional (many-to-many) relationships. Most of the models, however, ignore aspects such as the provision of a consistent set of operations, facilities to capture rules etc.

An assessment of the semantic data models discussed in this chapter was carried out using the framework we established in section 2.8. The resulting analysis is shown in Table 3-1. In this table, the entries like *high*, *medium* and *low* reflect the author's judgement based on the information that can be gathered from literature. Where the author was unable to form an opinion from the available information, corresponding entries are shown with a question mark. It is to be noted that this analysis is necessarily approximate because of the difficulty in understanding the varied vocabularies and the considerable ambiguity in the way concepts in one model relate to concepts in another.

Table 3-1: Assessment of Semantic Data Models

	Role Model	Basic Semantic Model	E - R Model	Structural Model	Extended Relational Model
Object Orientation	No	No	No	No	Yes
Semantic Expressiveness	Medium	Medium	Medium	Medium	High
Ease of Design	Low	Medium	Medium	Medium	Low
Neutrality	No	No	No	No	Yes
Operations	No	No	No	No	Yes
Facilities to specify constraints	?	?	?	?	?
Facilities to specify derived data	?	?	?	?	?
Freedom from physical considerations	No	Yes	Yes	Yes	Yes
Ease of Evolution	Low	Medium	Medium	Medium	Medium
Complexity	High	Medium	Medium	Medium	High

Table 3-1, continued

	Data Seman-tics	DIAM	Binary Relat-ional Models	0 - R Model	S H M
Object Orientation	Yes	No	No	Yes	Yes
Semantic Express'ness	Medium	Medium	Medium	Medium	High
Ease of Design	High	High	High	Medium	Low
Neutrality	No	No	No	No	Yes
Operations	Yes	Yes	Yes	Yes	Yes
Facilities to specify constraints	Yes	?	?	?	?
Facilities to specify derived data	Yes	?	?	?	?
Freedom from physical considerations	Yes	Yes	Yes	Yes	Yes
Ease of Evolution	Medium	Medium	Medium	Medium	Medium
Complexity	Low	Low	Low	Low	Medium

Table 3-1, continued

	FDM (Kersc-hberg)	FDM (Housel)	FDM (Katz & Wong)	FDM (Bune-man)	FDM (Ship-man)
Object Orientation	No	No	Yes	Yes	Yes
Semantic Express'ness	Medium	Medium	Medium	Medium	High
Ease of Design	Medium	Medium	Medium	Medium	High
Neutrality	No	No	No	No	Yes
Operations	No	Yes	?	Yes	Yes
Facilities to specify constraints	?	?	?	?	Yes
Facilities to specify derived data	?	?	?	Yes	Yes
Freedom from physical considerations	Yes	Yes	Yes	Yes	Yes
Ease of Evolution	Medium	Medium	Medium	Medium	High
Complexity	Medium	Medium	Medium	Medium	Low

Table 3-1, continued

	S D M	TAXIS	SBRM
Object Orientation	Yes	Yes	Yes
Semantic Express'ness	High	High	High
Ease of Design	Medium	Medium	Medium
Neutrality	Yes	Yes	Yes
Operations	?	Yes	?
Facilities to specify constraints	?	Yes	?
Facilities to specify derived data	Yes	Yes	?
Freedom from physical considerations	Yes	Yes	Yes
Ease of Evolution	Medium	Medium	Medium
Complexity	High	High	High

Among all the models discussed above, we consider the functional data models to be most attractive for the purpose of conceptual data modelling. The following six reasons support this claim:

1) Functional data models provide a semantically rich modelling environment. Shipman's proposals show how this can be achieved. The most important feature of these models is their ability to remove the sharp distinction between data and programs (derived data). All functions irrespective of whether they are stored functions like *name(person)* or computable functions like *sine* and *max* have equal rights. Any function may be composed with any other function of conformable type using the same syntax. Because of this, the information that can be extracted from such a system corresponds not only to that which can be obtained from simple access to stored data, but also to that which involves complex traversals of data structure, and/or computations. Thus, the functional approach makes it possible to "equate data in the system with what can be extracted, rather than with what is physically stored" [Kent 78]. The advantages of unifying data and programs (derived data) have already been discussed in section 2.7.

2) Functional data models provide "simple" data manipulation languages. The function application provides a natural mode of expressing user queries. As Folinus et al. observe, "the requests for answers, whether made to a processing program or to a stored database, are essentially requests for a value of a function, given argument values". Because of this, query formulations in these models are likely to be closer to the natural language form.

3) Functional data models provide data languages in which data manipulation facilities are neatly integrated with the general-purpose computation facilities. FQL [Buneman 79] shows how this can be achieved. Existing data manipulation languages, both procedural and non-procedural, do not provide a complete programming environment. To overcome this deficiency, these languages are invariably coupled to existing procedural programming languages. Such coupling is usually provided by either defining special subroutines to execute database functions or by embedding database constructs into an existing language. A preprocessor translates these constructs into run-time calls on a database system. Application programs consist of statements in the host programming language intermixed with statements in the query language to access the database system when required. Problems with such approaches include the difficulty of performing type checking across language interfaces, the trade-off between interpretation and compilation, and the unattractive nature of combining non-procedural query languages in procedural programming languages. Typical programming languages and typical query languages differ from each other too much in their description of

data structures and this embedding method does not produce a natural programming interface [Atkinson 78, Pirotte 80].

Of late, a number of efforts have been made to extend programming languages with database notions, e.g., PASCAL-R [Schmidt 77], RIGEL [Rowe 79], THESEUS [Shopiro 79], PLAIN [Wasserman 81] etc. The problem with these integrated languages is that they attempt to extend essentially procedural languages like PASCAL [Wirth 71] with non-procedural query language constructs. Such languages offer a variety of programming styles which are not strictly necessary for expressive powers: they range from a navigational style where database relations are examined one tuple at a time in an order fixed by physical storage to a style of non-procedural programming similar to that of typical query languages. Thus the resulting languages have a redundant and hybrid character which does not appear fully justified nor obviously necessary [Pirotte 80].

A major reason for the unsatisfactory nature of these attempts is the lack of a common concept between the data models and the programming languages. Functional data models are well suited to overcome this shortcoming as the concept of function used to model objects and inter-object associations in these models is also the basis of many programming languages like LISP [McCarthy 62], KRC [Turner 82], ML [Gordon 79], HOPE [Burstall 80] etc.. Hence, the data manipulation languages provided for these models can be expected to merge comfortably with the functional programming languages.

4) Functional data models can support all the three major (hierarchical, CODASYL, and relational) data models [Sibley 77, Shipman 81, Smith 80b, Gray 83]. This means that functional data models can be used to provide a global schema through which one can take a unified view of heterogeneous distributed databases some of which are CODASYL and some are relational. This is being tried in the heterogeneous distributed database system, MULTIBASE [Smith 80b]

5) By exploiting the well established mathematical theory of functions [MacLane 67], functional data models can provide a solid theoretical foundations for databases.

6) Since functions can be considered as logical access paths of a database, there is a strong possibility of providing efficient implementations of functional data models. In fact, Katz and Wong use the functional data model to design storage structures for a CODASYL DBMS to provide the desired access characteristics [Katz 83]. The advances in data flow computers [Arvind 78] is another factor favouring the functional data models. By exploiting parallelism,

these machines promise to provide much higher performance. The great advances in semiconductor device technology can be harnessed much more effectively in these architectures. Several groups engaged in data flow research have developed high-level languages suitable for data flow programming. The primary characteristic of these languages is that they are functional. It is easy to see that the functional data models will be in a better position to exploit the advances in this field.

3.4 Assessment of Functional Data Models

As discussed earlier, there are five different proposals of the functional data model that have appeared in the literature so far. Table 3-1 summarises the assessment of these models based on the set of criteria established in section 2.8. Here, we expand on this assessment.

The functional database model of Kerschberg et al. and the functional dependency model of Housel fail to meet our first criteria, i.e., the object orientation. They model the objects in the real world by their names and inter-object relationships in terms of attribute values. Regarding this, both these models suffer from the same disadvantages as the conventional name-based models like the CODASYL or the relational model.

Though the functional data model of Buneman and Frankel and the functional data model of Katz and Wong incorporate the object orientation, semantically they are insufficiently expressive. They classify entities into types (sets) but do not accommodate subtype-supertype hierarchy. In addition, they limit themselves to binary functions only. This means that to model many-to-many and non-binary relationships, they are forced to adopt the *excess entity* approach similar to that used in the CODASYL model. That is, they introduce a "relationship" entity and provide functions to map it into the participating entities. For example, the many-to-many relationship between students and courses in these models is represented by an *enrolment* entity and the two single-valued functions *student-of(enrolment)* and *course-of(enrolment)*. Each *enrolment* entity represents a single relationship between a student and a course.

Similarly, n-ary relationships in these models are handled by reducing them to binary form by defining a new artificial entity type corresponding to the relationship and then defining n binary relationships each corresponding to a relationship between the new entity type and one of the "components" of the original relationship. For example, the n-ary relationship

between a buyer, a seller, and an item is represented by a *sale* entity and the three single-valued functions *buyer-of(sale)*, *seller-of(sale)*, and *item-of(sale)*. The problems with the excess entity approach are that it obscures the actual relationship, and the artificial entities so created have no correspondence with the real-world objects. This also forces one to adopt different techniques to model different types of relationships. Regarding this, both these models suffer from the same disadvantage as the CODASYL model.

It is the Shipman's model which meets most of our criteria. It has the abstract object orientation firmly built into it. In addition, it classifies entities into entity types and incorporates subtype-supertype relationships among entity types. By admitting functions with multiple arguments and set-valued results, Shipman's model captures all types of inter-object relationships uniformly. Because of this, mandatory creation of potentially unnatural entity types is avoided. Shipman's model is also unique in exploiting the main advantage of functional approach, i.e., the ability to treat data and programs alike, by means of derived functions. Another important factor that is in favour of Shipman's model is the simplicity of the associated data language. Detailed assessment of this model appears in the next chapter.

3.5 Conclusions

Inspite of the many benefits of functional data models listed above, the literature contains few reports on their implementation and use. In an attempt to fully understand the advantages and problems of functional database management systems, this research concentrated on the construction of a full-scale functional database management system. The result of this is the Extended Functional Data Model (EFDM) system, implemented on a VAX 11/780 hardware configuration at Edinburgh.

The data model underlying EFDM is very close to the proposals of Shipman, though we have made some modifications and extensions. We call this *Extended Functional Data Model* to distinguish it from the "basic" functional data models proposed by others, which, as we discussed above, are semantically less expressive. Chapter 4 describes Shipman's proposals in detail and Chapter 5 elaborates on the modifications and extensions of these proposals carried out in EFDM.

Chapter Four

Functional Data Model

In this chapter, we introduce the basic concepts of the functional data model as proposed by Shipman [Shipman 81]. We illustrate these concepts using the constructs of the associated data language DAPLEX. The syntax of this language is reproduced in Appendix A.

4.1 Structures

Functional data model (FDM) models the real-world information as sets of entities and of functions mapping entities to entities. An entity is some form of token identifying a unique object in the database and usually representing a unique object in the real world [Gray 83]. For example, FDM models a student in the real world by a unique *student* entity in the database. Sets of entities with some common information are classified as *entity types* (or simply *types*). Entity types are arranged in a type hierarchy, so that they are all subtypes of the type **entity**, the system-provided type of all entities. Functions map a given entity into a set of target entities. For example, a function *courseof* defined to map a particular *student* entity to a set of *course* entities can be used to model the set of courses a student is taking.

The *intension* of a functional database is specified by a *functional schema* which consists of one or more *function definitions*. A function definition corresponds to a function name followed by a list of arguments enclosed in brackets followed by a single or double arrow followed by the result. That is, a function f_i is denoted as

$$f_i(a_{i1}, a_{i2}, \ldots a_{im}) \rightarrow r_i$$

or

$$f_i(a_{i1}, a_{i2}, \ldots a_{im}) \rightarrow\rightarrow r_i$$

where the a_{ij}s, not necessarily distinct, refer to argument entities of type a_{ij} and r_i refers to result entities of type r_i. A single arrow -> implies the function is single-valued and a double

arrow ->> implies it is multi-valued. Single-valued functions return a single result entity of type r_i on each application and multi-valued functions return a set of entities, all of type r_i, on each application.

The functional schema for a database to represent students and their tuition arrangements is shown in figure 4-1. The **declare** and **define** in this figure correspond to the data definition statements of DAPLEX language.

```
declare person()                    ->> entity
declare student()                   ->> person
declare staff()                     ->> person
declare course()                    ->> entity
declare event()                     ->> entity
declare tutorial()                  ->> event
declare lecture()                   ->> event

declare cname(person)               ->  string
declare sname(person)               ->  string
declare sex(person)                 ->  string

declare course(student)             ->> course
declare tutorial(student)           ->  tutorial
declare grade(student,course)       ->  string

declare course(staff)               ->> course
declare room(staff)                 ->  string
declare phone(staff)                ->  integer

declare title(course)               ->  string
declare lecture(course)             ->> lecture

declare day(event)                  ->  string
declare time(event)                 ->  integer
declare site(event)                 ->  string
declare room(event)                 ->  string

declare staff(tutorial)             ->  staff

define staff(course)                ->> inverse of course(staff)
define lecturer(student)            ->> staff(course(student))
define tutor(student)               ->  staff(tutorial(student))
```

Figure 4-1: Functional Schema for a Student Database

4.1.1 Entity Types

Consider the function *person()* shown in figure 4-1. This function evaluates to a set of entities of type **entity**. Since it has no arguments, there is only one possible result set. Shipman uses this fact to say that all members of this set have a distinct type, i.e., it defines the *person* entity type. (Note that *person* is triply overloaded here, i.e., it names the entity type, the set of *person* entities, and the function that produces that set.)

Similarly, the *course()* function defines the *course* entity type. Thus, all zero-argument functions define new entity types.

An important point about this model is that entities represent objects in the real world and not the numbers or other identifiers associated with the objects. Hence, entities by themselves cannot be printed. The only exceptions are those belonging to pre-defined entity types like **integer**, **string** etc. They are special in that, as well as being predefined, they have an established method of lexicographically representing instances of these types. The actual set of such built-in types is analogous to the base types in a programming language, and would be chosen for their utility. Other such useful types are **boolean**, **real**, **date**, and **time**.

In this model, the extensions of different entity types can overlap, i.e., a *student* can simultaneously be a *staff* entity as well. The notion of *role* (See 4.2.2) is used to determine the type of an entity during evaluation.

4.1.2 Functions

Functions with arguments model both the properties (attributes) of the objects and the relationships between various objects. For example, the function *cname(person)* which returns an entity of type **string** for each *person* entity can be considered as modelling a property of *person* entities whereas the function *course(student)* which returns a set of *course* entities for each *student* entity can be considered as modelling the relationship between *student* and *course* entities. Thus, this model makes no distinction between attributes and relationships.

This model allows multi-argument functions also. For example, the function *grade* with two arguments *student* and *course* types models the fact that every *student-course* pair is associated with a **string** entity indicating the grade that the student gets for that course. A difficulty with this function definition is that it specifies the function *grade* as defined for every *student-course* pair, while in fact it should exist only for those courses in which the student is enroled.

Shipman suggests the following definition to overcome this problem:

```
declare grade(student, course(student)) -> string
```

According to Shipman's proposals, all functions are treated as total. Hence, all single-valued functions must be explicitly initialised by the user and all multi-valued functions that are not initialised return the empty set.

4.1.3 Type Hierarchy

Consider the function *person()* again. The set of *person* entities returned by this function is a subset of the set of **entity** entities. Shipman uses this fact to say that the entity type *person* is a *subtype* of the entity type **entity**.

This subtype-supertype relationship can be extended to any level. Also, an entity type can have any number of subtypes. For example, the *person* entity type above has *student* and *staff* entity types as its subtypes.

An important consequence of this hierarchical relationship of types is that an instance of an entity type is also an instance of all its supertypes and a subtype inherits all the functions defined over all of its supertypes. This follows from the fact that a *student* entity is necessarily a *person* entity as well and hence all the functions applicable to *person* type are applicable to *student* entities as well.

4.1.4 Base and Derived functions

A function introduced by **declare** is called a *base function* and is represented by physically storing a table of arguments and results. For example, *cname(person)* function relating persons to their Christian names is represented by a table of *person* and **string** entities. The basic algorithm to evaluate these functions is search of the list of arguments, i.e., a comparison of the given argument with the list of arguments to determine the corresponding result value to be selected.

A function introduced by **define** is called a *derived function* and is represented by an algorithm to compute its result. For example, the *lecturer(student)* function in figure 4-1 is represented by a derivation procedure. Such functions are evaluated by executing the corresponding algorithms. Hence, data for these functions does not, conceptually at least, exist in the stored form in the database. Various mechanisms to define derived functions are discussed in section 4.3.

4.1.5 Function Names

FDM allows function name overloading, i.e., more than one function may have the same name. For example, though the functions *course()*, *course(student)* and *course(staff)* all have the same name, they are distinguished by their *internal names*. The internal name of a function is generated by enclosing in square brackets the external function name and the argument types over which it was originally specified. Thus the internal names of the above three functions are [*course()*], [*course(student)*], and [*course(staff)*] respectively.

4.1.6 Order

Multi-valued functions in this model evaluate to sets of entities in the mathematical sense; that is, the sets are considered unordered and do not contain duplicates. However, order forms a natural part of the real world information. Hence, DAPLEX provides for explicitly associating an order with entity types or multi-valued functions by using **in order** syntax. For example, to order the staff on their rank, the following declaration can be used:

declare staff() ->> person **in order by ascending** rank(staff)

In the above declaration, ordering is maintained by the system. Ordering can also be maintained by users by explicitly specifying the order via update statements. For example, to maintain user defined ordering for musical notes in a melody, the following declaration can be used:

declare notes(melody) ->> notes **in order**

The omission of **by** clause following **order** clause indicates that ordering is to be specified by update statements. More than one **by** clause can be used to indicate primary ordering, secondary ordering, tertiary ordering etc.

4.1.7 Entity Diagram

The functional data model admits a neat and concise graphical representation for the entity types and functions modelling a real-world application. Such a graphical representation is called the *entity diagram*.

Figure 4-2 illustrates such an entity diagram for a schema shown in figure 4-1. In this figure, diamond-shaped enclosures indicate entity types and the arrows indicate functions mapping their argument types into their result types. A solid arrow indicates a base function while a dashed arrow indicates a derived function. A single headed arrow indicates a

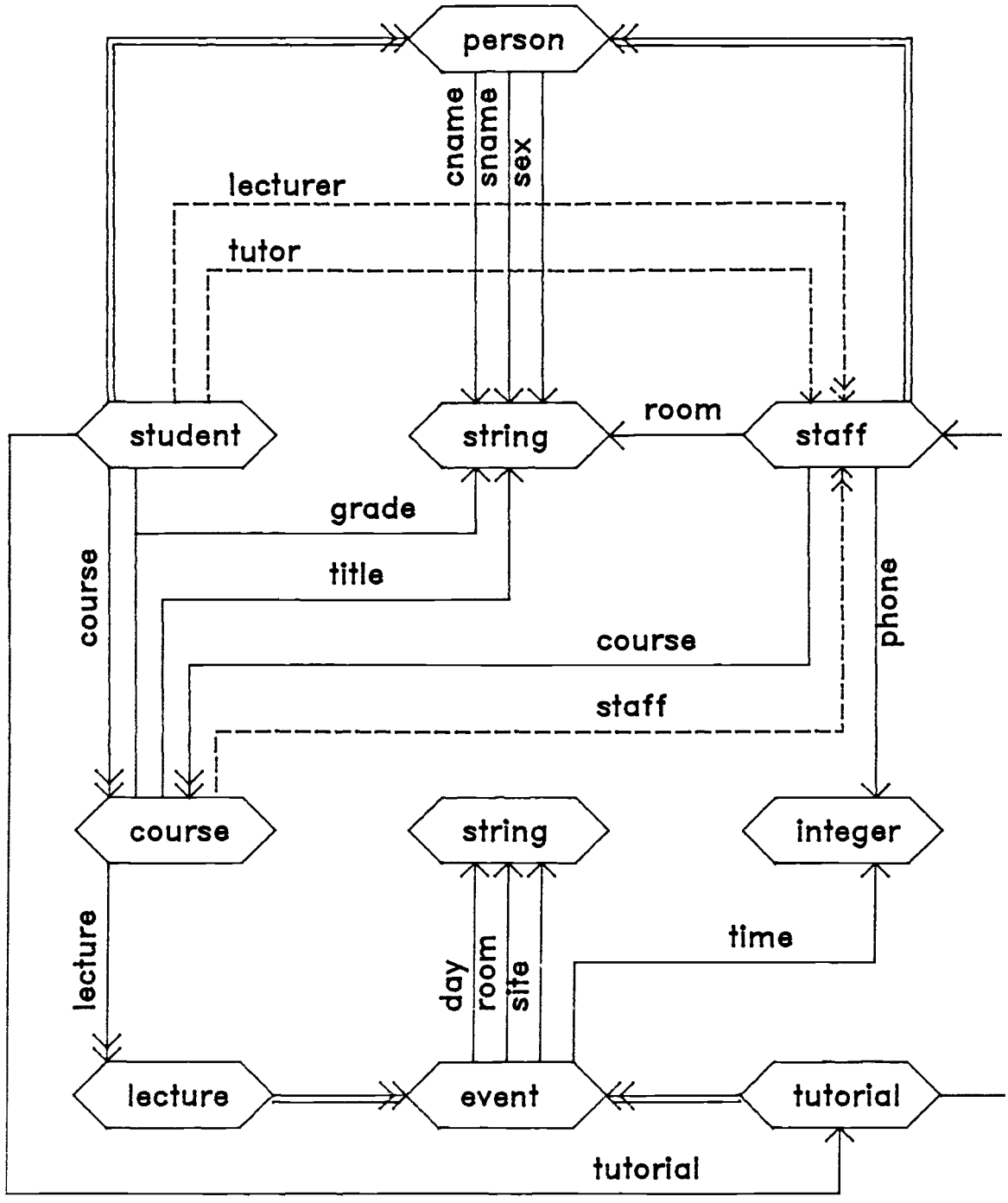

Figure 4–2: Entity Diagram for Student Database

single-valued function while a double headed arrow indicates a multi-valued function. A double-line arrow indicates subtype-supertype relationship. The multi-argument function *grade* is represented by the two arrows from the respective argument types getting merged before reaching the result type.

It can be seen that the entity diagrams closely resemble the semantic nets used for knowledge representation in the artificial intelligence field [Quillian 68].

4.2 Operations

DAPLEX is the integrated data definition and data manipulation language associated with this model. DAPLEX does not concern itself with the general-purpose computation facilities and data input/output operations. Shipman envisages DAPLEX to be embedded in a high level language for these facilities.

4.2.1 Data Definition

A new function can be added to the schema at any time. A **declare** statement is used to add a base function or an entity type while a **define** statement is used to add a derived function or an entity type.

4.2.2 Data Selection and Retrieval

FOR loop statements and expressions are the basic constructs for data selection. For example, a query to find out the names of all persons in the database corresponding to figure 4-1 appears as follows:

```
for each person
print cname(person), sname(person)
```

where the print statement is assumed to be provided by the high-level language in which DAPLEX is embedded. In this example, the FOR loop statement iterates over a set of entities of type *person*, executing its for-body, the print statement, for each member of the set.

There are two forms of FOR loop statement (see Appendix A):

```
for each set imperative |
for singleton imperative
```

where *set* refers to a *set expression* and *singleton* refers to a *singleton expression* and *imperative* refers to either a FOR-loop statement, an update statement, or a **print** statement. While a set expression evaluates to a set of entities, a singleton expression evaluates to a single entity.

Every expression has a *value*, a *role*, and an *order*. The expression value is the set of entities returned by evaluating the expression. The expression order is the ordering associated with these entities. (It is meaningful for set expressions only.) The expression role is the entity type under which these entities are to be interpreted. The role of an expression can always be determined by a static analysis of the expression. An expression's role can be changed without affecting its value or order by means of the **as** operator (see section 4.2.2.4).

4.2.2.1 Set Expressions

Every set expression has associated with it a reference variable. Operators which iterate over the set, such as **for each** in the above syntax, successively bind this variable to the entities in the iteration set. The reference variable typically appears in the body of the iterating operator and references the particular entity being considered in the current iteration. DAPLEX provides for the implicitly declared looping variables. For example, in the above query formulation to find out the names of all persons in the database, the name *person* implicitly declares a reference variable *person*, which actually appears in the body of the **print** statement.

DAPLEX also provides for the explicit specification of reference variables by using the **in** operator. For example, the above query can also be formulated as:

```
for each p in person
print cname(p), sname(p)
```

A set expression can be formed by a name corresponding to an entity type identifier or by a function application resulting in a set-valued result. For example, consider the function application *course(student)* where *student* is a variable corresponding to a *student* entity. Here, the result of the expression corresponds to the set of entities returned by applying *course(student)* function to the entity student.

In general, each argument of a function application can be either a set expression or a singleton expression. When the arguments for a function application are set expressions, the

result of the function application is obtained by iteratively applying the function to each tuple of the cross-product of argument sets and taking the union of all results so obtained. For example, consider the expression

```
course(student such that sex(student) = "f")
```

Here, the result is calculated by taking the union of sets of *course* entities returned by applying the *course(student)* function to each member of the argument set.

A set expression can also be formed by the general set former construction. This has the from

set1 **such that** *predicate*

where *set1* is any set expression and the *predicate* following **such that** evaluates to a boolean result. This form of set expression evaluates to those members of *set1* for which *predicate* is true. In evaluating *predicate*, the reference variable of *set1* is bound to the member of *set1* being tested. For example, consider the following query:

Find the surnames of all the students with the Christian name Pat.

```
for each student such that
   cname(student) = "Pat"
print sname(student);
```

Here only those *student* entities for which the predicate following **such that** evaluates to **true** are included in the result of the set expression.

DAPLEX also allows quantification for the formulation of predicates following **such that**. These are of the form

for *quant set predicate*

where the *predicate* evaluates to a boolean result and *quant* is any one of the following quantifiers: **some**, **every**, **no**, **at least**, **at most**, or **exactly** (last three quantifiers must be followed by a *singleton* evaluating to an integer result). Such quantified expressions always evaluate to a boolean value. For example, consider the following example:

Find the Christian name of all the students taking the IS1 course.

```
for each student such that
   for some course(student)
   title(course)="IS1"
print cname(student);
```

In this query, the expression following **such that** evaluates to **true** if at least one *course* entity in *course(student)* set meets the *title(course)*="IS1" test.

It is possible to specify the order for the results of set-valued expressions using **in order by** syntax as described in section 4.1.6. However, when the **by** clause is omitted while specifying order, the ordering is that of existing ordering associated with the evaluated set.

4.2.2.2 Singleton Expressions

A singleton expression is formed by a constant literal, a variable bound to a particular entity, or by a function application resulting in a single-valued result.

A singleton expression is also formed by the **the** operator followed by a set expression. The result of such an expression is the entity returned by the set expression, only if the result set has a cardinality of 1. For example, consider the following query:

Find the titles of courses taught by Hamish Dewar.

```
for the staff such that
    cname(staff) = "Hamish" and
    sname(staff) = "Dewar"
for each course(staff)
print title(course);
```

In the above formulation, **the** ensures that there is only one *staff* entity with *cname* as Hamish and *sname* as Dewar.

4.2.2.3 Aggregate functions

DAPLEX proposals include the aggregate functions such as **count, maximum, minimum, total, average** etc.

count function takes any set-valued expression as the argument and returns the cardinality of that set in integer form. For example, consider the following request:

Find the number of staff in CS department.

This can be expressed as

```
print count(staff such that dept(staff)="CS")
```

maximum and **minimum** functions take any set-valued expression with numeric type as the argument and return the maximum/minimum of the values in the argument set.

total and **average** functions take any multiset valued expression as the argument and return the total or average value of all the values in the argument set. (A *multiset* or *bag* is a set which may contain duplicate elements.) A special operator **over** is provided, which when used with a set-valued expression evaluates to a multiset. For example, consider the following request:

Find the average salary of staff in the CS department.

This query can be formulated as follows:

```
print average(salary(staff) over staff such that
dept(staff)="CS")
```

Here, for each member of the *staff* set, the given expression is evaluated and the resulting value is included in the average independent of whether or not duplicate values are present.

4.2.2.4 Specifying the Expression Role

The role of an expression can be explicitly specified by means of the **as** operator. For example, consider the following request:

Among the students who are also staff, list those who are taking a course they teach.

This query can be formulated as follows:

```
for each student such that
 some course(student) = some course(student as staff)
print cname(student), sname(student)
```

4.2.3 Database Updating

Update operations in DAPLEX correspond to creation of a new entity, assigning function values, and modifying function values.

4.2.3.1 Creating a new entity

A new entity is created using a singleton expression

a new *set*

where *set* corresponds to a set expression. For example,

a new person

creates a new *person* entity and associates it with the variable *person*. When a new entity belonging to a certain entity type is created, all the supertypes of that entity type get populated simultaneously with that new entity. For example,

a new student

creates a new entity which is included in the extension of both *student* and *person* entity types.

4.2.3.2 Assigning or modifying function values

let, **include**, and **exclude** statements are used for this purpose. The **let** statement replaces the existing function value, if any, by the new value. Otherwise, it assigns the specified value as the result for the given set of arguments. The **include** statement extends the existing result set of a multi-valued function for the given set of arguments. The **exclude** statement excludes the specified value set from the result set of a multi-valued function for the given set of arguments. For example, consider the following requests:

For the student with Christian name 'Moyana' and surname 'Johns', change her current assignment to tutorial group to that held on Mondays at 2.0 pm in room 2c.

This request can be formulated as follows:

```
for the student such that
    cname(student)="Moyana" and sname(student)="Johns"
let tutorial(student)= the tutorial such that
                       day(tutorial)="Monday" and
                       time(tutorial)="14" and
                       room(tutorial)="2c";
```

For the student with Christian name 'Moyana' and surname 'Johns', add the courses CS1 and IS1 to her current course assignments.

This request can be formulated as follows:

```
for the student such that
    cname(student)="Moyana" and sname(student)="Johns"
include course(student)={the c1 in course such that
                         title(c1)="CS1",
                         the c2 in course such that
                         title(c2)="IS1"}
```

For the student with Christian name 'Moyana' and surname 'Johns', drop the course CS1 from her current course assignments.

This request can be formulated as follows:

```
for the student such that
    cname(student)="Moyana" and sname(student)="Johns"
exclude course(student)={the c in course such that
                         title(c)="CS1"}
```

4.3 Derived Functions

Derived functions are defined using **define** statements. For example, given the two functions

```
course(student) ->> course
staff(course) ->> staff
```

the following derived function

```
define staff(student) ->> staff(course(student));
```

returns the set of *staff* entities for a given *student* entity by evaluating the composite function.

A derived entity type is considered as a subtype of the role of the defining expression. For example, the following derived entity type

```
define female() ->> person such that sex(person) = "f";
```

defines a subtype of type *person*, which returns those *person* entities meeting the qualification.

DAPLEX provides the following special operators for defining derived functions:

inverse of : In order to define a derived function that relates all the students taking a particular course for the database of figure 4-1, we can proceed as follows:

```
define students(course) ->> student such that
                            for some c in course(student)
                            c = course
```

DAPLEX provides a special operator **inverse of** to simplify the derivation of the above function. Using this operator, the above function can be defined as

```
define students(course) ->> inverse of course(student)
```

Inverse functions can be defined for one-argument functions only. A single valued function may have the inverse function which is single- or multi-valued. Similarly, a multi-valued function may have a single- or multi-valued function as its inverse.

transitive of : If there exists a function

```
manager(employee) -> employee
```

which returns the manager of a given employee, the following derived function

```
define superiors(employee) ->> transitive of manager(employee);
```

returns the set containing the manager of a given employee, the manager's manager, the manager's manager's manager etc. The **transitive of** operator can be used to define one-argument functions only.

union of, intersection of, difference of : The **union of, intersection of,** and **difference of** operators applied to set-valued expressions can be used to define both entity types and functions with arguments. If a new entity type is defined, the new type is considered to be a subtype of each of the types of the set-valued expressions in the right hand side. If, on the other hand, a function with arguments is defined, then each of the set-valued expressions in the right-hand side must be of the same type which becomes the type of the function.

For example, the function,

define customer() ->> **union of** person, company, govtagency

defines the *customer* entity type consisting of entities belonging to either *person*, *company*, or *govtagency* entity type.

Similarly, the function,

define studentstaff() ->> **intersection** of student, staff

defines the *studentstaff* entity type consisting of entities which belong to both *student* and *staff* entity types, and the function,

define nonteaching() ->> **difference of** employee, staff

defines the *nonteaching* entity type which consists of those *employee* entities which do not belong to *staff* type.

compound of : The **compound of** operator is used to define new entity types only. This operator creates derived entities corresponding to the elements of the cartesian product of its operands. For example, if we have an entity type *student* and a function

course(student) ->> course

the following derived entity type,

define enrolment() ->> **compound of** student, course(student)

returns entities of *enrolment* type. The new type being defined will be a subtype of **entity** and will include one entity for each *student -course* tuple. In addition, the system implicitly defines the two functions

```
student(enrolment) -> student
course(enrolment)  -> course
```

which return the *student* and *course* entities for each *enrolment* entity.

4.4 User Views

All users of a database system might not want to see the database as defined in the central schema. The properties which are considered relevant and the mechanisms by which they are most naturally referenced vary across differing world views. Some users might prefer, for example, to view the enrolment of a student in a course as an entity having its own properties, while others might only be interested in the courses taken by a student. For them, dealing with enrolments as objects would be unnatural and awkward.

Shipman proposes the use of derived functions for defining different user views of a database. Because complex algorithms can be used in defining derived functions, view definition through derived functions is especially powerful. To separate the user's name space from the global name space, Shipman proposes that each user view be defined using a *module* or *package* facility provided by a high level language in which DAPLEX is to be embedded. Consequently, no details of this facility is included in his proposals.

4.4.1 Operations from Views

For a view to be useful, users must be able to apply retrieval and update operations to it. These operations from the view must be translated to functionally equivalent operations on the base functions. Because the mapping from the base functions to derived functions of a view is functional, retrieval operations from a view can always be mapped into equivalent retrieval operations on the base functions. However, such a mapping for translating updates from a view into equivalent updates on base functions may not always exist, and when it does exist, it may not be unique [Codd 74]. Views and updates exist for which no sequence of operations on the base functions will correspond to view updates (*undefined updates*) or more than one sequence of operations on the base functions may correspond to same view update (*ambiguous updates*) [Dayal 78]. In general, the ambiguity cannot be resolved automatically.

The strategy adopted here is to disallow updates from views unless the mechanism for that particular update has been explicitly specified in terms of available update operations. This is similar to the strategy proposed in the RIGEL [Rowe 79] language. Again, specific details of this mechanism are not well defined in DAPLEX.

4.5 Constraints

Shipman proposes the following forms of constraint specifications:

1) As an instruction to abort any update transaction which causes failure of a given pre-defined condition. For example, consider the following constraint: a department's head must belong to the same department.

This is specified as follows:

```
define constraint nativehead(department) ->
    dept(head(department))=department
```

This statement defines a boolean function *nativehead* which evaluates to **true** if the following condition holds good and false otherwise, and a constraint which instructs the system to abort any update transactions which leave the function value **false** for any department.

2) As a *trigger* facility to execute a specified program whenever any update transaction causes a given pre-defined condition to be satisfied. For example, consider the following request: inform the department head whenever more than 45 students are enroled in a class.

This is specified as follows:

```
define trigger overbooked(class) ->
    count(students(class)) > 45
    sendmessage(head(dept(class)),"overbooked:",
    name(class))
```

This statement defines a boolean function *overbooked* which evaluates to **true** if the following condition holds good and **false** otherwise, and an imperative which is executed whenever that function evaluates to **true** for any class.

4.6 Assessment of Shipman's Proposals

This section provides a detailed assessment of Shipman's proposals based on the set of criteria discussed in section 2.8.

1) **Object orientation**: This model makes a clear distinction between entities and names. The user is not required to be aware of the mapping from real world objects to values which act as tokens for them in the stored data. User-defined names of objects neither act as representatives of objects nor do they participate in defining inter-object associations. Hence, the referential integrity [Codd 79, Date 81] is guaranteed by the model itself. For example, consider the following update request:

Add CS1 to the set of courses taken by Angela Pearson.

This is expressed in DAPLEX as follows:

```
for the student such that
  cname(student)="Angela" and sname(student)="Pearson"
include course(student) = {the course such that
                              title(course) = "CS1"}
```

The above statement checks that the *course* entity to be included does indeed exist in the database. If it is not present, the update statement is not executed and the user informed about the non-existing entity.

2) **Semantic expressiveness:** This model organises entities into entity types, which are themselves organised in a subtype-supertype hierarchy. Each fact about an object is modelled as a function. Hence, semantically this model is highly expressive. For example, to a question "what is known about persons?", the functional schema of figure 4-1 provides the answer that each person is known to have attributes *cname* (Christian name), *sname* (surname), and *sex*. In addition, it provides the information that a person can be a student or a staff member. If he/she is a student, the functional schema provides the information that he/she attends a set of courses (since *course(student)* is multi-valued) and belongs to a tutorial group (since *tutorial(student)* is single-valued). If he/she is a staff, the functional schema provides the information that he/she teaches a set of courses (since *course(staff)* is multi-valued) and each tutorial is run by a staff member (since *staff(tutorial)* is single-valued).

```
student(studentno, cname, sname, sex)
staff(staffno, cname, sname, sex, room, phone)
course(courseno, title)
tutorial(tutorialno, day, time, site, room)
lecture(lectureno, day, time, site, room)
attends(studentno, courseno, grade)
belongs(studentno, tutorialno)
has(courseno, lectureno)
lectures(staffno, lectureno)
runs(staffno, tutorialno)
```

Figure 4-3: Relational Schema for the Student Database

In contrast with this, consider the relational schema for the same database shown in figure 4-3. It is not easy to answer the above question "what is known about persons?" just by examining the relational schema. Firstly, the information that both students and staff members are persons is not present in this schema. Secondly, to find out the relevant facts about, say students, one must first identify the token (*key*) that stands for students and then examine all the relations in which that token appears. The correct interpretation of such relationships may

become a significant problem if the schema has hundreds of relations with some of the entities having compound keys.

In FDM, extensions of different types can overlap, i.e., a *person* entity can belong to both *student* and 'staff' types. This corresponds to the various *roles* objects on the real world play. As discussed in Chapter 2, this fulfills an important requirement of information modelling.

3) **Ease of design**: It is easy to arrive at a schema using this model. Functions in this model represent irreducible relations as defined by Hall et al. [Hall 76a] and Biller [Biller 79]. Because of this, the information is canonically modelled, i.e., the application modelled as a set of functions in this model cannot be further decomposed, and there is only one fully decomposed form to represent any given set of facts in a model of real world. As shown in [Biller 79], such irreducible relations avoid the problems of normalisation present in record-oriented models.

4) **Neutrality**: There is no distinction between attributes and relationships in this model. Both are uniformly modelled as functions. It does, however, distinguish entities from relationships, but unlike other models which make this kind of distinction, this model allows one to interpret a relationship as an entity in its own right and vice versa. For example, consider the following schema:

```
declare student() ->> entity
declare course() ->> entity
declare course(student) ->> course
declare grade(student,course) -> string
```

The relationship between students and courses, defined by the function *course(student)* can however be interpreted as an entity type by the following derived function:

```
define enrolment() ->> compound of student,course(student)
```

Since the definition of the *enrolment()* function creates two implicitly defined functions *student(enrolment)* and *course(enrolment)* functions (see section 4.3), we can treat the *grade(student,course)* function as a property of the *enrolment* by the following derived function:

```
define grade(enrolment) ->
            grade(student(enrolment),course(enrolment))
```

Alternatively, if we had started with the following schema:

```
declare student() ->> entity
declare course() ->> entity
declare enrolment() ->> entity
declare student(enrolment) -> student
declare course(enrolment) -> course
```

it is an easy matter to derive the following functions:

```
define course(student) ->> course(e in enrolment
          such that student(e) = student)
define grade(student,course) ->
          grade(the e in enrolment such that
          student(e) = student and course(e) = course)
```

Thus, we see that this model provides for translation between various semantic interpretations. This feature is very useful for providing multiple user views of the same database.

In contrast, some of the semantic models advocate modelling both entities and relationships by a similar mechanism [Smith 80a, Codd 79] in order to provide neutral interpretation. For example, consider the following object definitions in the Semantic Hierarchy Model of Smith and Smith [Smith 80a]:

```
def author     : com
                    name, address,...
                end
def paper      : com
                   title,.....
                end
def authorship : com
                   paper, author
                end
```

That is, the relationship between *paper* and *author* is modelled by the object *authorship*, which has no correspondence with anything in the real world. Creation of such unnatural objects is avoided in the functional model, because the relationship between *author* and *paper* in FDM is modelled as a base function,

```
declare author(paper) ->> author
```

in one direction, and as a derived function,

```
define paper(author) ->> inverse of author(paper)
```

in the other direction. Even if the relationship is information-bearing, there is still no necessity to model the relationship as an entity because of the multi-argument functions allowed in the model. However, FDM also allows to look at a relationship as an entity by means of the compound of operation as shown above, if the user so desires.

5) **Operations:** It supports a complete set of operations. The data selection is purely in terms of logical properties and no knowledge of storage structures is required to formulate requests. In addition, the query formulations based on the function application provide a natural mode of expressing queries. For example, if *s* denotes an instance of a student, then the expression *cname(s)* yields his Christian name, and the expression *cname(tutor(s))* yields the Christian name of his tutor.

Because of the functional notation, queries in DAPLEX are close to the natural language form. For example, consider the following query on the student database of figure 4-1.

Which female students attending CS1 course belong to the tutorial group run by Malcolm Atkinson?

This query can be formulated as:

```
for each student such that
   sex(student) = "female" and
   cname(tutor(student)) = "Malcolm" and
   sname(tutor(student)) = "Atkinson" and
   for some course(student)
     title(course) = "CS1"
print cname(student), sname(student)
```

where *tutor(student)* refers to a derived function as shown in figure 4-1.

The same query expressed in relational algebra form of ASTRID [Gray 81] for the schema of figure 4-3 looks as shown below:

```
malcolm := staff selected on [cname = "Malcolm" and
           sname = " Atkinson"] projected to staffno;
malgroups := malcolm joined to runs projected to
           tutorialno;
female := student selected on [sex = "female"]
          projected to studentno, cname, sname;
malstuds := malgroups joined to belongs joined to
           female projected to studentno, cname, sname;
cs1 := course selected on [title = "CS1"] projected to
           courseno;
cs1studs := cs1 joined to attends projected to
           studentno;
result := malstuds joined to cs1studs projected to
           cname, sname;
output result;
```

The comparison of two formulations clearly shows the conciseness and clarity of the query expressed in the functional model.

Another important point about the functional languages is the possibility of relating databases and programming languages through the common concept of function. As discussed in the previous chapter, the attempts with the relational model and the CODASYL model have been hampered by a lack of such a common concept.

6) **Facilities to specify the derived data**: The derived function mechanism in this model makes it possible to capture derived data as part of the schema. As discussed in section 2.7.1, this has many useful applications. One such use of derived functions is to model procedurally-defined relationships. For example, the full name of a person can be defined as a procedure

```
define fullname(person) -> cname(person) ++ " " ++ sname(person)
```

Derived functions also serve as an encoding of inference rules. For example, function composition shows how two relationships can infer a third as in

define tutor(student) -> staff(tutorial(student))

or a function infers its inverse as in

define student(course) ->> **inverse of** course(student)

Another use of derived functions is to break down a complex query into a series of simpler steps. For example, consider the following query:

What are the names of students who take a course taken by Angela Pearson?

This query formulation can be broken down into a derived function definition and a FOR loop statement, as shown below:

```
define samecourse(s1 in student, s2 in student) ->
     for some c1 in course(s1)
     for some c2 in course(s2)
     c1 = c2

for the s1 in student such that
   cname(s1) = "Angela" and sname(s1) = "Pearson"
for each s2 in student such that
   samecourse(s1, s2)
print cname(s2), sname(s2);
```

In effect, the derived function capability allows application semantics to be encoded into the data description, thereby allowing requests to be expressed directly in terms of those semantics.

As discussed in section 4.4, another use of derived functions is to provide for the definition of diverse user views. Collections of derived functions can be packaged to present an appropriate view of the database to a user.

7) **Facilities to specify constraints**: As discussed in section 4.5, this model provides facilities to specify constraints.

8) **Ease of evolution**: The schemas in this model can gracefully evolve. The functions in this model represent irreducible facts and hence represent natural increments to the schema. The **declare** and **define** statements of DAPLEX allow dynamic additions to the schema, thus providing incremental design/update of the database structure.

9) **Freedom from physical considerations**: Since the users need not be aware of how the functions are implemented, this model provides a complete freedom from physical considerations.

10) **Complexity**: It uses a minimal number of different terms to describe a database, thus minimising the number of terms that have to be learned by users and implemented in the system. Hence, it is simple to understand and use.

Chapter Five

Extended Functional Data Model

In this chapter we discuss the various facilities of our implementation of the functional data model, *Extended Functional Data Model (EFDM)* system. We also provide a brief comparison of EFDM with the proposed ADAPLEX database management system, currently under development at Computer Corporation of America (CCA) [Smith 81, CCA 83].

EFDM is an interactive system. Users create, operate, and modify their databases through interactive commands. Both the model and the language underlying EFDM closely follow the proposals of Shipman. A significant difference, however, is that instead of embedding DAPLEX in a compile-and-run procedural language for a working system, we have formulated a self-contained language. The syntax of some of the DAPLEX constructs has also been simplified. The database update facilities are enhanced to provide for the explicit movement of entities from one entity type to another and explicit deletion of entities from the database. Schema modification facilities include the explicit deletion of functions.

A number of new features have also been added to provide many useful facilities such as checks for schema consistency, defining and using views, loading bulk data from the operating systems, naming and storing user queries, transferring the results of queries to the operating system files etc.

A brief description of these modifications and extensions is given in the following sections. More detailed information about EFDM can be found in the user manual [Kulkarni 83]. A summary of the EFDM syntax appears in Appendix B.

5.1 Structures

As in Shipman's proposals, EFDM models the real world information as sets of entities and functions mapping entities to entities. Likewise, entities have types which are arranged in a type hierarchy, so that they are all subtypes of the type **entity**. Functions can again have zero, one or more arguments and a function may be single-valued or multi-valued. Subtypes inherit functions defined on their supertypes. A function which is represented by physically storing a table of arguments and results is called a *base function* and a function introduced by an algorithm to compute its result is called a *derived function*.

We have made two changes to Shipman's proposals as far as data model constructs are concerned. They are as follows:

1. Unlike Shipman's proposals, single-valued functions in EFDM are assumed to be partial by default. This has two advantages: a) it is possible to accommodate objects for which some data is not known, and b) inverses of functions can be freely defined even if they are only defined on a subset of an entity set.

2. Shipman allows the arguments of functions to be arbitrary expressions. The reason for this seems to be the problem of handling multi-argument functions with entity types as arguments, as such functions may not be well defined for all combinations of argument entity sets. Hence, by adopting expressions as arguments, one can use them as constraints on the possible combinations of argument entities, i.e., the function is defined for only those entities evaluated by the corresponding expressions. In contrast, the arguments of functions in EFDM, if any, must all be entity types. Since functions in EFDM are allowed to be partial, entity types acting as arguments for multi-argument functions do not cause any problems. However, the fact that the function should possess values for only a subset of the cross-product of argument entity sets can be accommodated by specifying it as an explicit constraint. (See section 5.5)

5.2 Operations

EFDM provides operations corresponding to data definition, data manipulation, I/O, and general-purpose computation. Data definition and data manipulation operations essentially correspond to those provided in DAPLEX. In addition, EFDM supports arithmetic, string manipulation, and boolean operations as well as recursion for defining derived functions. EFDM provides print and output statements for output operations and a bulk load facility for inputting data from the operating system files [Kulkarni 83].

In contrast, DAPLEX limits itself to data definition and data manipulation operations.

Shipman envisages embedding DAPLEX in a general purpose high-level language to provide these facilities. There are two possible criticisms of this approach:

1. There is a considerable mismatch between the normal programming language objects and the DAPLEX objects. The programming language objects are characterised by a value and a structure. In contrast, DAPLEX objects have neither a value nor a structure. They are atomic units which gain definition from their relationships with other objects. In fact, they can exist independently from such relationships. Because of this, DAPLEX entity types do not serve the same purpose as normal programming language types. Hence, embedding DAPLEX in an algol-like languages violates the principle of data type completeness [Strachey 67]. This is sure to result in a complex language.

2. DAPLEX programs are predominantly applicative or functional in style. Hence, embedding DAPLEX in a procedural language is sure to result in two different styles of programming in the same language.

5.2.1 Data Definition

The data structure of an application is described in the form of function definitions using **declare** and **define** statements. The **declare** statement is used to enter a new base function or entity type definition and the **define** statement is used to enter a new derived entity type or function. Like DAPLEX, we allow the individual function declaration statements to occur at any time. In addition to **entity**, three primitive types: **string**, **integer**, and **boolean** are supported.

5.2.2 Maintaining Schema Consistency

As each new fact is added to the schema, it is necessary to make sure that it is not inconsistent with the existing information in the database. One possible reason for inconsistency in the functional schemas is due to the failure on the database designer's part to ensure that the same fact is not described by more than one base function. For example, assume that there exists a base function

```
student(course) ->> student
```

Suppose now the user intends to add another function

```
course(student) ->> course
```

If, in reality, this function is the inverse of the old function, adding it as a base function will mean that the same fact is represented by two independently updatable functions, and this will surely lead to inconsistencies in the database. On the other hand, if the new function

corresponds to a new fact, say relating to the major courses taken by a student, it is to be added as a base function.

Since there is no way for the system to infer what is intended, EFDM checks with the user/designer to confirm all base function addition requests. For instance, suppose that the user intends to add a base function

tutor(student) -> staff

between the *student* and *staff* types, where the only functions relating these types are as shown in the entity diagram corresponding to figure 5-1. If we consider the entity diagram as an undirected graph with the entity types corresponding to nodes and functions corresponding to edges, various paths between any two nodes correspond to all the existing base and derived functions that directly relate the corresponding two types and the possible implicit functions relating the two types that can be derived using function inversion and composition. In the present example, various paths between *student* and *staff* nodes are listed below:

course(student), **inverse of** course(staff)
tutorial(student), staff(tutorial)

That is, there are no functions that directly relate the two types, but there are two possible implicit paths.

The user can examine the two "paths" and see whether the new function he intends to add corresponds to any of these "paths". For example, the user may discover that the new function corresponds to the second path, i.e., the composition of *tutorial(student)* and *staff(tutorial)* functions. He can then abort the earlier request and define it as a derived function. On the other hand, if he is sure that the new function does not correspond to any of the "paths", he can let the function addition proceed as base function.

Thus, EFDM helps the database designers to avoid inconsistencies in the schema by encouraging them to examine whether each new information that enters the system is 'initial' or whether it can be derived from other information that already exists. This kind of exploration also enhances the user understanding by stimulating and generating questions as to how the reality is abstracted and which assumptions are made. Many incomplete and inconsistent assumptions are thereby detected leading to a more valid and consistent model.

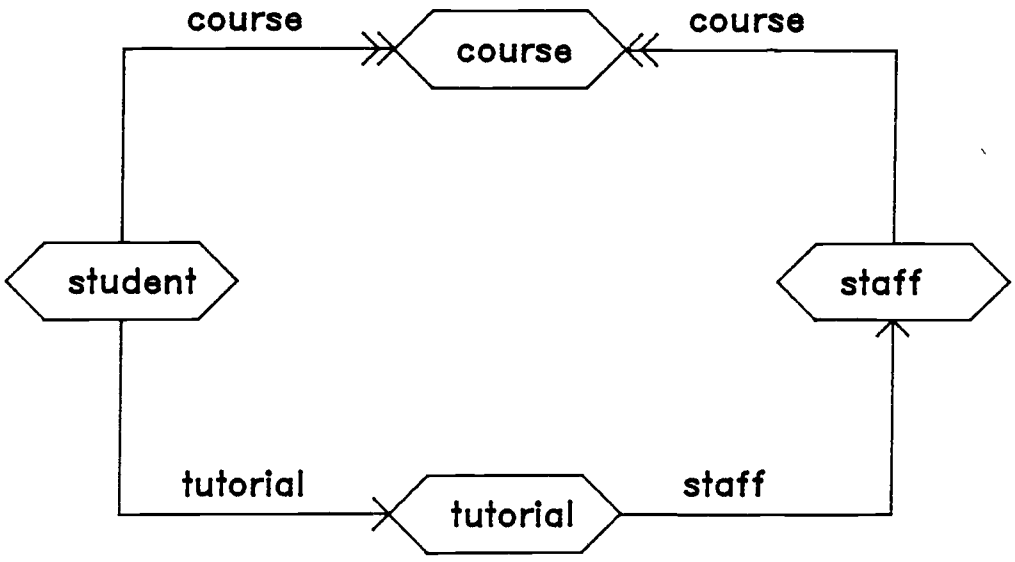

Figure 5–1: *Partial Entity Diagram for Student Database*

5.2.3 Schema Updating

Shipman's proposals are limited to the addition of new functions only. However, users may also want to drop existing functions if they are no longer interested in maintaining them. For this purpose, we provide drop statement. For example, to drop the *course(student)* function, we use:

```
drop course(student);
```

To avoid inconsistencies, **drop** may cause cascade deletion of functions that depend on the function being deleted. Prior to executing the request, the user is provided with a list of such implicit deletions and confirmation is sought to proceed with the execution.

5.2.4 Meta Data

The meta data of the schema corresponding to an application is held in a set of EFDM functions shown in figure 5-2. These functions are automatically populated and modified when **declare**, **define** or **drop** statements are processed. Only the document function may be explicitly updated by the user. The contents of these functions can be retrieved with the usual retrieval statements. So a user may use such queries to discover the form of a database. To facilitate this the functions given in figure 5-3 are defined.

```
function()                  ->> entity
name(function)              ->  string
arguments(function)         ->> function
result(function)            ->  function
type(function)              ->  string
status(function)            ->  string
text(function)              ->  string
document(function)          ->  string
```

Figure 5-2: The functions to hold meta data of a schema

5.2.5 Data Selection and Retrieval

As in DAPLEX, FOR loop statements and expressions are the basic constructs for data retrieval. However, there are some differences between the two.

One difference is about the variable declaration associated with set expressions. DAPLEX provides for an implicit variable to be associated with every set expression. The translator then decides whether a name refers to a set (entity type) or an instance (variable). We

```
entitytype()                ->> f in function such that
                                nargs(f)=0
supertype(entitytype)       ->  result(entitytype)
supertypes(entitytype)      ->> transitive of
                                supertype(entitytype)
subtype(entitytype)         ->> e in entitytype such that
                                result(e)= entitytype
subtypes(entitytype)        ->> transitive of
                                subtype(entitytype)
fnsover(entitytype)         ->> f in function such that
                                nargs(f) ~= 0 and
                                some e in arguments(f) has
                                (e = entitytype or
                                some e1 in
                                supertypes(entitytype) has
                                e = e1 )
fnsyielding(entitytype)     ->> f in function such that
                                nargs(f) ~= 0 and
                                result(f) = entitytype
```

Figure 5-3: The derived functions for querying meta data

have removed this name overloading and made it compulsory to specify a variable explicitly for every set expression by means of the **in** operator.

Another syntactic change from DAPLEX is the avoidance of the use of the **for** keyword for expressing quantified set expressions. In EFDM, the keyword **for** is used for FOR loops only. Quantified set expressions are specified by the following syntax,

quant set (**has**|**have**) *singleton*

where the *singleton* following **has** or **have** evaluates to a boolean result and *quant* is any one of the following quantifiers: **some**, **all**, **no**, **at least**, **at most**, or **exactly** (last three quantifiers must be followed by a *singleton* evaluating to an integer result). For example, consider the following query used in the previous chapter:

Find the Christian name and surname of all the students taking the IS1 course.

This is expressed using the revised syntax as follows:

```
for each s in student such that
some c in course(s) has
csnumber(c)="IS1"
print cname(s),sname(s);
```

Also, EFDM allows the following operators to combine set-valued expressions. (In DAPLEX, these set operators are allowed for defining derived functions only.)

union for set union
intersection for set intersection
difference for set difference

In addition, EFDM allows arithmetic, string, and boolean operations. The following arithmetic operators with usual precedence rules are allowed to combine integer-valued expressions:

+ for addition
- for subtraction
* for multiplication
rem for remainder

There is only one string operator ++ defined on string-valued expressions. It concatenates the two operand strings to form a new string.

The following boolean operators are allowed to combine boolean-valued expressions:

and, **or**, **not**

Relational operators (<, <=, >, >=, =, ~=) are allowed between singleton expressions only. The first four operators are allowed for integer- or string-valued expressions only.

5.2.6 Packaged Queries

A query can be named by preceding the query statement with

program *programid* **is**

For example, the following statement

```
program females is
for each p in person such that sex(p)= "f"
print cname(p), sname(p);
```

assigns the name *females* to the corresponding query. Such a named query can be executed any time by typing its name in response to the system prompt.

The information about currently defined queries is held in pre-defined EFDM structures. These are shown in figure 5-4.

```
query()       ->> entity
name(query)   ->  string
text(query)   ->  string
```

Figure 5-4: The functions to hold meta data for queries

These can be queried using the above data retrieval facilities. For example, to list all the existing query definitions, we can use

```
for each q in query
print text(q);
```

Any existing query can be removed using the **drop** statement. For example, the following statement,

drop females;

removes the above query definition.

5.2.7 Displaying the Results

The **print** statement is used to display the results. This has the following syntax:

print *singleton* { "," *singleton* }

where each *singleton* expression yields printable entities, i.e., strings, integers or boolean values.

The results of a query normally appear on the screen. If the results are to be stored in a file, it is necessary to package the query using the **program** statement as above. The command

output *programid filename*;

executes the query, creates a file with the specified name and directs the output to that file instead of the screen.

5.2.8 Database Updating

As in DAPLEX, the update facilities in EFDM include the creation of new entities and assigning or modifying function values. However, unlike DAPLEX, EFDM provides operations to move entities from one entity type to another. In other words, EFDM allows extending or reducing the set of types for a given entity. This kind of facility is necessary in cases, say, the user has created a *person* entity and that person becomes a student at some later date, or the user has created *a student* entity and that student ceases to be a student at some later date.

Again, unlike DAPLEX, EFDM allows explicit deletion of entities from the database. DAPLEX assumes that entities get effectively deleted when the users take away all references to it. However, in our experience the users are often interested in the opposite, i.e., specifying an entity deletion in one step so that the system removes all references to it automatically.

In addition, unlike DAPLEX, EFDM does not insist that values for all single-valued functions applicable to an entity be provided with values at the time of its creation itself. This is

because the functions in EFDM are assumed to be partial by default. However, to be able to assign values at a later time, it should be possible to identify the newly created entity. Hence, at least one function (or a group of functions) must be populated which distinguishes that entity from all the existing entities in that type.

5.2.8.1 Extending the set of types of an entity

The syntax for including an existing entity into the extension of a specified entity type is the same as that used to include entities into multi-valued function extensions. For example, to include a student with Christian name 'John' and surname 'Smith' into *staff* type, we use

```
include staff = {the s in student such that
                 cname(s)="John" and sname(s)="Smith" }
```

5.2.8.2 Reducing the set of types of an entity

The syntax for this case also is the same as that used to exclude entities from the multi-valued function extensions. For example, to exclude the student with Christian name 'Moyana' and surname 'Johns' from the *staff* type, we use,

```
exclude staff = {the s in student such that
                 cname(s)="Moyana" and sname(s)="Johns"}
```

Excluding an entity from the extension of a type results in removing its reference from the extensions of all subtypes of that type, if any, and from all functions in which it is participating either as an argument or result. Hence, before carrying out the operation, a list of these implicit updates is displayed and the user is asked to confirm the request.

It is to be noted that excluding an entity from the extension of an entity type which has other user-defined supertypes only removes a specialisation of that entity and the entity itself continues to exist in the database. For example, the above operation only removes the specified entity from the *staff* type, while the entity itself continues to exist in the database as *student* entity. Removal of an entity from the extension of an entity type which has no other user-defined supertypes results in deleting that entity from the database itself. The same effect can be achieved by delete syntax described below.

5.2.8.3 Entity Deletion

Entities can be deleted using

delete *singleton*

which deletes the specified entity and causes a cascade deletion of all functions which reference this entity and all its subtype entities, again consulting the user before the cascade

proceeds. For example, to delete a student with Christian name 'Moyana' and surname 'Johns' from the database, we use

```
delete the s in student such that
        cname(s) = "Moyana" and sname(s) = "Johns";
```

The difference between this statement and the **exclude** statement is that while **exclude** removes a specified entity from the specified type and its subtypes, **delete** removes a specified entity from the specified type, its subtypes and its supertypes.

5.3 Derived Functions

As in DAPLEX, derived functions are defined using **define** statements. However, there are two differences:

1. EFDM provides the **union**, **intersection**, and **difference** as general set operators, instead of being specific to defining derived functions;

2. EFDM allows recursion while defining general-purpose derived functions. For example, we can define functions like:

```
define power(i in integer, j in integer) ->
                                    if j=0 then 1
                                    if j>0 then i*power(i,j-1);
```

5.4 User Views

EFDM provides a view mechanism which while providing a different perspective of the global information, also acts as an authorisation mechanism. Using this mechanism a central database administrator who has access to the entire database can define different overlapping user views. The underlying assumptions of this mechanism are:

1. There exists a *global* view from which all user views are derived. That is, we assume the structure and contents of this global view are arrived at by integrating the different application views to one common community view.

2. The global view encompasses all the information required by all the users. If certain information required by a user is not in his view, he must request a central authority (database administrator) to include it in his view, who will, in turn examine whether the information requested by him is already available in the global view and if not, he will take steps to include it in the global view and then include it in the user's view.

Views are defined using **deduce** statements. For example, for the student database of figure 4-1, we can define a view called *malestudents* as

```
view malestudents is

deduce male()          ->> entity using student
                               such that sex(student)="m"
deduce name(male)      ->  string using name(student)
end
```

All functions introduced by **deduce** are treated as derived functions. Notice that **deduce** is used to define view functions instead of **define**. This is because view function definitions involve change of name space; names before the using keyword refer to names in the local name space whereas names after the using keyword refer to names in the global name space. In fact, every view definition automatically creates a different name space, which is completely independent of the global name space as well as the name spaces of other views. The users of a particular view are not allowed to see the global name space or the name spaces of other views.

Views can be dropped with the

drop *viewname*

command.

The information about currently defined views is held in pre-defined EFDM structures. These are shown in figure 5-5.

```
view()                 ->> entity
name(view)             ->  string
context(view)          ->  view
text(view)             ->> string
password(view)         ->  string
document(view)         ->  string
```

Figure 5-5: The functions to hold the meta data for views

5.4.1 Operations from Views

On initiating the system, the user is prompted for the view name, which is in effect a password. If the view corresponding to the user-specified name exists in the database, the name space corresponding to that view is made available to the user. If the view corresponding to the user-specified name does not exist in the database, a message is given to the user and the session is terminated. The user can then request the central database administrator to create the view.

Once within a view, users can pose the usual EFDM requests to carry out the database operations using the names available in their name space. Each query statement issued from a user view is translated into a corresponding query on the global name space by recursively applying the view definition mapping.

Since all the functions in a view are treated as derived functions, updating through view functions is allowed only if procedures for updating the corresponding global data are provided for each view function. Shipman's proposals in this regard are incomplete in the sense that they cover only those cases in which a view update results in a single update action on stored data. In general, the situation is more complicated and a view update may result in multiple updates on stored data. For example, creation of new entities from views may result in creation of new entities in the global view with simultaneous updates on some function values. To handle this case, the language needs a construct to specify the execution of group of statements executed in a particular order. Hence, presently updating through views is not allowed.

The only schema changes that are allowed from a view are either defining derived functions from the set of functions they are allowed to have access to or dropping the derived functions they have created. This is in keeping with the assumption that the global view has all the information required by different users. This means that the individual users are not allowed to introduce any base functions or stored data of their own.

5.5 Constraints

The management of the semantic integrity of a database has four components:

1. *Expression* of constraints: This is usually done as assertions that apply to the database. An assertion also includes the qualification of the units of data, or *variables* to which the constraint applies. Variables are qualified by the conjunction

of their type and values. A qualification may specify one or more or all instances of a type.

2. *Checking* for constraint violations: The database integrity is potentially threatened by every update, and therefore, every update is associated with some checking.

3. *Maintenance* (i.e., enforcement) of constraints: The maintenance of constraints usually consists of rejecting or undoing updates which cause violations or of performing further updates to compensate for the faulty ones.

4. *Reporting* constraint violations: The reporting of constraint violations involves informing the user details about entities that fail to meet a particular constraint at the time of its specification or about constraints that are violated as a result of particular update action.

Shipman's proposals include a few tentative suggestions on how to specify the constraints, but these are not exhaustive. In what follows, we report our investigations on identifying an useful set of constraints. However, we have not been able to implement a general constraint handling facility and hence, no definite syntax is proposed to specify these constraints.

5.5.1 Constraints on entity identification

For a given database, the users may be interested in distinguishing individual entities so that they can refer to an entity in an unambiguous manner. This can be specified as an integrity constraint. This is in contrast to some of the existing data models where the object identification mechanisms are made part of the conceptual structure itself, e.g., the concept of *key* in relational model. In some of these models, once specified in the conceptual structure, the users cannot drop or change these specifications.

In FDM, an object can be identified by a group of function values taken together. Thus, certain sets of functions associated with a given entity type can be designated as *unique*, i.e., no two entities of that entity type can have the same combination of values for those functions. In practice, there may be more than one set of such functions which uniquely designate the objects of that type. For example, employees in an organisation may be uniquely identified both by their employee numbers and national insurance numbers.

Since subtypes inherit all the functions associated with their supertypes, it follows that if an entity type has unique functions specified for it, all its subtypes also inherit those unique functions. Hence, it is not necessary to specify unique functions for any of the subtypes of a given entity type. However, the users may optionally define additional sets of unique functions for some or all subtypes of a given entity type.

It is not necessary that all entity types in the database have unique functions specified for them. In case an entity type has no unique functions specified for it, the objects belonging to that entity type cannot be directly distinguished. However, they may be indirectly distinguished by their association with other objects. To refer to such objects, the users may have to go via their associated objects.

5.5.2 Constraints on entity associations

There are two cases of such types of constraints. These are:

1. It is often the case that every object belonging to an entity type must, at all times be associated with another object in the database. For example, an employee must always belong to a department. These constraints can be modelled by specifying the corresponding functions as *total*, i.e., these functions must be provided with values (not necessarily distinct) for each entity belonging to their argument types.

2. In general, extensions of different entity types can overlap. There may be cases when such overlap is not permitted. For example, when no constraint to the contrary exists, a given *person* entity can have both *student* and *staff* types simultaneously. If this is not acceptable, it can be prevented by a corresponding constraint which specifies them to be disjoint.

5.5.3 Constraints on the values of the functions

There are two cases for such types of constraints. These are:

1. Certain functions may be constrained to have fixed values, i.e., the function value for a given entity can only be assigned at the time it is created and this value cannot be changed throughout the life time of that entity. For example, an employee's employee number can only be assigned at the time of hiring him and cannot, in general, be changed during his employment.

2. Certain functions may be constrained to have certain values or range of values for every argument value. For example, consider the following constraints:

    a. A department's head must belong to the same department.

    b. The salary of an employee must be greater than a certain prescribed minimum and be less than a certain prescribed maximum which depends on his status.

    c. The *course* entity in the multi-argument function *grade(student, course)* must correspond to those provided by the *course(student)* function.

Such constraints can be expressed as general predicates associated with the corresponding functions.

5.5.4 Constraints on cardinality

This type of constraint specifies the number of possible values for the population of entity types and multi-valued functions. For example, there may be a constraint that the number of managers must be less than the number of non-managers, or the number of courses a student can take must be a minimum of 3 and a maximum of 5.

5.5.5 Constraints on existence

This type of constraint specifies that certain objects exist in the database only if some other objects also exist. For example, there may be a constraint that the details about an employee's children can exist in the database only if the details about the corresponding employee exist in the database, or the *jobtitle* attribute of an employee can exist only if he is assigned to a department etc.

5.6 Comparison with ADAPLEX

ADAPLEX is the proposed Ada-compatible database management system which supports a composite language called ADAPLEX [Smith 81, CCA 83]. ADAPLEX is an embedding of a subset of DAPLEX into the language ADA [Ichbiah 79]. In choosing ADA as the host language, ADAPLEX hopes to exploit the *modules*, *tasks*, and *generics* etc. of ADA to provide some of the encapsulation needed for supporting a number of concurrent users.

For specifying the database structure, ADAPLEX adds two new type constructors to ADA. These are: *set types* and *entity types*. A set type encapsulates the mathematical notion of a set. An entity type is similar to an access-to-record type, in that, it has named components and may be created dynamically. However, unlike the access-to-record type, each component of an entity type is designated by an entity function. An example of an entity type declaration in ADAPLEX is:

```
type person is entity
  name : STRING (1..30);
  age  : INTEGER
  phone: set of STRING(1..11);
end entity;
```

In an earlier specification of ADAPLEX [Smith 81], the subtype relationship between entity

types was specified by the **contain** statement and the extent overlap among entity types was specified by the **share** statement. For example,

contain staff **in** person;
contain student **in** person;
share staff **with** student;

specifies that both *staff* and *student* are subtypes of *person* type, i.e., both staff members and students are persons and some staff members may also be students and vice versa. In this version, when there was no specific overlap between two entity types, the two types were constrained by default to be disjoint (i.e., non-overlapping).

In the latest specification of ADAPLEX [CCA 83], the above declarations would be written as:

subtype staff **is** person;
subtype student **is** person;
overlap staff **with** student;

In addition, the new version specifies that the collection of subtypes for a supertype must cover their supertype, i.e., an entity value belonging to a supertype must also belong to one or more of its subtypes.

ADAPLEX has also introduced new constructs for specifying uniqueness constraints. A uniqueness constraint specifies, for a particular entity type or subtype, a collection of components whose values are conjointly unique in every value of that type or subtype. A given entity type can have zero, one, or more uniqueness specifications.

Data manipulation constructs in ADAPLEX closely follow those of DAPLEX with some minor modifications.

Compared to the approach adopted in EFDM, we can list the following problems with ADAPLEX approach:

1. Problems with the data model: Firstly, in order to make the entity type definitions compatible with ADA type definitions, ADAPLEX has made a substantial change in the semantics of the DAPLEX entity types. In complete opposition to the spirit of DAPLEX, the entity functions cannot be introduced dynamically in ADAPLEX. Secondly, the lack of multi-argument functions in ADAPLEX means that it is necessary to adopt the *excess entity* approach used in the CODASYL model to model n-ary relationships. Thirdly, it is rather strange to insist that a supertype must be covered by its subtypes. This prevents the accommodation of, say those persons who are neither staff members nor students if *staff* and *student* are the only subtypes of *person* entity type. Fourthly, ADAPLEX does not support the derived function mechanism. As we saw earlier, the functional approach gains considerable modelling power by means of the derived function mechanism.

By failing to exploit this, ADAPLEX falls short of the modelling power achieved in EFDM.

2. Lack of data type completeness: In ADAPLEX, all database types, type constructors and operators become available in the ADA language, but the inverse is not entirely true. For example, ADAPLEX allows the construction of sets of entities, strings, and scalar types but prohibits the construction of sets of objects of any other type. Though the Ada access-to record types look much like database entity types, the semantics of the two are entirely different. First, each entity type is associated explicitly with the collection of dynamically created objects of the type. Second, entity types are allowed a much richer subtype structure than access-to-record types. Third, a strong discipline is imposed on the components that entity types may contain. The language contains a number of such exceptions and special rules. Because of this, it is a complex and confusing language.

3. Different styles in the same language: As an example of this, consider the syntax for accessing the component attribute of an entity. Though the syntax for entity type declarations is clearly based upon that for access-to-record declarations, the syntax for accessing a component attribute of an entity is based upon the DAPLEX syntax. For example, the *name* field of an entity *p* of the *person* type defined above would be represented as:

   ```
   name(p);
   ```

   rather than as

   ```
   p.name;
   ```

   In addition, since the underlying data types of ADA are not obviously amenable to the DAPLEX function and set operators, it will not occur to most programmers that many of the procedural forms of Ada could be replaced by DAPLEX statements.

Chapter Six

Implementation of EFDM

In this chapter, we discuss issues connected with the implementation of EFDM. First, we provide a brief history of the activities leading to the choice of the persistent algorithmic language, PS-algol [Atkinson 81a, Atkinson 83a, Atkinson 83b] as the implementation language for EFDM. We then present the software architecture of the implementation followed by a short discussion of the particular implementation strategy we have chosen and the problems that arise due to large volumes of data.

6.1 Brief History

The implementation of the functional data model started with the development of a prototype supporting almost all the basic concepts of the functional data model and a substantial subset of the DAPLEX. It was developed using the PASCAL language [Wirth 71]. This implementation provided an interactive user interface, using which users could enter base or derived functions, populate the base functions with data using update statements, and retrieve and display the data from the database using FOR loop statements. Access to meta data was also provided by a special set of pre-defined functions (see 5.2.4), which users could query using the usual FOR loop statements.

In this version, all the data structures were core based and hence, there was no provision to support persistent data. In order to support persistent data, an attempt was made to interface this system to an existing low-level persistent data space management system called *Chunk Management System* (CMS) [Atkinson 83c]. CMS is a set of routines that manage a large disk-based heap upon which variable size chunks of untyped data may be securely stored. It provides the atomic operations of creating, writing, reading, extending, or deleting a chunk of storage. It also provides a secure multi-user transaction facility to effect these atomic operations.

By providing a machine and operating system independent environment, the CMS simplifies the task of implementing database management software. However, CMS does not record any structural information associated with the data and it is the programmer's responsibility to ensure type consistency and provide adequate structural transformations.

At the same time, the work of the Data Curator Group resulted in the persistent algorithmic language, PS-algol [Atkinson 81a]. By abstracting the storage management, PS-algol allows one to treat all data as though it is in the main store. Because of this, there is no necessity for programs written in this language to explicitly interface with a low-level data handler. This made it unnecessary to continue the above implementation strategy and the system was completely re-written in PS-algol language. Another consequence of switching over to PS-algol was the significant reduction of the source code (about one third) compared to the earlier PASCAL version. Reduced code volume meant reduced coding times and this had a better than proportional effect on debugging times.

The following section decribes this language in more detail.

6.2 PS-algol Language

PS-algol is an algol-like language derived from the strongly typed programming language S-algol [Morrison 79]. S-algol stands somewhere between Algol W [Wirth 66] and Algol 68 [van Wijngaarden 69]. It was designed using three principles first outlined by Strachey [Strachey 67] and Landin [Landin 66]. These are:

1. The principle of correspondence.

2. The principle of abstraction.

3. The principle of data type completeness.

S-algol data types comprise of the usual scalar types, vectors and structures with any number of fields, and any data type in each field. However, the unusual features of the S-algol language universe of discourse are that it has strings as a simple data type [Morrison 82a], pictures as compound objects [Morrison 82b] and run time checking of structure classes. Because of the last facility, a pointer may roam freely over the world of structures, i.e., a pointer is not bound to a structure class. However, when a pointer is dereferenced, a run time check occurs to ensure the pointer is pointing at a structure with the appropriate field. S-algol provides the operators *is* and *isnt* to check the class referenced by a pointer.

In addition, S-algol identifier declarations are initialising, i.e., the programmer must specify the initial value for an identifier at the time of its declaration. By forcing the user to specify an initial value one type of error, that of an uninitialised name, is completely eliminated. Another useful feature of the language is that a programmer may specify whether the value of an identifier is constant or variable. A constant may be manipulated in exactly the same manner as a variable except that it may not be updated.

Syntactically, PS-algol is exactly identical to S-algol. However, the major difference between the two languages is the way they manage the heap provided for automatic storage management facilities. The S-algol heap contains only transient data whereas the PS-algol heap may contain some persistent data. That is, the data defined on the S-algol heap is lost when the program terminates. If a data structure built upon the heap is to be preserved until another run of the same program, or used in a different program, the data must be converted to some other representation and output to a file. In contrast, PS-algol supports a persistent heap on which a data structure built in one run of a program may be preserved to be used in other runs of the same or other programs.

The persistence of the data in PS-algol is achieved by storing that part of the heap identified as persistent in a database with its type and pointer structure intact. The means of identifying persistent data in PS-algol is the same as those used in languages like S-algol for identifying limited data persistence during the run of a program, viz., data reachability. In these languages, garbage collection preserves all data that is reachable from some other data. PS-algol extends this principle by introducing a new origin for the transitive closure of references, under explicit user control, which differentiates persistent data and transient data. Thus when a transaction is committed, it is possible to identify a root object from which all persistent data is reachable. Hence the preservation of data is a consequence of arranging that there is a way of using that data.

The movement of persistent data between main store and backing store is handled by the PS-algol run-time support system. On the first dereference of a pointer to a structure containing persistent data, that structure is copied to the heap from the secondary storage, possibly carrying out minor translations. Thereafter it is operated on by the same mechanism as for any other data on the heap. When a transaction is committed, all the data on the heap that is reachable from the persistent objects used during the transaction are transferred back to the disk.

This feature makes it possible for PS-algol to provide persistence as an orthogonal property of data and further simplifies the provision of access to persistent data. PS-algol also provides a library of routines to manipulate special structure classes called *tables*. A table is an ordered set of pairs. Each pair consists of a key and a value. A key may be an integer or string value, and the value is a pointer to a structure instance or table. The set of procedures to manipulate the tables include creation of an empty table, storing a pair, a mechanism for associative lookup implemented as B-trees [Bayer 72] or equivalent algorithms, and iteration over tables.

The following two programs illustrate some of the facilities of PS-algol.

The first program uses the PS-algol tables facilities to create a database of persons. The database is created as a PS-algol **table** structure with the person's name as key and a structure containing the name of a person's manager and his salary as the associated entry.

```
! Declare structure class "person"
structure person (string manager; int salary)

!create the database as a table
let person.db := table
let name      := ""

let finished := false
while ~ finished do
begin
  let p := person ("",0)
  write " Name    : "; name          := reads
  write " Manager: "; p (manager) := reads
  write " Salary : "; p (salary)  := readi
  write "'n More persons (y/n) ? "
  finished := read ~= "y"
  !enter p into the database with name as the key
  enter (name,person.db,p)
end
! structures accessible from the root table
! are made persistent; so enter person.db
! into root table.
enter ("Persons", root.table,person.db)
!end of transaction.
commit
```

The next program, given the name of a person, retrieves the name of his manager, his manager's manager, etc.

```
structure person (string manager; int salary)
! get the persons table from the database
let person.db := lookup ("Persons", root.table)

procedure print.manager (string name)
!-------------------------------
begin
  let p = lookup (name, person.db)
  if  p = nil
    then write "'nManager of ", name, " not found"
    else
    begin
      write "'nManager of ", name, ":", p(manager)
      print.manager (p(manager))
    end
end

!Main Program
write " Name of person: "
print.manager (reads)
```

Compared with ordinary high level languages, PS-algol offers the following attractive benefits for implementing applications involving persistent data, such as database management systems:

1. In conventional implementations of database management systems, a major part of the code is devoted to organising data movement. In contrast, the programmers using PS-algol never explicitly organise data movement; it happens automatically when they try to use data. Because of this reason, source code volumes are much reduced. Reduced code volumes imply reduced coding times as well as reduced debugging times. All of this leads to the possibility of implementing a complex system like a database management system in a much shorter time than hitherto was possible and more importantly, with much less effort.

2. In conventional implementations of database management systems, much space and time is taken up by code to perform translations between the program's form of data and the form used for the long term storage medium. In addition, the data type protection offered by the programming languages on its data is often lost across the mapping. Both these problems are solved in PS-algol because both the type and structure of the data is preserved in its persistent heap.

3. Often in a conventional system, to simplify program structure, large portions of data are transferred and translated, prior to any data being used. This results in redundant I/O and data translation in both directions which is a significant overhead if a small part of a large data structure is being read or changed. This does not happen with PS-algol.

4. In a traditional system, every access to database items passes through many layers of subroutine calls and mappings. This increases the execution time considerably. PS-algol avoids such layering costs as the first use of a data item automatically brings it to the heap in the right form for the current program. Thereafter, access is direct.

5. In traditional DBMS implementations, data movement is organised separately from the programs accessing the data. To increase the probability that related data is still in the buffer pool, such systems mainly depend on statistically based disk-buffer caches. Often this is not effective enough. In contrast, PS-algol collects only the actively required data, thus giving better control on the data movement. In addition, as the user behaviour usually has only a slowly shifting focus of interest, it is probable the collection of data on the heap will have a high relevance to the present computation.

The above five factors coupled with the fact that PS-algol is the only implemented persistent algorithmic language, and we wished to evaluate it, were instrumental in deciding to choose it for implementing EFDM.

6.3 Architecture of EFDM

The block-diagrammatic representation of EFDM implementation is shown in figure 6-1. User requests formulated using the EFDM syntax go through the lexical analysis and the syntax analysis phases. The syntax analyser produces a syntax tree for each successfully analysed statement. In addition, it handles all schema modification requests by issuing appropriate calls to the database handler to manipulate the underlying data structures. Other types of requests, i.e. data retrieval and update requests, are passed down to the interpreter. The interpreter traverses the syntax tree formed by the syntax analyser issuing calls to the database handler whenever interaction with the database is required. The database handler provides storage and retrieval facilities for all the data stored in the system. This includes both user data and system data such as meta data.

The lexical analyser, syntax analyser, interpreter, and the database handler are all written in PS-algol language. Appendix 4 gives a list of routines that constitute the database handler. Because the PS-algol language itself takes care of the run-time movement of data between secondary store and main store, the database handler routines manipulate the data structures as though all data is in the main store. Consequently, the complexity of the database handler is much reduced.

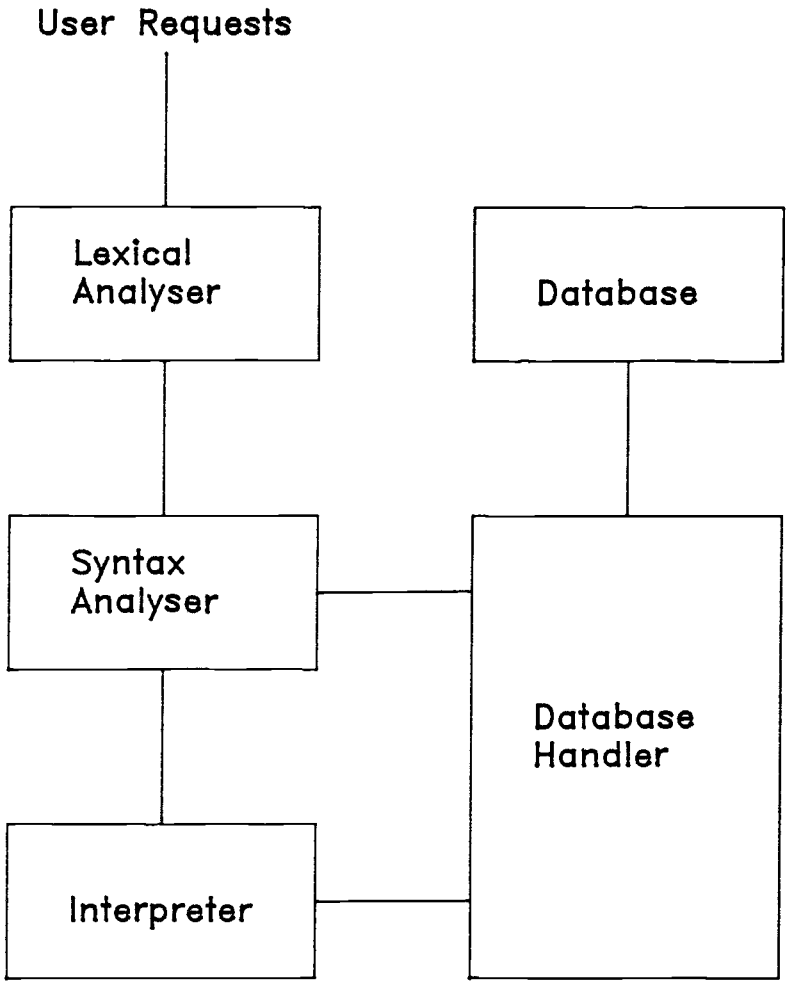

Figure 6–1: *Block Diagram of EFDM*

6.4 Data Structures

The data structures to implement the data model essentially fall into one of the following four categories:

1. Representation of individual entities.
2. Representation of sets of entities.
3. Representation of entity type hierarchy.
4. Representation of functions mapping entities to entities.

It is the choice of data structures for representing functions which is most important. The efficiency of the implementation critically depends on how the function values are stored and how the function values for a given entity are evaluated. In addition, it is imperative that entities are internally identifiable irrespective of how they are identified externally. Therefore, the implementation must have a mechanism which assigns a unique entity identifier to each entity upon creation.

In the PASCAL version, the entity identifiers were integer numbers, assigned and maintained by the system and each function was stored as a binary or n-ary relation consisting of such entity identifies. Though this makes it easy to accommodate schema changes, it suffers from an excessive storage overhead. For example, if there are ten binary functions applicable to an entity , the entity identifier must be stored in all the ten binary relations. In addition, it results in an overly fragmented database and since it is frequently the case that values for multiple functions applied to the same entity are often needed together, this has an adverse effect on performance of the system.

Hence, we have adopted a different implementation strategy in the PS-algol version. Here, the values of all one-argument non-inherited functions applicable to an entity are stored at one place in a record-like structure shown in figure 6-2. To access the values of functions defined on the supertype, a pointer to its immediate super-entity is included in each entity structure. As the language itself ensures that a pointer to each structure is unique, we have chosen the pointer to each entity structure as its entity identifier. This avoids us having to invent and maintain unique entity identifiers. All multi-argument functions are then implemented separately as unnormalised n-ary tables involving such entity identifiers. It may be noted that the designers of ADAPLEX DBMS have also advocated a similar implementation strategy [Chan 82].

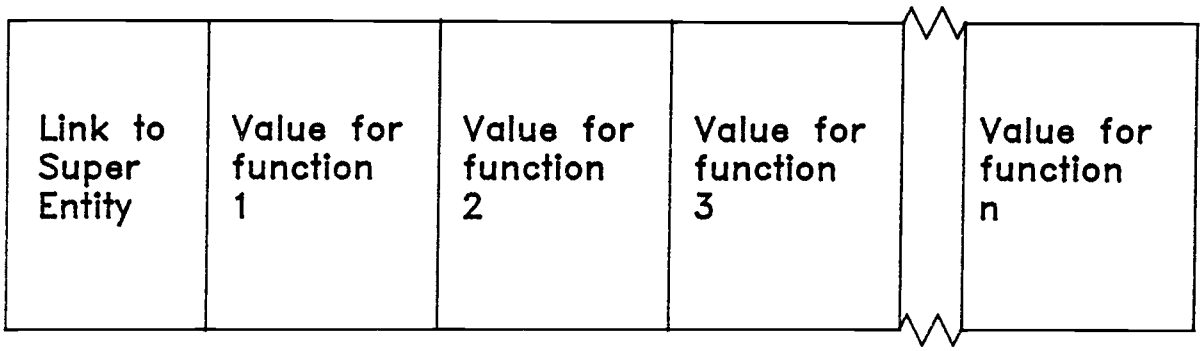

Figure 6–2: Entity Data Structure

The corresponding PS-algol structure for an entity is shown in Appendix C. We represent the group of function values corresponding to an entity as a vector, the upper bound of which varies dynamically as functions get added or dropped. (We can also store the group of function values as a linked list; we rejected this because: 1) linked lists require more storage compared to vectors; 2) accessing individual function values in a linked list is slower compared to vectors; and 3) accommodating schema changes is more complicated in a linked list from compared to vector form.) The elements of such a vector are of type *pntr* as, in general, the EFDM functions yield entities as result. Sets of entities are currently implemented as linked lists of such structures. However, to allow fast access on some function values, it is proposed to provide an indexed structure.

Creating a new entity results in creation of such a structure and adding it to the corresponding linked list. Deletion of an entity results in removing all references to the corresponding structure and finally the entity structure itself from the corresponding linked list.

The above data structures refer to the base functions only. The data for a derived functions is calculated every time it is accessed. To facilitate this, the pointer to the syntax tree corresponding to its definition is stored along with each derived function, The user queries involving derived functions are processed by replacing the references to those functions by the corresponding definition tree. The resulting syntax tree is then executed against the base functions of the database.

To handle the incremental schema changes, the implementation adopts different techniques depending on the number of arguments the function has and the nature of the function. The addition or deletion of a one-argument base function results in creating the new instances of the modified entity structure and copying corresponding values from the old instances to the new instances. On the other hand, the addition or deletion of a multi-argument base function results in the addition or deletion of a table without affecting the existing entity structures. The addition or deletion of derived functions has no effect on the stored entity structures.

The meta data corresponding to a schema is held in two forms: one as PS-algol structures and the other as EFDM functions (see Section 5.2.4). The latter form is necessary so that the users can query the information about the schema using EFDM syntax. The meta data in the form of PS-algol structures is maintained to provide fast access to the structures during run time. Both forms are updated automatically as the users modify the schema.

Compared to the PASCAL version, the implementation scheme adopted in PS-algol version

has many advantages. Firstly, in the scheme we have chosen, the entity identifier is not stored in each function representation and this results in reduced storage requirements. Secondly, the values of one-argument functions for a given entity are provided simply by field dereference instead of table lookup. Thirdly, function composition is achieved by pointer following instead of an expensive "join-like" operation. Both these factors result in faster execution times. Fourthly, several tables must be touched in PASCAL implementation if several functions of an entity participate either in query formulation or in updating. In contrast, PS-algol scheme results in less access to disk as many function values for an entity are brought in simultaneously.

6.5 Problems due to Large Data Volumes

The data structures discussed in the previous section are suitable for moderate volumes of data, typically found in personal data bases. For larger data volumes, the performance will depend more on the "access methods" of the PS-algol run-time system. Since there are no language constructs to influence the efficiency of the run-time system, it may be necessary to adopt different implementation architectures as the data volumes grow.

The mechanisms to accommodate and contain schema changes will also need careful thought as data becomes large. In the current implementation, we modify the data description and repopulate the data everytime a function is added or dropped. Obviously, this will not be acceptable for large scale data. In fact, such a strategy may not even be necessary, as Atkinson et al. point out in their proposal for a new persistent language, NEPAL, [Atkinson 81b]

> In a database describing 300 000 race horses (many of which are dead) a change to the data describing horses need not propagate immediately to all horses, since most of them will never be referenced, or will not be referenced before the next change in the data description.

Hence, it should be possible to accommodate changes without catastrophic consequences rippling through the rest of the system. The solution suggested in the NEPAL language proposal consists of creating new versions of structures with transition rules between the current and the preceding versions. Whenever a structure is referenced, the transitions are applied to yield the appropriate instance to work on. As a side effect of the transition the instances migrate to the later versions. Clearly, some such strategy will be useful in this context also.

Chapter Seven

Applications of EFDM

In this chapter, we present a few practical applications of EFDM. The first application is about organising a database system for personal use. We show how to design, set up, use and maintain the database system using EFDM. The second application concerns the use of EFDM as a prototyping tool during the design of a large database system.

7.1 Organising small personal information systems using EFDM

By small information systems, we mean those systems which are designed for personal use. These are characterised by small amounts of data and processing needs. These systems may or may not have a complex structure, but one distinguishing feature of such systems is that the same person acts as both the designer and the final user of the system. If the database management systems are to be useful for such users, they must meet a number of special requirements. These are as follows:

1. It should be possible to devise a quick implementation of the system as soon as the logical structure has been specified. It should provide data structuring facilities without insisting on the decisions about how to physically store, address, or relate the data elements.

2. It should support a simple design methodology. Users should not be required to learn a large number of disparate concepts, nor should they be required to choose from different design alternatives. The data structures should reflect the semantics of the application. The underlying data model should also guarantee a minimum consistency of data.

3. It should be interactive and easy to use. It should provide an integrated language using which they should be able to create their databases, retrieve information from their databases, modify the structure or contents of their databases or carry out general-purpose computation. Users should not be required to learn different

languages for different activities, i.e., one language for data definition, another for data manipulation, another for general-purpose computation etc.

4. Such systems should be flexible, i.e., users should be able to change the schema without much effort. Users typically wish to implement only a part of the intended system and based on the experience gained, make several modifications by trial and error until they are satisfied with the result. Even after the system comes into use, it may often have to be modified to reflect the changing needs of the user/designer. As the system evolves, users may like to add new elements to the structure, modify existing structures or drop some elements for which they do not have sufficient data.

5. The system must be self-documenting, i.e., users should be able to find out *what is in the system* as easily as they can query about the data.

EFDM is eminently suitable for such users. The particular benefits of EFDM for this purpose are:

1. EFDM enables a quick and economical implementation of a planned system as soon as the logical data structure has been specified.

2. It supports a simple design methodology. The logical design of a database can be easily done using the concepts of the functional data model. In many cases, the structure in terms of functions may be "obvious", i.e., users are not required to carry out complicated mappings from the conceptual structures to the data structures of the model. EFDM structures also captures a large part of the data semantics. Referential integrity is guaranteed by the model itself.

3. It is interactive and simple to use. It provides all the facilities like data definition and data manipulation and general-purpose computation in one language. Users can set up, use, and maintain the database using simple, interactive commands.

4. It is flexible, i.e., structure can be modified interactively at any time. The **declare** and **define** constructs of EFDM provide a natural way to describe increments to the schema. This information can be easily integrated with earlier stored information. Any existing information can be deleted using **drop** without making the database inconsistent.

5. It is self-documenting. An annotated schema is made a part of the database itself and a distinguished entity type called *function* is provided to store the meta data which gets updated automatically as the schema changes. This makes it possible to examine the database structure interactively.

We will now illustrate the use of EFDM for managing the personal collections of reference papers. Every researcher collects a huge number of research publications over a period of

time. Keeping track of the details of individual papers by means of manual methods is often a difficult problem. The following sections describe in step by step how to design, set up, operate, and maintain a database for this application using EFDM.

7.1.1 Designing the Schema

The first step in designing the schema is to prepare a list of entity types corresponding to the objects of interest in the application. Such entity types can often be identified by listing the objects that have some identifiable attributes. For example, in the present application such objects are papers, authors, journals etc. Another means of identifying the entity types is to list the objects that act as units of database update. For example, in the present application the most common database update is likely to be entering the details of a new paper whenever the person receives one. Next step in the design process is to assign meaningful names to the entity types so identified. For the present application, the following constitute the list of entity types: i) *paper* ii) *author*, iii) *topic*, and iv) *journal*.

Next, we identify various properties or attributes of these objects and the relationships between them. The examination of the most likely queries will give an important insight into such properties and relationships. For example, for each paper we would be interested in knowing the title, abstract, list of authors who wrote it, and the journal in which it is published. In addition, we would like to know the list of topics under which it is classified. We model each such attribute and relationship as a function with the proper number of arguments and result. In addition, we examine each function to see whether it is single-valued or multi-valued. For example, there can be more than one author for a paper. Hence the function relating the *paper* and *author* entity types should be multi-valued. Similarly, a paper can be classified in more than one topic. For example, the paper on the TAXIS language can be classified under *query languages*, *semantic data models*, *AI techniques* etc. Hence, the function relating the *paper* and *topic* entity types has to be multi-valued. On the other hand, a paper can appear in one journal only. Hence, the function relating these two entity types must be single-valued.

At this stage, we examine whether some of the objects classified under different entity types share any properties or relationships. If so, we can invent a common supertype for the corresponding entity types, remove all the functions common to those entity types, and associate them with the supertype. In the present example, we have no such relationships. Hence, the final schema for this example is as shown in figure 7-1.

```
declare paper( )              ->> entity
declare title(paper)          ->  string
declare abstract(paper)       ->  string

declare author( )             ->> entity
declare name(author)          ->  string
declare address(author)       ->  string

declare journal( )            ->> entity
declare name(journal)         ->  string
declare details(journal)      ->  string

declare topic( )              ->> entity
declare name(topic)           ->  string

declare author(paper)         ->> author
declare journal(paper)        ->  journal
declare topic(paper)          ->> topic
```

Figure 7-1: Schema for the Paper Database

Next, we examine the schema to see whether we can arrive at some useful derived functions using function composition or special operators like **inverse of**, **transitive of** etc. For example, in the present example, we can define a derived function relating *author* and *journal* entity types as the composition of *journal(paper)* function and the inverse of *author(paper)* function, which gives a list of all journals in which a particular author has published. Figure 7-2 gives a complete list of such derived functions.

```
define papers(author)     ->> inverse of author(paper)
define topics(author)     ->> topic(papers(author))
define journals(author)   ->> journal(papers(author))

define papers(topic)      ->> inverse of topic(paper)
define authors(topic)     ->> author(papers(topic))
define journals(topic)    ->> journal(papers(topic))

define papers(journal)    ->> inverse of journal(paper)
define authors(journal)   ->> author(papers(journal))
define topics(journal)    ->> topic(papers(journal))
```

Figure 7-2: Derived functions for the Paper Database

7.1.2 Setting Up the Database

We can now set up the database system corresponding to the above schema. Each function is interactively entered into the system using **declare** or **define** command. Each of the base functions is then populated with the corresponding data. For this purpose, we can use the update statements provided by EFDM. Again, this can be done interactively. For example, to enter the data about a paper with the title "Database Systems" written by M.P.Atkinson and published in "Journal of Documentation" we use the following update statement:

```
for a new p in paper
let title(p) = "Database Systems"
let author(p) = {the p1 in author such that
                    name(p1)="M.P.Atkinson"}
let journal(p)= the j in journal such that
                  name(j)="Journal of Documentation";
```

The above statement checks that the entities corresponding to the author named M.P.Atkinson and the journal named "Journal of Documentation" do already exist in the database. If any of these entities are not present, the update statement will not be executed and the user is informed about the missing entities. The user can then create new entities corresponding to the unknown author or journal and then submit the above statement again. Thus, the concept of *referential integrity* [Codd 79] is automatically guaranteed by the system.

Note that it is not necessary to input all the data about an entity at the time of its creation itself. For example, to assign the above paper to topics "Data models", and "Natural language interfaces" at a later time, we use the following update statement:

```
for the p in paper such that title(p)="Database Systems"
let topic(p)= {the t1 in topic such that
                  name(t1)="Data models",
                  the t2 in topic such that
                  name(t2)="Natural language interfaces"};
```

If the initial data is rather large, the above method of creating a database may seem rather laborious. In such a case, users can make use of the bulk load facility provided by the system [Kulkarni 83]. This allows the user/designer to create his database from two operating system files, one containing the function declarations and the other containing the data arranged in the form of tables. Once the database is created, it is likely that this system will be updated sporadically, i.e., whenever the person receives a new paper and hence, it will be more convenient to enter the data interactively as demonstrated above.

7.1.3 Using the Database

Once the database is set up, we can pose queries like:

Q1. What are the papers by the author 'x'?

```
for the m in author such that name(m) = "x"
for each p in papers(m) print title(p);
```

Q2. What are the papers by the author 'x' in topic 'y'?

```
for the m in author such that name(m) = "x"
for each p in papers(m) such that
some t in topic(p) has name(t) = "y"
print title(p);
```

Q3. What are the papers by the author 'x' in topic 'y' published in journal 'z'?

```
for the m in author such that name(m) = "x"
for each p in papers(m) such that
   name(journal(p))="z" and
   some t in topic(p) has
   name(t) = "y"
print title(p);
```

Q4. What are the papers in topic 'x' ?

```
for the t in topic such that name(topic) = "x"
for each p in papers(t) print title(p);
```

EFDM also allows one to name the queries and store them in the database itself or to output the results of a query to an operating system file.

7.1.4 Maintaining the Database

EFDM allows one to modify the database structure interactively at any time. For example, if the user wants to assign a grade attribute for each paper, he can do so simply by adding the following function:

```
declare grade(paper) -> integer;
```

and populate that function, either interactively or by using the bulk load facility.

If the user wants to extend the database to hold the information about papers held by his friends, he can simply create another entity type with its own attributes like name of the friend, his room number and the telephone extension etc. and add another function relating

the *paper* type with the newly created entity type. The following function declarations illustrate this:

```
declare friend()          ->> entity
declare name(friend)      ->  string
declare room(friend)      ->  integer
declare phone(friend)     ->  integer
declare held.by(paper)    ->  friend;
```

If the user finds that certain functions are no longer of interest, he can remove them from the database. For example, to drop the *grade* function, he can simply issue a command like,

```
drop grade(paper);
```

and the system will remove the function as well as all the data associated with it. If the removal is going to create any inconsistency in the database, the system will warn the user about it.

The notable feature of this method of managing information is the speed and flexibility provided by EFDM. It took the author a few minutes to design the schema and another few hours to create the database using EFDM, most of which was spent on creating the data file.

7.2 Designing prototypes of large database systems using EFDM

By large database systems we mean those systems which are designed for a community of users. These are characterised by a large amount of data and processing needs. These systems will also have a complex structure. Typically, persons who design such systems will be different from those who will eventually use them. Database management systems suitable for such systems will have to meet different requirements from those discussed in the previous section. Issues like efficiency, sophisticated report generators, statistical analysis packages, security and safe concurrent operations assume higher priority over ease of use or ease of design.

Today there are a number of database management systems available in the market for handling large amounts of data, and satisfying the requirements like efficiency, security and concurrent operations. But the major problem with using these systems is that they are highly inflexible and it is immensely expensive to effect any changes once the database is in operation.

Hence, it is important that the initial specifications are so formulated as to meet all the requirements of the community of users they are intended to serve. However, as we

mentioned above, people who design these systems are different from the people who finally use them, and it is impossible to meet this aim. Serious problems in design arise from false perception of reality resulting from a natural language barrier between the *database naive* user and the *application naive* database designer and the lack of appropriate tools for communication between them.

To bridge this gap, current design methodologies recognise the value of a good requirement gathering/analysing procedure with emphasis on the consultation with the eventual users [Yao 78b]. The need for this is recognised by most authors [Kahn 76, Bubenko 76] but is carried through only by a few. The main problem is that no proper techniques have emerged to aid this phase. Frequently emphasis is put on the development of requirement specification languages and automated analysis procedures [Teichroew 77] which are helpful only when the physical reality has been correctly perceived. Another problem with such methods is the inherent inability to perceive all the needs in the initial stages. The users are often unable to foresee all the implications and interactions in such systems.

Using a prototype of the final system in the design process is an effective solution to this problem. Being able to use even a crude system allows the user and the database designer to converse in more real terms. Prior to a full implementation, the prototype is generated and loaded with small amounts of data. Users then try out the system and identify the problems in the schema design and propose changes. This possibility to be able to *see* the system running at an early stage in a system development project may save a great deal of money as the user gets a chance to re-specify his needs before large resources have been spent on a tailor-made and highly effective implementation. The prototype approach can be tried as a first step to any schema design using the existing facilities of the target system. It is better, however, to use a design tool that has been built specifically for this purpose.

An important requirement for such a tool is that it should be simple to use. Communication with and feedback from the users are crucial to the success of the requirements analysis step. This implies understanding of the description of the database by unsophisticated users. The description cannot therefore be initially in terms of a very complicated and difficult-to-understand data model. Another important requirement of such a tool is that it should possess fast and economical means to effect changes. It is important that the users are able to see the effects of their comments as early as possible.

EFDM is eminently suitable as such a tool. A particular advantage of EFDM for this kind of

application is the simplicity of the data model, and its ability to effect quick schema changes. As demonstrated in the previous section, the schemas designed using this model reflect the structure of the real world in a fairly "obvious" way, and hence such schemas should be easy to understand by unsophisticated users. Another useful feature of EFDM for this kind of applications is the facility it provides for maintaining the consistency of the schema.

An example of such an application is the design of a database system for overall administration of the University of Edinburgh. A possible schema for this application using EFDM is shown in Appendix D. As pointed out earlier, a prototype system corresponding to such a schema can form an effective tool to start the dialogue between database designers and the eventual users to gather the requirements for the actual database system.

In a complicated example such as this, the process of designing the schema itself can be made more systematic by a computer-aided design system. In fact, a database design and integration tool based on another variant of the functional data model [Housel 79] has been reported recently [Yao 82]. The schema design can even be automated on the lines similar to Gerritsen's method [Gerritsen 75], which takes the names of entities, data item descriptions, and the queries that must be supported as input and arrives at a schema for the CODASYL database management systems.

This method of developing prototypes presupposes that an EFDM schema can be correctly translated to the target DBMS data structures. Currently, such target DBMS are restricted to one of the three systems, viz., hierarchical, CODASYL, and the relational systems. Preliminary investigations into translating the functional schemas to the CODASYL and the relational model have been reported by [Shipman 81, Smith 80b, Gray 83, Hepp 83]. The ADAPLEX database management system [Smith 80b], when it becomes available, will be the nearest system to EFDM, although the data model underlying ADAPLEX DBMS is not as powerful as the data model underlying EFDM.

Chapter Eight

Directions for Further Research

This chapter presents some broad directions for extending our work to address some serious problems in the database field that have defied solution so far. We also present a brief discussion of some of the problems that have not been addressed in this thesis.

8.1 Deferring View Updates

An important continuation of this work would be to extend the view mechanism discussed in Chapter 5. As they are implemented at present, updates may be made through a view if update procedures are defined. It is also assumed that all updates done from a view will be applied immediately, i.e., propogated up to the global view. However, in order to conduct experiments of 'what-if' nature on databases appropriate in management and design, we need to have a mechanism which allows updates from a view to appear to have happened within that view but from outside that view they appear not to have happened. This can be achieved easily if we allow the update procedures for view functions to be modified by preceding them with the word defer. We can interpret this as meaning that each time an update is requested, the system records enough data to permit it to be done later. Thus, a user can experiment with the data by making complex and extended updates without interfering with other people's work until he chooses to do so.

Two new operations are then required:

apply *viewname*
reset *viewname*

When a view is created the set of deferred updates is empty. As successive user sessions operate on this view a sequence of updates will be accumulated. The interaction of the operations in the sequence of updates is examined and an equivalent net effect sequence is built. apply causes this accumulated sequence of updates to be applied in the view's context,

and the stored sequence of updates to be set empty. **reset** explicitly abandons a sequence of updates.

We anticipate using a differential file mechanism [Severance 76] to implement this. (Ross [Ross 83] discusses this technique more fully.) In fact, a more subtle implementation is necessary in this context. The reason being that a deferred update may no longer be applicable due to changes in the database since it was recorded. At present we see that as a user problem. People will normally avoid such situations by agreeing on the territory each works, as they do now when teams are involved in design and management. But such a division of tasks is never perfect, and must in any case be checked. When it is checked, the apply is aborted with the sequence left unchanged, and the user informed of the problem. (Note that it is difficult to describe the problem such that the error message is meaningful to the user. This is a research challenge.) When the user has understood the real world problem of two people's work interfering, he will decide as to how to resolve it, and update his view or get someone to modify the main body of data. Subsequent application of the view should not come across the same problem.

Note that this is related to the idea of optimistic concurrency control [Kung 81, Agrawal 82] but it is packaged linguistically to permit experiments with updates to the data, and the possibility of making revisions and retrying when an attempt to apply the view results in a clash with other changes.

8.2 Implementing Transactions

As well as creating, using, and dropping a view, users can also establish a view as the context for subsequent operations. To do this, three statements are necessary:

quote <string>
open <viewname>
close <viewname>

At the start of a session the database holds no quoted passwords, each time **quote** is encountered the value of the string expression which follows is added to the set of quoted passwords.

open then makes the specified view as the current context for operations if there is no constraint on its use, or if one of the passwords which enable its use has been quoted. The only names then available are names provided by that view. Subsequent operations, **define** definitions, queries, updates and new view definitions are interpreted in, and modify the name space of that current view.

The opening and closing of views can be used to define transactions. A transaction starts when a view is opened, and is committed when the **close** statement is encountered. If no **close** is encountered the operations will not be recorded.

8.3 Views with Memory

As they stand in Chapter 5, views are memoryless. **define** and **deduce** statements may extend their name space, **drop** may reduce it, but no other updates on the structure of a view are possible. However, the different views might also want different versions of the meta-data, i.e., base relations, and there might be no intention of merging the data. This can be achieved by allowing **declare** statements to be used within a view. This would then associate base functions (i.e., explicitly stored data) with the view. These would only be usable from this view and its descendants, but could be manipulated in the usual way.

This allows more general variants of the same database. It also allows people to *possess* private data, optionally *related* to the original data, without any commitment to pass on, or to allow access to others. This seems to give an equivalent to the proposed federated architectures [Hiembigner 81, Mcleod 80], except that there may be many layers of federation.

If every use of a view starts with an explicit **open** command and ends with an explicit **close** command, we then have a mechanism for providing transient data. **declare** statements occurring in a view in the current session, which is never closed will store data during the session, but the data will vanish at the end of it. Thus we have a mechanism for differentiating transient and persistent data.

Its one drawback seems to be the laboriousness of defining views so that all the external names wanted are imported. The alternative is to introduce another concept for access control which seems less attractive. We believe the problem is best overcome by having a good cut and paste screen editor at the user interface.

Note that with the one concept of view, and the avoidance of exceptions as to where **declare**, define etc. may be written, we have in one uniform and simple language achieved a number of effects, which have hitherto required different notations and treatment and have hence either been omitted or led to a more complex language. These effects are: i) views, ii) meta-data edits, iii) 'what-if' experimental grouping of operations on the data, iv) protected data spaces, v) federations, vi) identification of persistence. Although it looks feasible, the implementation of this approach has yet to be researched.

8.4 Query Optimisation Issues

High level query languages such as the one provided by EFDM allow one to write queries whose execution time can be reduced greatly if the query language processor rephrases the query before executing it. Such improvements are called *optimisations* and a great deal of literature has grown around this concept, mainly in the context of relational query languages [Astrahan 76, Smith 75, Hall 76b, Wong 76].

Ullman [Ullman 82] categorises the optimisation ideas into two groups. One group consists of algebraic manipulations - transformations that are applied with little or no concern as to how the data model constructs are stored. The second group consists of strategies to take advantage of the storage of the data model constructs such as indices etc. In addition, Hall [Hall 76b] describes a third type of technique based on *delayed evaluation* of algebraic expressions.

The only work on optimisation in the context of functional data model has been reported by Buneman et al. [Buneman 82]. Their implementation of the functional query language, FQL, exploits the *lazy evaluation* technique [Friedman 76, Henderson 76], which is somewhat similar to the delayed evaluation technique mentioned above. This technique achieves a reduction in both the main storage space and the number of accesses to the secondary storage by delaying the evaluation of expressions until their values are needed. For example, consider a query that finds the names of employees who are under 30 years of age and are paid more than the average salary for all employees. We can expressing this query in EFDM as follows:

```
for each e in employee such that
  age(e) < 30 and
  salary(e) > average(salary(e1)
              over e1 in employee)
print name(e);
```

Under the lazy evaluation technique, the average salary is calculated only if some employee is under 30, otherwise the average salary is not calculated at all.

Buneman et al. also report on an algebraic optimisation technique based on the compile-time detection of "constant-valued" sub-expressions. This *do-loop* optimisation is well known to compiler writers, and consists of moving expressions in hierarchic iterative structures outward as far as possible. Thus, in the statement

```
for each x in person
 for each y in person such that
 mother(y) = sister(x)
print name(x),name(y)
```

the expression *sister(x)* can be considered as a constant expression for the inner loop and thus is a candidate for moving it outside that loop.

There are basically two means for this: the first is a pre-pass over the query which identifies constant expressions and moves them creating temporary variables as needed. In the second method, each expression, when evaluated, replaces itself on the expression tree. Before every iteration, those expressions dependent upon the iteration variable get restored in the tree. Thus constants, relative to that particular iterative block, get evaluated only once the first time through. This is the technique used by Buneman et al. Similar schemes in which expressions are replaced by their values have been described by Henderson and Morris [Henderson 76] and Turner [Turner 79].

A second algebraic optimisation technique deals with ordering of simultaneous comparison expressions in the query formulations involving objects that satisfy multiple relationship constraints [Greenfeld 74]. The problem concerns finding an optimum evaluation order. A simple example is that of finding the sons of Bill:

```
for each p in person such that
  name(father(p)) = "Bill" and
  sex(p) = "male"
print name(p);
```

The order in which the clauses are processed may have a marked effect on performance, since it is expected that there will be many more males than children of Bill. But the analysis is much more complex, depending in part upon the asymmetries of access.

Another algebraic optimisation technique, identified but not implemented by Buneman et al. [Buneman 82], deals with the syntactic manipulation of expressions. This involves the transformation of a query into another, more efficient representation. For example, consider the query that prints the names of those employees who earn more than the average salary of the employees in their department:

```
for each e in employee such that
 salary(e) > average(salary(e1) over e1 in employee
                     such that dept(e) = dept(e1))
print name(e);
```

Straight-forward evaluation of this query results in recomputing the average for each employee. However, the evaluation can be made more efficient if the query is subjected to a set of syntactic transformations leading to:

```
for each d in department
for each e in employees(d) such that
 salary(e) > average(salary(e1) over e1 in employees(d))
print name(e)
```

which will prevent recomputation of average for each employee because it can now be recognised as a constant subexpression in the inner loop.

8.5 Derived Data Control

Derived data control means the materialisation of derived data [Koenig 81]. In the current implementation, derived data is recalculated whenever it is referenced. All references to derived functions used in a query are replaced by the corresponding function definitions and the resulting query is then executed against the base functions of the schema. Relational systems that provide derived relations or views of the relational model have also based the materialisation of derived data on a similar recalculation strategy. Both System R [Astrahan 76] and INGRES [Stonebraker 76] store only the view definition.

Such a strategy for derived data control may not be acceptable when accesses to derived functions are frequent or the required computations are costly. Two techniques have appeared in the literature which promise a better performance. The technique proposed in PRTV [Todd 76] improves on the *recalculate when accessed* approach by storing a newly recalculated derived relation R until a subsequent modification is made to the base relations in terms of which R is defined, at which time the previously calculated value is destroyed. References to R prior to such a modification to base relations are directed to the previously calculated value of R, thereby saving the cost of recalculation. While references to R following such a modification are resolved by recalculating R and saving its value for future references. The actual improvement from this approach depends upon the usage patterns of the database, in particular the ratio of derived relation accesses to base relation modifications.

The technique proposed by Koenig and Paige [Koenig 81] improves on the PRTV approach in some cases by "maintaining" the stored value of a derived variable V whenever any definitional parameter of V is modified instead of destroying it. That is, V is incrementally updated to reflect changes to the definitional parameters of V. Thus, all references to the derived variable V may be directed to its currently stored value. This approach will be more efficient compared to the PRTV technique only if the execution costs of the code required to maintain a derived variable V is less than the cost of the periodic recalculations of V.

Investigations of similar techniques in the context of functional systems is a serious research issue.

8.6 The Search for a Uniform Language

As we mentioned in Chapter 5, it is important for a database management system to provide a language which in one continuous spectrum would provide simple general queries, updates, and the power of a general purpose programming language. FQL [Buneman 79] has already demonstrated how powerful this can be and our language has comparable capabilities. The main difference is that FQL adopts the formalism of sequences while our language is based on sets.

The main advantage of sets over sequences is that most of the database operations can be expressed in terms of set operations. For example, adding a *person* entity into an entity type or a relationship can be specified as a set union operation. Similarly, removal of an object from the database can be specified as a set difference operation. However, the main problem with sets is that there is no notion of repetetion in its members, as in sequences. Sometimes the repetetion of an object as the output of an operation is meaningful. For example, if we want to calculate the sum of the salaries of all employees, we need to allow the repetetion of the same number. Sequences are useful in this context. It is an important research issue to decide whether both sets and sequences are needed in a language, or given one, whether the other can be provided easily.

Another issue in the language design is that of base types. Databases designed to model abstractions of real-world systems do not just deal with integers or strings. Attributes in a database, for example, weight or salary may both well be measured by integer values. Nevertheless, they are interpreted in different ways and should therefore be treated as different types. The weak type checking and compatibility rules of most programming languages are not adequate for database problems. For example, in Pascal [Wirth 71] and Euclid [Lampson 77] the following two types are the same, hence, compatible

```
type height = 1..100
type weight = 1..100
```

Consequently, meaningless operations, such as comparing height and weight, cannot be detected automatically by considering the operands of comparison. In the database world, some work has been done in the context of relational languages [Schmidt 78, Brodie 80, McLeod 76]. Providing such base types in functional languages like DAPLEX is an important area of research.

The concept of type hierarchy also needs further investigation. At present, an entity type

can have only one supertype. However, there may be cases when an entity type has more than one supertype. For example, a customer can be a person, a company, or a government agency. Though this can be handled as a derived entity type using set union operator, the semantics of such types needs to be investigated. Even within a strict type hierarchy, implicit property inheritance from supertypes to subtypes, as it happens in the present implementation, may not be acceptable in some cases. It may be necessary to specify explicitly which properties of the supertype are to be inherited by a subtype.

However, the main challenge for functional data languages arises from adequately modelling and packaging update and I/O operations without disharmony. Essentially the idea of explicitly controlling the implementation or propagation of updates is antithetical to pure functional programming. Yet organising the storage of data and hence updates is a dominant computing activity central to databases. It is necessary to investigate whether the set of system-provided functions can be expanded to include I/O operations and whether a convenient and consistent language can be developed to satisfy the range of programming needs using the functional parameter handling mechanisms. It is also necessary to investigate whether the present syntactic form of the language can be retained or whether it is necessary to adopt a form closer to familiar programming styles.

Chapter Nine

Conclusions

In this chapter, we present a brief summary of the research work reported in this thesis. We also list what we feel are the important lessons learned from it.

9.1 Summary of the Research

Conceptual data modelling forms an important stage in the design of a database system. Data modelling has been an active area of research for the past few years. What prompted this activity is the fact that the present generation of database management systems are based on concepts that are far from being 'natural'. The user needs to map his thought processes to the data structures provided by these models in complex ways, and the design process itself is based on ad-hoc methodologies.

To overcome these problems, data modelling researchers have proposed a number of new data models. Based on this research, we identify a set of concepts that are 'natural' for modelling data (chapter 2). Broadly, these concepts are based on the set of entities corresponding to the objects on the real world, a classification of those entities into entity types, the identification of attributes of and relationships among such entity types, and a set of constraints that help to maintain the data consistency. There is also an increasing recognition of the importance of organising types into a type hierarchy, and the rules or inference mechanisms, ideas borrowed from the artificial intelligence field.

Chapter 2 also lists a set of criteria to assess the merits of different data models. To be able to act as a tool to design conceptual schemas, we require a data model to:

1. be based on object orientation;

2. be semantically expressive;
3. provide a simple database design methodology;
4. provide constructs that have a neutral interpretation;
5. have a well defined set of operations;
6. provide facilities to specify constraints;
7. provide facilities to accommodate derived data;
8. be free from physical considerations;
9. provide constructs that can evolve gracefully;
10. be simple and easy to use.

Against this set of criteria, we assess the conventional data modelling approaches based on the hierarchical, CODASYL/DBTG, and the relational data models. We identify a number of significant problems with these approaches. (Chapter 2.) These are as follows:

1. They lack object orientation.
2. Semantically, they are insufficiently expressive.
3. It is difficult to design databases.
4. The constructs of these models lack neutrality of interpretation.
5. They fail to accommodate derived data.
6. It is difficult to accommodate database evolution.

In addition to these, CODASYL and the hierarchical systems suffer from the following additional problems:

1. They tend to provide navigational, element-at-a-time operations.
2. They lack proper constructs to specify explicit constraints.
3. They do not provide a complete separation between logical and physical aspects.
4. They are difficult to understand.

In chapter 3, we provide a brief description of twenty-one semantic data models and follow

it up with a rough assessment based on the set of criteria mentioned above. Out of these models, we select the functional data models for further investigation because:

1. They can provide a semantically rich modelling power by removing the distinction between programs and data.

2. They can provide high-level data languages based on function application.

3. They can provide data manipulation and general-purpose computation in one language.

4. They are based on solid mathematical foundations.

5. They can support most of the conventional data models, which means that the functional data models can be used to provide a global schema in a heterogeneous distributed database system.

6. They are easy to implement. In fact, these models have the inherent ability to provide highly efficient implementations by exploiting parallel processing architectures.

A number of functional data models with varying capabilities and associated data languages have been reported in the literature. We choose the one proposed by Shipman as it is based on a simple set of concepts. It is object based and incorporates the idea of type hierarchy. The definition of both the schema and its extension is around *atomic* units of information, i.e., irreducible relationships. It accommodates the concept of derived data naturally by means of intensionally defined functions, which are manipulated in the same way as those defined extensionally. In addition, it provides a well defined set of operations on the data it models. Chapter 4 describes this model and the associated data language in greater detail.

Chapter 5 provides the logical details of our implementation based on this model. This chapter also provides a critical examination of the model and the language proposed by Shipman, and describes a series of changes to the model, as well as improvements and extensions to the language. We show how a simple device to control the visible name space forms the basis for providing multiple views of a database. We also introduce new features like specification of integrity constraints and describe their use.

We were greatly helped in the implementation task by the availability of the persistent algorithmic language, PS-algol. Chapter 6 gives a brief description of the PS-algol language and explains how it helps in implementing database systems. The implementation details are covered in Chapter 6.

Chapter 7 discusses how the implementation can be used as a personal database management system for managing moderate volumes of data and as a dynamically reorganising schema design tool for prototyping of large database systems. A particular attraction of EFDM is its ability to permit users to design their schemas in an incremental fashion, checking for consistency of the schema at each stage. This way users are forced to examine their assumptions and the way the reality is abstracted before committing themselves for a particular database implementation. The resulting schema can also serve as a complete documentation of the contents and meaning of the database it is modelling. To facilitate this, the annotated model of the schema itself is made part of the database.

Chapter 8 lists a number of directions in which the research reported here can be extended. For example, it discusses how the same concept of view promises simple provision of experiments, version management, federations and nested transactions. It also discusses few issues connected with the query optimisation and the derived data materialisation. Chapter 8 also lists some of the extensions to our language, especially the issues connected with base types and type hierarchy.

9.2 Contributions of the Research

The principal contribution of this research work is the critical discussion of semantic data models in general, and of the functional data model proposed by Shipman, in particular. This work examines both the data model and the language proposed by Shipman and discusses a series of changes to the model and improvements to the language, culminating in an interactive system suitable for implementing personal information systems or as a dynamically re-organising schema design tool.

This work clearly demonstrates the fact that the functional approach to data modelling, in addition to being amenable to computer processing, can also accommodate the semantic data modelling concepts, concepts such as entity orientation, type hierarchy, and derived data. The use of the last concept to model programs as data, though not novel in other fields, is certainly novel in the database context. The presence of these derived functions helps to simplify the formulation of user queries to the databases and to provide multiple user views. Our work also shows how the functional data model and the language implemented in the present work is neatly extensible to handle integrated data languages, views, experiments and federations.

This work also demonstrates the efficacy of the use of persistent algorithmic languages like PS-algol for implementing database management systems. The database researchers have so far been deterred by the lack of proper tools to carry out implementations of novel data models in order to assess them. The use of conventional languages for implementation work implies duplicating the complex storage management software for each implementation task. It is hoped that future languages will increasingly recognise the merit of treating the persistence as an orthogonal property of the data.

Our work clearly demonstrates that the functional data model with advanced modelling constructs is indeed implementable. Other semantic data modelling approaches have also succeeded in providing semantically rich modelling constructs, but it has not been easy to see how they can be implemented. As a result, there is an impression that semantic data models, in general, are too abstract and hence they can only be used to design *enterprise descriptions* solely to aid human understanding [Chen 76]. Some of them are so complex that, even if they can be implemented, they are clearly unsuitable for the majority of the users. Codd, in his proposal of the Extended Relational Data Model, RM/T [Codd 79], even suggests that *the extensions in RM/T are primarily intended for the minority consisting of database designers and sophisticated users*. It is hoped that the present work will dispel such impressions and demonstrate that a semantic data model with advanced data modelling constructs can be implemented and can be used by both sophisticated and non-sophisticated users.

To date, the shortcomings of the data models underlying present generation of database management systems has been discussed only at the academic level. It is hoped that the use of EFDM will generate sufficient interest in semantic data modelling concepts, and convince the users of the shortcomings of classical data models. If there is to be a wide spread use of the semantic data modelling concepts, it is imperative that systems incorporating such concepts and simple enough to be used by a large user community are made available widely.

Appendix A
DAPLEX Syntactic Specification

```
program = {statement}.
statement = declarative | imperative
declarative = declare funspec ("->" | "->>") expr[order]|
   define funspec ("->" | "->>")
     (expr |
      inverse of funcspec |
      transitive of expr |
      compound of tuple |
      (intersection|union) of expr {","expr}|
      difference of expr "," expr|
     )[order]
   define constraint funcspec "->" boolean|
   define trigger funcspec "->" boolean imperative|
   perform update using imperative.
funcspec = funcid "(" [tuple] ")".
tuple = expr {","expr}.
expr = set|singleton.
set = mvfuncall|typeid|
   "{"[singleton {","singleton}] "}"|
   set such that pred|
   set comp (singleton|quant set)|
   identifier in set|expr as typeid|
   "(" set ")"|gpsingleton.
singleton = constant|vblid|svfuncall|aggcall|
   pred|the set|a new typeid|
   the set(preceding|following) singleton|
   "(" singleton ")"|gpsingleton.
svfuncall = funcall.
mvfuncall = funcall.
funcall = funcid "(" [tuple]")".
aggcall = aggid "(" bag ")".
bag = expr|singleton over tuple.
pred = boolean|
   for (singleton|quant set) pred|
   (singleton|quant set) comp (singleton|quant set)|
   quant set (exist|exists).
comp = ">"|"<"|"="|eq|ne|lt|gt|le|ge.
quant = some|every|no|
   (at (least|most)|exactly) integer.
integer = singleton.
string = singleton.
boolean = singleton.
constant = int|str|bool.
int = digit {digit}.
str = """ character {character}""".
bool = true|false.
imperative = forloop|update|gpimperative.
```

```
forloop = for each set [order] imperative|
    for singleton imperative.
order = in order
    {by [(ascending|descending) singleton].
update = let svfuncall "=" singleton|
    (let|include|exclude) mvfuncall "=" expr|
    insert mvfuncall "=" (singleton|set [order])
        (preceding|following) singleton.
vblid = identifier.
typeid = identifier.
funcid = identifier.
aggid = identifier.
```

NOTE: The syntax specification appearing in this and the next appendix is described in terms of the syntax specification language proposed by Wirth [Wirth 77]. Terminals are represented by bold-face words, and by non-alphanumeric symbols enclosed in quote marks (and, if a quote mark appears as a terminal itself, it is written twice). Syntactic categories are represented by lower case italic words. Repetetion is denoted by curly brackets, i.e., $\{\alpha\}$ stands for $\epsilon|\alpha|\alpha\alpha|\alpha\alpha\alpha|\ldots$. Optionality is expressed by square brackets, i.e., $[\alpha]$ stands for $\alpha|\epsilon$. Parantheses merely serve for grouping, e.g., (a|b)c stands for ac|bc.

Appendix B

EFDM Syntactic Specification

```
command = imperative|
   declare funspec ("->" | "->>") typeid|
   define funspec ("->" | "->>") fundef|
   program programid is imperative|
   output programid fileid|
   view viewid is
   {deduce funcspec ("->"|"->>") typeid using
    fundef} end.
   drop (funspec|programid|viewid)|
   load|
   programid.
imperative = for each set imperative|
   for singleton imperative|
   update|print stuple.
set = vblid in set1
   [such that predicate] [as typeid]
set1 = mvfuncall|typeid|"{"stuple"}"|
   "(" set {(union|intersection|difference) set} ")".
singleton = exp1 {or exp1}
exp1 = exp2 {and exp2}
exp2 = [not] exp3
exp3 = exp4 [compop exp4]
exp4 = [prefix] exp5 {addop exp5}
exp5 = exp6 {mulop exp6}
exp6 = exp7 [as typeid]
exp7 = constant|vblid|svfuncall|aggcall
   the set|a new typeid|
   quant set (has|have) predicate|
   "(" singleton ")".
svfuncall = funcid "(" stuple ")".
mvfuncall = funcid "(" mtuple ")".
stuple = singleton {","singleton}.
mtuple = expr {","expr}.
expr = set|singleton.
aggcall = (count|max|min) "(" set ")"|
   (total|average) "(" singleton over mtuple ")".
update = let funcall "=" expr|
    include (funcall|typeid) "=" set|
    exclude (funcall|typeid) "=" set|
    delete singleton.
funcall = funcid "(" stuple ")".
fundef =
    (expr |
      inverse of funcspec |
      transitive of expr |
      compound of tuple.
     ).
```

```
funcspec = funcid "(" [arglist] ")".
arglist = typeid {","typeid}.
compop = ">"|"<"|"="|">="|"<="|"~".
quant = some|all|no|
    (at (least|most)|exactly) integer.
integer = singleton.
predicate = singleton.
constant = int|str|bool.
int = digit {digit}.
str = '"' character {character}'"'.
bool = true|false.
vblid = identifier.
typeid = identifier.
funcid = identifier.
programid = identifier.
viewid = identifier.
identifier = letter {(letter|digit|".")}.
prefix = "+"|"-".
addop = "+"|"-"|"++".
mulop = "*"|"/"|rem.
```

Appendix C
Functional Specification of EFDM

1. Data Structures

```
1) structure str.struct(string str)
2) structure int.struct(int val)
3) structure bool.struct(bool bval)
4) structure list.struct(pntr head,tail)
5) structure fun.struct(string fname,type,status,ftext;
                        int nargs,fnno;
                        pntr args,result,uses,used,fnson,
                             cons,fval,mptr,nextfn)
```

This structure is used to represent functions of the data model. The various components of this structure have the following meaning:

```
fname  -- name of the function
type   -- "svfun" for single-valued functions
          "mvfun" for multivalued functions
status -- "base" for base functions
          "derived" for derived functions
          "system" for those functions used by the system to
           store meta data, etc.
ftext  -- text of the function declaration
nargs  -- number of arguments the function has
fnno   -- the total number of base, non-inherited,
          one-argument functions if the function is
          an entity type declaration,
          a number indicating the index into the
          vector of function values if the function
          base one-argument function,
          0, otherwise.
args   -- arguments of the function arranged as a list structure.
          must have the type list.struct where the individual
          elements of list have the type fun.struct.
          Elements in the list must be ordered according to the
          order of arguments in function.
res    -- result of the function with the type fun.struct.
uses   -- list of functions used in the definition of this function;
          must be of of type list.struct where the individual
          elements must be of type fun.struct; order is not
          important.
used   -- list of functions that use this function in their
          definition. Same comments as above.
fnson  -- nil if functions has arguments, otherwise a list of
          functions that have this entity type as
          one of the arguments. must be of type list.struct.
```

order not important.

6) **structure** ent.struct(pntr otype, super, nextobj; *pntr ftup)

This structure is used to represent data model entities. Description of the components of this structure is as follows:

otype -- pointer to the entity type of the entity. must point to an instance of fun.struct.
super -- pointer to another entity belonging to the immediate supertype of the entity type of the entity. must point to an instance of ent.struct.
nextobj-- ponter to the next entity belonging to the same entity type. must point to an instance of ent.struct.
ftup -- a vector containing values for one-argument, non-inherited functions. Individual elements may be of type str.struct, int.struct, bool.struct, or ent.struct for single-valued functions, and list.struct for multi-valued functions.

7) **structure** mvfunval.struct(pntr avallist, rvallist, nextval)

This structure is used to represent function values for multiargument functions. Description of the components of this structure is as follows:

avallist-- list of argument entities. must be of type list.struct with individual elements ordered according to the argument position.
rvallist-- pointer to an instance of str.struct or int.struct or bool.struct or ent.struct for single-valued functions and an instance of list.struct for multi-valued functions. order not important.

2. Database Handler Routines

1) **procedure** appendfn(string name, type, stat; pntr args, res -> pntr)

2) **procedure** deletefn(pntr fnp)

3) **procedure** create.entity(pntr fnp)

4) **procedure** include.entity(pntr ent, fnp)

5) **procedure** exclude.entity(pntr ent, fnp)

6) **procedure** appendfnval(pntr fnp, argvallist, resvallist)

7) **procedure** addfnval(pntr fnp, argvallist, resvallist)

8) **procedure** deletefnval(pntr fnp, argvallist, resvallist)

9) **procedure** getfn(string name; pntr args -> pntr)

10) **procedure** getfnval(pntr fnp, argvallist -> pntr)

Appendix D

Schema for University Administration

```
declare organisation()          ->> entity
declare university()            ->> organisation
declare admin.office()          ->> organisation
declare service.unit()          ->> organisation
declare faculty()               ->> organisation
declare department()            ->> organisation

declare name(organisation)      -> string
declare address(organisation)   -> address
declare phone(organisation)     ->> phone

[Administering faculties]

declare degree()                ->> entity
declare diploma()               ->> entity
declare certificate()           ->> entity
declare subject.type()          ->> entity
declare subject()               ->> entity
declare course()                ->> entity

declare dean(faculty)           -> teaching.staff
declare associate.dean(faculty) ->> teaching.staff
declare sao(faculty)            -> nonteaching.staff
declare secretary(faculty)      -> nonteaching.staff
declare department(faculty)     ->> department
declare degree(faculty)         ->> degree
declare diploma(faculty)        ->> diploma
declare certificate(faculty)    ->> certificate
declare subjecttypes(faculty)   ->> subject.type
declare subject(faculty)        ->> subject

declare name(subject.type)      -> string
declare directors.of.studies(subject.type) ->> teaching.staff

declare name(subject)           -> string
declare subjecttype(subject)    -> subject.type

declare name(degree)            -> string
declare prereq(degree)          ->> degree
define  offer(degree)           ->> inverse of degree(faculty)
declare courseyear(degree)      ->> course.year
declare subject(degree)         ->> subject

[Administering departments]

declare course.year()           ->> entity
declare term()                  ->> entity
```

```
declare task()                    ->> entity
declare assignment()               ->> task
declare exam()                     ->> task
declare project()                 ->> entity
declare document()                 ->> entity

declare head(department)           -> teaching.staff
define  faculty(department)       ->> inverse of department(faculty)
declare course(department)         ->> course
declare project(department)       ->> project

declare tasks(course.year)        ->> task
declare name(course.year)          -> string
declare course(course.year)       ->> course
declare coordinator(course.year) -> teaching.staff
declare exams(course.year)        ->> exam
declare assignments(course.year)->> assignment

declare weight(task)              -> task.weight
declare sets.task(task)           -> teaching.staff
declare marks.task(task)          ->> teaching.staff

declare date.set(assignment)      -> date
declare deadline(assignment)      -> date

declare date(exam)                 -> date
declare term(exam)                 -> term
declare held.at(exam)              -> room

declare name(project)              -> string
declare startdate(project)        -> date
declare deadline(project)         -> date
declare budget(project)            -> money
declare funding(project)          ->> organisation
declare report(project)           ->> document

declare name(document)             -> string
declare issuedate(document)       -> date
declare abstract(document)        -> string
declare author(document)          ->> person
```

[Administering personnel]

```
declare person()                  ->> entity
declare student()                 ->> person
declare ugmsc.student()           ->> student
declare science.ug.student()      ->> ugmsc.student
declare research.student()        ->> student
declare staff()                   ->> person
declare teaching.staff()          ->> staff
declare nonteaching.staff()       ->> staff
define  tutor()                   ->> teaching.staff UNION
                                      research.student
define  demonstrator()            ->> teaching.staff UNION student

declare surname(person)            -> string
declare firstname(person)         -> string
declare middlenames(person)       ->> string
declare sex(person)                -> sex
declare birthday(person)          -> date
declare number(person)             -> integer
declare extension(person)         -> phone.extension
declare home.address(person)      ->> address
declare marital.status(person) -> marital.status
declare spouse(person)             -> person
declare qualification(person)  ->> degree
declare country.of.birth(person) -> country
declare nationality(person)       -> country
declare permanent.domicile(person) -> country
```

[Administering staff]

```
declare works.for(staff)                -> organisation
declare room(staff)                      -> room
declare designation(staff)              -> designation
declare current.appt.date(staff)        -> date
declare increment.date(staff)           -> date
declare scale.code(staff)               -> scale.code
declare annual.salary(staff)            -> money
declare other.payments(staff)           -> money
declare date.of.review(staff)           -> date
declare income.tax.month(staff)         -> month
declare income.tax.code(staff)          -> income.tax.code
declare account.number(staff)           -> account.number
declare national.insurance.no(staff)-> integer
declare union.affiliation(staff)     -> union
declare union.subscription(staff)    -> money
declare staff.club.member.no(staff) -> integer
declare staff.club.subs(staff)          -> money
declare retirement.age(staff)           -> year
declare employment.history(staff)    ->> employ.history
```

[Administering students]

```
declare enrollment.number(student)  -> eno
declare term.residence(student)      -> address
declare year.of.first.entry(student) -> year
declare year.of.course.entry(student) -> monthyear
declare year.of.course(student)      -> course.year
declare date.of.enrollment(student) -> date
declare status(student)              -> student.status
declare fee(student)                 -> money
declare type.of fee(student)         -> fee.type
declare from.university(student)     -> university
declare grant.awarding.body(student) ->> grant.body
declare last.full.time.school(student) -> school.detail
declare other.institutes.attended(student) ->> school.detail
declare previous.school.results(student) ->> school.results
declare degree.awarded(student)      -> degree

declare course(ugmsc.student)          ->> course
declare tutorial(ugmsc.student)      -> tutorial
declare grade(ugmsc.student,task)    -> marks
declare final.result(ugmsc.student,course.year) ->  result

declare lab.session(science.ug.student) -> lab.session

declare supervisor(research.student) ->> staff
declare period.of.study(research.student) -> integer
declare date.of.starting.study(research.student) -> date
declare research.costs.charged(research.student) -> money
declare research.costs.collected(research.student) -> money
```

[Administering teaching]

```
declare teaching.event()          ->> entity
declare lecture()                 ->> teaching.unit
declare tutorial()                ->> teaching.unit
declare lab.session()             ->> teaching.unit

declare time(teaching.event) -> time
declare day(teaching.event) -> day
declare room(teaching.event) -> room

declare demonstrator(lab.session) -> demonstrator

declare tutor(tutorial) -> tutor
declare parity(tutorial) -> parity

declare lecturer(lecture) -> teaching.staff
```

[Miscellaneous]

```
declare committee()                 ->> entity

declare member(committee)           ->> person
declare convener(committee)         ->> person

declare date.of.entry(school.detail)  -> date
declare date.of.leaving(school.detail)-> date
declare subject(school.detail)        ->> subject

declare board(school.result)          -> organisation
declare date(school.result)           -> monthyear
declare grade(school.result)          -> degree

declare sex()                       ->> entity
declare address()                   ->> entity
declare text(address)               -> string
declare building()                  ->> entity
declare name(building)              -> string
declare room()                      ->> entity
declare number(room)                -> integer
declare extension(room)             -> integer
declare building(room)              -> building
declare country()                   ->> entity
declare name(country)               -> string
declare designation()               ->> entity
declare marital.status()            ->> entity
declare money()                     ->> entity
declare union()                     ->> entity
declare month()                     ->> entity
declare year()                      ->> entity
declare monthyear()                 ->> entity
declare income.tax.code()           ->> entity
declare account.number()            ->> entity
declare scale.code()                ->> entity
declare student.status()            ->> entity [part time/full time]
declare fee.type()                  ->> entity [home, overseas etc.]
declare grant.body()                ->> entity
declare phone()                     ->> entity
declare phone.extension()           ->> entity
declare task.weight()               ->> entity
declare employ.history()            ->> entity
declare eno()                       ->> entity
declare marks()                     ->> entity
declare result()                    ->> entity
declare parity()                    ->> entity [odd/even]
```

References

[Abrial 74] Abrial, J.R.
Data Semantics.
In Klimbie, J.W and Koffman K.L. (editor), *Database Management*, . North-Holland, 1974.

[Agrawal 82] Agrawal, R. and DeWitt, D. J.
Further optimism in optimistic methods of concurrency control.
Technical Report #470, Computer Science Department, University of Wisconsin-Madison, 1982.

[ANSI 75] ANSI/X3/SPARC Study Group on Database Management Systems.
Interim Report.
FDT 13(2), 1975.

[ANSI 78] Tsichritzis, D.C., and Klug, A.
ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Database Management Systems.
Information Systems 3, 1978.

[Arvind 78] Arvind, K.P. and Gostelow, W.P.
An Asynchronous Programming Language and Computing Machine.
Technical Report, University of California at Irvine, 1978.

[Astrahan 76] Astrahan, M. M. *et al.*
System R: relational approach to database management.
ACM Transactions on Database Systems 1:97-137, June, 1976.

[Atkinson 78] Atkinson, M. P.
Programming languages and databases.
Technical Report CSR-26-78, Computer Science Department, University of Edinburgh, 1978.
Also in *Proceedings of the Fourth International Conference on Very Large Databases* (1978).

[Atkinson 79] Atkinson, M.P.
Database Systems.
Journal of Documentation 35(1):49-91, March, 1979.

[Atkinson 81a] Atkinson, M.P., Chisholm, K.J. and Cockshott, W.P.
PS-algol: an Algol with a Persistent Heap.
ACM SIGPLAN Notices 17(7), July, 1981.

[Atkinson 81b] Atkinson, M.P (editor).
Data Bases.
In *Infotech State of the Art Report. Database*, . Pergamon Infotech, 1981.

[Atkinson 83a] Atkinson, M.P., Chisholm, K.J., Cockshott, W.P. and Marshall, R.M.
Algorithms for a Persistent Heap.
Software Practice and Experience 13(7), March, 1983.

[Atkinson 83b] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. and Morrison, R.
An Approach to Persistent Programming.
Computer Journal 26, 1983.

[Atkinson 83c] Atkinson, M.P. Chisholm, K.J. and Cockshott, W.P.
CMS-A Chunk Management System.
Software-Practice and Experience 13:273-285, 1983.

[Azmoodeh 82] Azmoodeh, M. and Lavington, S.H.
A Scheme for Representing Information and its Implications for Storage Technology.
Technical Report IFS/2/82, University of Manchester, 1982.

[Bachman 77] Bachman, C.W. and Daya, M.
The Role Concept in Database Models.
In *Proceedings of International Conference on Very Large Data Bases.* ACM, 1977.

[Backus 78] Backus, J.
Can Programming be Liberated from the von Neumann style? A functional Style and its algebra of Programs.
Communications of ACM 21(8), August, 1978.

[Bayer 72] Bayer, B. and McCreight, A.
Organisation and Maintenance of Large Ordered Indexes.
Acta Informatica 1:173-189, 1972.

[Biller 79] Biller, H.
On the notion of irreducible relations.
In Brachhi, G. and Nijssen, G.M. (editor), *Data Base Architecture*, pages 277-296. North-Holland, 1979.

[Blasgen 77] Blasgen, M. W. *et al.*
System R: an architectural overview.
IBM Systems Journal 20:41-62, 1977.

[Brachhi 76] Brachhi, G., Paolini, P., and Pelagatti, G.
Binary Logical Associations in Data Modelling.
In Nijssen, G.M. (editor), *Modelling in Database Management Systems.* North-Holland, 1976.

[Breutmann 79] Breutmann, B., Falkenberg, E., and Mauer, R.
CSL: A Language for Defining Conceptual Schemas.
In Brachhi, G. and Nijssen, G.M. (editor), *Database Architecture.* North-Holland, 1979.

[Brodie 80] Brodie, M.L.
Application of Data Types to Database Semantic Integrity.
Information Systems 5:287-296, 1980.

[Brown 75] Brown, A.P.G.
Modelling a Real World System and Designing a Schema to Represent it.
In Douque, B,C.M., and Nijssen, G.M. (editor), *Data Base Description.* North-Holland, 1975.

[Bubenko 76] Bubenko, J.A., et al.
From Information Requirements to DBTG data structures.
In *Proceedings of ACM SIGMOD SIGPLAN Conf. on Data: Absraction, Definition and Structures.* ACM, 1976.

[Bubenko 77] Bubenko, J.A.
Validity and Verification Aspects of Information Modelling.
In *Proceedings of International Conference on Very Large Data Bases.* ACM, 1977.

[Bubenko 80a] Bubenko, J.A.
Data Models and their Semantics.
In Atkinson, M. (editor), *Infotech State of the Art Report on Database,* . Infotech, 1980.

[Bubenko 80b] Bubenko, J.A.
Information Modelling in the Context of System Development.
In *Proceedings of IFIP Congress 80*, . North-Holland, 1980.

[Buneman 79] Buneman, P. and Frankel, R.E.
FQL - A Functional Query Language.
In *Proceedings of International Conference on Management of Data*. ACM-SIGMOD, 1979.

[Buneman 82] Buneman, P., Frankel, R.E. and Nikhil, R.
An Implementation Technique for Database Query Languages.
ACM Transactions on Database Management 7(2), June, 1982.

[Burstall 80] Burstall, R.M., MacQueen, D.B., and Sannella, D.T.
HOPE: An Experimental Applicative Language.
Technical Report CSR-62-80, University Of Edinburgh, 1980.

[CCA 83] *ADAPLEX: Rationale and Reference Manual*
1983.
Computer Corporation Of America.

[Chamberlin 76] Chamberlin, D. D.
Relational data-base management systems.
ACM Computing Surveys 8:43-66, March, 1976.

[Chan 82] Chan, A., et. al.
Storage and Access Structures to Support a Semantic Data Model.
In *Proceedings of Eighth International Conference on Very Large Data Bases*, . ACM, 1982.

[Chang 78] Chang, A.
DEDUCE 2: Further Investigations of Deduction in Relational Databases.
In Gallaire, H. and Minker, J. (editor), *Logic and Databases*, pages 201-236. Plenum Press, 1978.

[Chen 76] Chen, P.P.S.
The Entity-Relationship Model: Towards a Unified View of Data.
ACM Transactions on Database Systems 11(1), March, 1976.

[CODASYL 71] Codasyl Committee on Data System Languages.
CODASYL Data Base Task Group Report.
Technical Report, ACM, 1971.

[Codd 70] Codd, E.F.
A Relational Model for Large Shared Databanks.
Communications ACM 13(6):377-387, 1970.

[Codd 72] Codd, E.F.
Further Normalisation of the Data Base Relational Model.
In *Data Base Systems, Courant Computer Science Symposia Series, Vol. 6.*, . Prentice-Hall, 1972.

[Codd 74] Codd, E.F.
Recent Investigations in a Relational Database System.
In *Information Processing 74*, pages 1017-1021. North-Holland, 1974.

[Codd 79] Codd, E.F.
Extending the Relational Model of Data to Capture More Meaning.
ACM Transactions on Database Systems 4(4), December, 1979.

[Cullinane 75] *Integrated Database Management System (IDMS) Data Definition Languages, Utilities and GCI Reference Guide and Data Manipulation Language Programmer's Guide*
1975.

[Date 81] Date, C.J.
Referential Integrity.
In *Proceedings of 7th International Conference on Very Large Data Bases*. VLDB, 1981.

[Date 83] Date, C.J.
The Systems Programming. Volume 2: *An Introduction to Database Systems.* Addison-Wesley, 1983.

[Dayal 78] Dayal, U. and Bernstein, P.A.
On the Updatability of Relational Views.
In *Proceedings of 4th International Conference on Very Large Data Bases.* ACM, 1978.

[Earnest 75] Earnest, C.P.
Selection and higher level structures in networks.
In Douque, B.C. and Nijssen, G.M. (editor), *Data Base Description*, . North-Holland, 1975.

[Eswaran 75] Eswaran, K.P., and Chamberlin, D.D.
Functional Specifications of a Subsystem for Database Integrity.
In *Proceedings of International Conference on Very Large Data Bases.* ACM, 1975.

[Falkenberg 76] Falkenberg, E.D.
Concepts for Modelling Information.
In Nijssen, G.M. (editor), *Modelling in Data base Management Systems.* North-Holland, 1976.

[Folinus 74] Folinus, J.J., Madnick, S.E., and Shutzmann, H.B.
Virtual Information in Database Systems.
FDT, SIGFIDET 6, 1974.

[Friedman 76] Friedman, D.P. and Wise, D.S.
CONS should not evaluate its arguments.
In *Automata, Languages, and Programming*, . Edinburgh University Press, 1976.

[Fry 76] Fry, J. P. and Sibley, E. H.
Evolution of data-base management systems.
ACM Computing Surveys 8:7-42, March, 1976.

[Fry 78] Fry, J.P., and Teorey, T.J.
Design and Performance Tools for Improving Database Usability and Responsiveness.
In Shneiderman, B. (editor), *Databases: Improving Usability and Responsiveness*, . Academic Press, 1978.

[Gerritsen 75] Gerritsen, R.
A Preliminary System for the Design of DBTG data structures.
Communications of ACM 18(10), October, 1975.

[Gordon 79] Gordon, M. J., Milner, A. J. R. G., and Wadsworth, C. P.
Lecture Notes in Computer Science. Volume 78: *Edinburgh LCF.* Springer-Verlag, 1979.

[Gray 81] Gray, P. M. D.
Use of automatic programming and simulation to facilitate operations on CODASYL databases.
In M. P. Atkinson (editor), *Database*, pages 345-369. Pergamon Infotech, 1981.

[Gray 83] Gray, P.M.D.
The Functional Data Model Related to the CODASYL Model.
In Stocker, P. (editor), *Databases: Role and Structure*, . Cambridge University Press, 1983.
To be published.

[Groenfeld 74] Groenfeld, N.R.
Quantifiers in a Relational Data System.
In *Proceedings of AFIPS National Computer Conference*, . AFIPS, 1974.

[Hall 76a] Hall, P.J., Owlett, J., and Todd, S.
Relations and Entities.
In Nijssen, G.M. (editor), *Modelling in Database Management Systems*. North-Holland, 1976.

[Hall 76b] Hall, P.A.V.
Optimisation of a Single Relational Expression in a Relational Database.
IBM Journal of Research and Development 20(3):244-257, 1976.

[Hammer 78] Hammer, M. and McLeod, D.
The Semantic Data Model: a Modelling Mechanism for Database Applications.
In *Proceedings of International Conference on the Management of Data*. ACM-SIGMOD, 1978.

[Hammer 81] Hammer, M. and McLeod, D.
Database Description with SDM: A Semantic Database Model.
ACM Transactions on Database Systems 6(3), Sept, 1981.

[Hayes 77] Hayes, P.J.
On Semantic Nets, Frames, and Associations.
In *Proceedings of 5th International Joint Conference on Artificial Intelligence*, pages 99-107. , 1977.

[Henderson 76] Henderson, P. and Morris, J.H.
A Lazy Evaluator.
In *3rd ACM Symposium on Principles of Programming Languages*, pages 95-103. ACM, 1976.

[Hepp 83] Hepp, P.
A DBS Architecture Supporting Coexisting Query Languages and Data Models.
PhD thesis, University Of Edinburgh, 1983.
to be submitted.

[Hiembigner 81] Hiembigner, D. and Mcleod, D.
Federated Information Bases (A Preliminary Report).
Technical Report TR-105, University of Southern California, October, 1981.

[Honeywell 72] *Integrated Data Store Reference Manual*
1972.
BR69.

[Housel 79] Housel, B.C., Waddle, V., and Yao, S.B.
The Functional Dependency Model for Logical Database Design.
In *Proceedings of 5th International Conference on Very Large Databases*. VLDB, 1979.

[IBM 75] *Information Management System Virtual Store (IMS/VS) Reference Manual*
1975.
GH120-1260-3.

[Ichbiah 79] Ichbiah et al.
Rationale of the Design of the Programming Language Ada.
ACM Sigplan Notices 14(6), 1979.

[ISO 82] ISO TC97/SC5/WG3.
Concepts and Terminology for the Conceptual Schema and the Information Base.
Technical Report ISO/TC97/SC5 - N695, ISO, 1982.

[Kahn 76] Kahn, B.K.
A method for Describing the Information Required by the Database Design Process.
In *Proceedings of ACM SIGMOD International Conference on Management of Data*, . ACM SIGMOD, 1976.

[Katz 83] Katz, R.H. and Wong, E.
Resolving Conflicts in Global Storage Design Through Replication.
ACM Transactions on Database Systems 8(1):110-135, March, 1983.

[Kent 78] Kent, W.
Data and Reality.
North-Holland, 1978.

[Kent 79] Kent, W.
Limitations of Record-based Information Models.
ACM Transactions on Database Systems 4(1), 1979.

[Kerschberg 75] Kershberg, L. and Pacheco, J.E.S.
A Functional Database Model.
Technical Report, Pontificia Universidade Catolica, Rio de Janeiro, 1975.

[Kerschberg 76] Kerschberg, L., Klug, A., and Tsichritzis, D.
A Taxonomy of Data Models.
Technical Report CSRG-70, University of Toronto, 1976.

[Koenig 81] Koenig, S. and Paige, R.
A Transformational Framework for the Automatic Control of Derived Data.
In *Proceedings of the 7th International Conference on Very Large Data Bases.* ACM, 1981.

[Kowalski 74] Kowalski, R.
Predicate Language as Programming Language.
In *Information Processing 74*, pages 569-574. North-Holland, 1974.

[Kulkarni 83] Kulkarni, K.G.
Extended Functional Data Model - User Manual.
Technical Report, University of Edinburgh, June, 1983.

[Kung 81] Kung, H. T. and Robinson, J. T.
On optimistic methods for concurrency control.
ACM Transactions on Database Systems 6:213-22, June, 1981.

[Lampson 77] Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G., and Popek, G.L.
Report on the programming language EUCLID.
SIGPLAN Notices 12(2), 1977.

[Landin 66] Landin, P.J.
The Next 700 Programming Languages.
Communication of ACM 9(3):157-164, 1966.

[Langefors 66] Langefors, B.
Theoretical Analysis of Information Systems.
1966.
Studentlitteratur, Lund.

[Langefors 77] Langefors, B.
Information Systems Theory.
Information Systems 2(2), 1977.

[MacLane 67] Maclane, S. and Birkhoff, G.
Algebra.
The Macmillan Co., New York, 1967.

[McCarthy 62] McCarthy, J., et al.
LISP 1.5 Programmer's Manual
1962.
MIT Press.

[McLeod 76] McLeod, D.J.
High Level Domain Definitions in a Relational Data Base System.
In *Proceedings of ACM SIGMOD-SIGPLAN Conference On Data.* ACM, 1976.

[Mcleod 80] Mcleod, D. and Heimbigner, D.
A Federated Architecture for Database Systems.
In *Proceedings of the National Computer Conference.* AFIPS, 1980.

[Minker 78] Minker, J.
An Experimental Relational Database System Based on Logic.
In Gallaire, H. and Minker, J. (editor), *Logic and Databases*, pages 107-147. Plenum Press, 1978.

[Morrison 79] Morrison, R.
S-algol Language Reference Manual.
Technical Report CS/79/1, University of St. Andrews, 1979.

[Morrison 82a] Morrison, R.
The String as a Simple Data Type.
ACM SIGPLAN Notices 17(3), 1982.

[Morrison 82b] Morrison, R.
Low Cost Computer Graphics for Micro Computers.
Software, Practice and Experience 12:767-776, 1982.

[Munz 78] Munz, R.
The WELL System: A Multi-user Database System Based on Binary Relationships and Graph-Pattern-Matching.
Information Systems 3:99-115, 1978.

[Mylopoulos 80] Mylopoulos, J., Bernstein, P.A. and Wong, H.K.T.
A Language Facility for Designing Database Intensive Applications.
ACM Transactions on Database Systems 5(2), June, 1980.

[Nau 83] Nau, D.S.
Expert Computer Systems.
Computer 16(2), 1983.

[Nijssen 80] Nijssen, G.M.
Database Semantics.
In Atkinson, M. (editor), *Infotech State of the Art Report on Database*, . Infotech, 1980.

[Pirotte 80] Pirotte, A. and Lacroix, M.
User Interfaces for Database Application Programming.
In *Infotech State of the Art Conference on Database*, . Infotech Limited, 1980.

[Quillian 68] Quillian, M.R.
Semantic Memory.
In Minsky, M. (editor), *Semantic Information Processing*, . MIT Press, 1968.

[Ross 83] Ross, G.D.M.
Virtual Files: A Framework for Experimental Design.
PhD thesis, University of Edinburgh, 1983.
to be submitted.

[Roussopoulos 75] Roussopoulos, N. and Mylopoulos, J.
Using Semantic Networks for Database Management.
In *Proceedings of International Conference on Very Large Data Bases*. ACM, 1975.

[Rowe 79] Rowe, L., and Shoens, K.
Data Abstraction, Views and Updates in RIGEL.
In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 71-81. ACM-SIGMOD, 1979.

[Schiel 83] Schiel, U.
A Semantic Data Model and its Mapping to an Internal Relational Data Model.
In Stocker, P. (editor), *Databases: Role and Structure*. Cambridge University Press, 1983.
To be published.

[Schmid 75] Schmid, H.A., and Swenson, J.R.
On the Semantics of the Relational Model.
In *Proceedings of International Conference on the Management of Data*. ACM-SIGMOD, 1975.

[Schmidt 77] Schmidt, J.W.
Some High Level Language Constructs for Data of Type Relation.
ACM Transactions on Database Systems 2(3):247-281, September, 1977.

[Schmidt 78] Schmidt, J.W.
Type Concepts for Database Definition.
In Shneiderman, B. (editor), *Databases: Improving Usability and Responsiveness*, . Academic Press, 1978.

[Senko 73] Senko, M.E., Altman, E., Astrahan, M., and Fehder, P.
Data Structures and Accessing in Data Base Systems.
IBM Systems Journal 12(1), 1973.

[Senko 75] Senko, M.E.
The DDL in the Context of a Multilevel Structured Description: DIAM II with FORAL.
In Douque, B.C.M. and Nijssen, G.M. (editor), *Data Base Description*. North-Holland, 1975.

[Severance 76] Severance, D. and Lohman, G.
Differential files: Their Application to the Maintenance of Large Databases.
ACM Transactions on Database Systems 1:256-267, September, 1976.

[Sharman 77] Sharman, G.C.H.
Update-by-Dialogue: An Interactive Approach to Database Modification.
In *Proceedings of ACM SIGMOD International Conference on Management of Data*. ACM-SIGMOD, 1977.

[Shipman 81] Shipman, D.W.
The Functional Data Model and the Data Language DAPLEX.
ACM Transactions on Database Systems 6(1):140-173, March, 1981.

[Shopiro 79] Shopiro, J.E.
THESEUS - A Programming Language for Relational Databases.
ACM Transactions on Database Systems 4(4), December, 1979.

[Sibley 77] Sibley, E.H., Kerschberg, L.
Data Architecture and Data Model Considerations.
In *Proceedings of AFIPS National Computer Conference*, pages 85-96. AFIPS, 1977.

[Smith 75] Smith, J.M. and Chang, P.Y.
Optimising the Performance of a Relational Algebra Database Interface.
Communications of ACM 18(10):568-579, 1975.

[Smith 77] Smith, J.M. and Smith, D.C.P.
Database Abstractions - Aggregation and Generalisation.
ACM Transactions on Database Systems 2(2), June, 1977.

[Smith 80a] Smith, J.M. and Smith, D.C.P.
Conceptual Database Design.
In Atkinson, M. (editor), *Infotech State of the Art Report on Database*, . Infotech, 1980.

[Smith 80b] Smith, J.M., Bernstein, P.A., et. al.
Basic Architecture of Multibase.
Technical Report, Computer Corporationof America, November, 1980.

[Smith 81] Smith, J.M., Fox, S. and Landers, T.
Reference Manual for ADAPLEX.
Technical Report, Computer Corporation of America, January, 1981.

[Stonebraker 75]
Stonebraker, M.
Implementation of Integrity Constraints and Views by Query Modification.
In *Proceedings of ACM SIGMOD International Conference on the Management of Data*. ACM, 1975.

[Stonebraker 76]
Stonebraker, M., Wong, E., Kreps, P., and Held, G.
The Design and Implementation of INGRES.
ACM TODS 1(3):189-222, September, 1976.

[Stonebraker 80]
Stonebraker, M.
Retrospective on a database system.
ACM Transactions on Database Systems 5:225-240, June, 1980.

[Strachey 67] Strachey, C.
Fundamental Concepts in Programming Languages.
Oxford University, 1967.

[Sundgren 74] Sundgren, B.
Conceptual Foundation of the Infological Approach to Data Bases.
In Klimbie, J.W. and Koffeman, K.I. (editor), *Data Base Management*, . North-Holland, 1974.

[Taylor 76] Taylor, R. W. and Frank, R. L.
CODASYL data-base management systems.
ACM Computing Surveys 8:67-103, March, 1976.

[Teichroew 77] Teichroew, D. and Hershey, E.A.
PSL/PSA: A computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems.
IEEE Transactions on Software Engineering , 1977.

[Todd 76] Todd, S.
The Peterlee Relational Test Vehicle - A System Overview.
IBM Systems Journal 15(4):285-307, 1976.

[Tsichritzis 76] Tsichritzis, D. C. and Lochovsky, F. H.
Hierarchical data-base management: a survey.
ACM Computing Surveys 8:105-123, March, 1976.

[Tsichritzis 77] Tsichritzis, D. C. and Lochovsky, F. H.
Data base management systems.
Academic Press, 1977.

[Tsichritzis 82] Tsichritzis, D.C., and Lochovsky, F.H.
Data Models.
Prentice-Hall, Inc., 1982.

[Turner 79] Turner, D.A.
A new implementation technique for applicative languages.
Software - Practice and Experience 9:31-49, 1979.

[Turner 82] Turner, D.A.
Recursion Equations as a Programming Language.
In *Functional Programming and its Applications*, . Cambridge University Press, 1982.

[Ullman 82] Ullman, J.D.
Principles of Database Systems.
Pitman, 1982.
Second Edition.

[van Wijngaarden 69]
van Wijngaarden, A., et al.
Report on the Algorithmic Language Algol 68.
Numerische Mathematik 14:79-218, 1969.

[Wasserman 81] Wasserman, A.L., Shertz, D.D., Kersten, M.L., Reit, R.P., and van de Dippe, M.D.
Revised Report on the Programming Language PLAIN.
ACM SIGPLAN Notices , 1981.

[Wiederhold 79] Wiederhold, G. and El-Masri, R.
Structural Model for Database Design.
In *Proceedings of International Conference on the Entity-Relationship Approach to Systems Analysis and Design*. North-Holland, 1979.

[Wirth 66] Wirth, N. and Hoare, C.A.R.
A Contribution to the Development of Algol.
Communications of ACM 9(6):413-431, 1966.

[Wirth 71] Wirth, N.
The Programming Language PASCAL.
ACTA Informatica 1, 1971.

[Wirth 77] Wirth, N.
What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definition.
Communications of ACM 20(11):822-823, November, 1977.

[Wong 76] Wong, E. and Youssefi, K.
Decomposition- a Strategy for Query Processing.
ACM Transactions on Database Systems 1(3):223-241, 1976.

[Wong 79] Wong, E. and Katz, R.H.
Logical Design and Schema Conversion for Relational and DBTG Databases.
In *Proceedings of International Conference on Entity-Relationship Approach to Systems Analysis and Design*, . North-Holland, 1979.

[Yao 78a] Yao, S.B., et al.
Data-Base Systems.
Computer 11(9):46-60, September, 1978.

[Yao 78b] Yao, S.B., Navathe, S.B. and Weldon, J.L.
An Integrated Approach to Logical Database Design.
In *NYU Symposium on Logical Database Design*, . NYU, 1978.

[Yao 82] Yao, S.B., Waddle, V., and Housel, B.
Database Design and Integration using a Functional Data Model and a Transactions Specification Language.
IEEE Transactions on Software Engineering , 1982.