Binary Relation Database BIRD: Issues of Representation and Implementation

Piercy, Richard Michael

Binary Relation Database BIRD : Issues of Representation and Implementation

Richard Michael Piercy

A thesis submitted for the Degree of

Master of Science

of the **University of Durham**

School of Engineering and Applied Science

(Computer Science)

University of Durham

December 11, 1989

Abstract

This thesis presents a study of two issues, integrity and homogeneity of information representation, within the area of databases. Treatment of these issues were studied within the standard and semantic database models, leading to the proposal of a new model, the Binary Relation Database, BIRD. The BIRD model uses the binary relationship as the basis for the representation of all database data and meta–data.

The inadequacy of integrity definition facilities within current database technology are elaborated in this thesis and were taken into account in the BIRD system. The effects of inhomogeneity of database data and meta–data in current databases are described and the benefits of the homogeneity of information representation in BIRD demonstrated.

BIRD was implemented as a prototype database system, using Modula–2, – the implementation and subsequent evaluation of the system are included in this thesis. A simple user menu driven user interface to BIRD was constructed, – the user may manipulate information at any conceptual level in the system in a homogeneous manner. The user is free to manipulate information from any conceptual level at any time, – BIRD ensures that the database is returned to a consistent state before the next operation may take place.

The new model proposed in this thesis fulfilled its objectives, – suggestions for further and implementation oriented work are presented at the end of the thesis.

Acknowledgements

This work was supported by the Science and Engineering Research Council.

I wish to thank the staff of Computer Science for their support and friendship during the course of my undergraduate and postgraduate studies. Thanks are also due to F. W. Calliss for help along the way, particularly with the subtleties of LaTeX word processing and to my long suffering house–mate, Dr. S. T. Wait, for his culinary expertise and help with proof reading.

Contents

List of Figures

Dedicated to my parents, Brian and Lois.

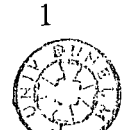
Chapter 1

Introduction

This thesis presents a study of issues relating to integrity and homogeneity of information representation in database systems. This study led to the development of a new database model, the Binary Relation Database, BIRD, which represents information in terms of binary relationships between objects. Readers who desire an overview of the issues described above and a description of the BIRD model are referred to Piercy and Slade [35].

1.1 Introduction to Databases

Introductory texts providing a broad background to the many issues of database technology may be found in [11, 26, 45]. Many definitions of the concept, 'database', may be found in the literature, Sundgren [45, pp10] describes a database as a, "... well organised collection of data. One should be able to process, update, and make additions to the contents of a database in a simple and flexible way. It should also be easy to make different kinds of unplanned as well as planned retrievals of data from the database."



A database system provides convenient, efficient and centralised control over a body of data. The alternatives to using a database are to record the information manually or to keep it in system files, each with one or more application programs to access the information contained therein. The use of a database system has many advantages detailed below :–

- Consistency – the storage of a single copy of information which might otherwise be duplicated in multiple locations eliminates redundancy and the potential for inconsistency. In practise, it is sometimes necessary to duplicate information in the database, to avoid problems associated with particular representations. However if the duplicates are known and contained within a single system, then the chances of inconsistency is reduced.

- Flexibility – new applications requiring different information from the database may be easily accommodated since there is one repository of information with a query language interface. A file based system would find it harder to accommodate a new application since new application programs may have to be written to extract the required data, plus problems arise if the pertinent information is distributed in many different files.

- Multi–user – the database management system, DBMS, may provide facilities for many applications to access the database concurrently without danger of anomalous effects.

- Security – a DBMS will provide facilities to define selective access to the information for different users. Facilities are also provided to protect the contents of the database against system crashes.

- Integrity – the DBMS may provide facilities for the definition of integrity constraints. These constraints catch data values which are outside pre–defined ranges and thus aid the maintenance of the integrity of the database data.

- Distribution of Data – many database systems will allow information stored at physically separate locations to be considered part of a single database. In

this way there is no need for duplication of information at separate sites and thus problems of inconsistency are avoided.

1.2 Expository Example

Within this thesis, an example of a hypothetical factory parts ordering database is used in many chapters for the purposes of exposition. The hypothetical situation to be represented is a manufacturing company which receives orders from other companies for its products.

Each order received from a client is dated and includes the client's name and address. Associated with each order is a list of order items, which are comprised of part numbers with an associated quantity figure. On receipt of the order, the company assigns it a unique order number which may be used to identify individual orders in the database.

This example is then extended, in the chapter detailing the operation and evaluation of BIRD, to include a record of the invoices which the company raise in response to the goods dispatched, as well as the orders sent to their suppliers and invoices received from them. The invoices include similar information as the orders, – a unique invoice number; company name; company address; date; item list; unit cost per item and total cost.

1.3 Structure of the Thesis

The thesis describes record oriented database models, semantic database models before concentrating upon the issues of integrity and homogeneity of information representation which led to the development of the BIRD model.

The study of record based database models is centred upon an analysis of the three dominant data models which have evolved since interest in databases started in the 1960's, including a description of the currently popular relational data model. Semantic data models are introduced in the next chapter, – these models have been developed since the early 1970's with a view to capturing more of the semantics of the application environment than the record oriented models. These models have so far only been used in database design, as a medium to facilitate communication between the relevant parties.

The subsequent chapters analyse issues of data integrity and homogeneity of information representation with respect to traditional and semantic models. This analysis provides the motivation for the proposal of a new database model, BIRD, which addresses the problems identified earlier in the thesis.

The description of BIRD commences with a statement of the principles underlying the development of BIRD. The design and implementation of BIRD are then documented including a full evaluation of the resulting system. The last chapter presents the conclusions of the thesis and includes ideas for further work. The appendices found at the end of the thesis consist of a section describing the function of the constituent procedures of the BIRD system, followed by the bibliography.

Chapter 2

Record Oriented Database Models

Traditional databases represent information in records, which are arranged and connected in different ways according to the model. This chapter reviews the record oriented models, comparing and contrasting their features, with the last section presenting a summary of their features. The ability of each of the models to maintain data integrity is described in detail in the later chapter on database integrity. Issues related to the homogeneity of information representation in record oriented models may be found in the later chapter entitled, 'Homogeneity of Information Representation in Databases'. Introductory texts which review the different database models in more detail may be found in [15, 26, 33, 43].

2.1 Hierarchical Data Model

The Hierarchical Data Model, HDM, such as IBM's Information Management System, IMS [28], is the oldest traditional data model, and is based on the premise

that application domain information is hierarchical in nature. In the HDM, records are organised in rooted trees, where nodes correspond to records and arcs represent parent–child relationships. A rooted tree is a graph where there are no cycles and only one to one or one to many relationships exist between parent and child nodes respectively.

A record within the HDM is a collection of fields, where each field may contain at most one value. No record may exist in the HDM without a parent record except for the special root record, and thus the deletion of a record causes the deletion of all records below it in the hierarchy. The utility of this restriction might be demonstrated in a company database where the deletion of an employee record would cause the deletion of all personnel records associated with that employee record. In certain circumstances one may wish to retain records without parents, – for example in an ordering system where each part number in an order may be associated with a part description. If the current orders did not include a particular part number then one would not wish the part's description to be deleted from the database since it might be referenced by future orders.

In the HDM there is no way of directly modelling a many to many relationship, such as the *teaching* relationship between teachers and pupils, since a child record may not have more than one parent record. Two approaches are used to overcome this problem, – records may be duplicated, see figure 2.1, or *connection* records may be employed, see figure 2.2. Connection records, sometimes known as virtual or buffer records contain no application domain information, – they are just used as a connection to other records.

Both of the solutions described reduce the many to many relationship between teachers and pupils into multiple one to many relationships. The use of connection records to represent many to many relationships is preferable to the duplication of information shown in figure 2.1, since connection records are not associated with data inconsistency problems although they do introduce an extra level of indirection into the database. Both solutions involve an overhead in storage space and obscure

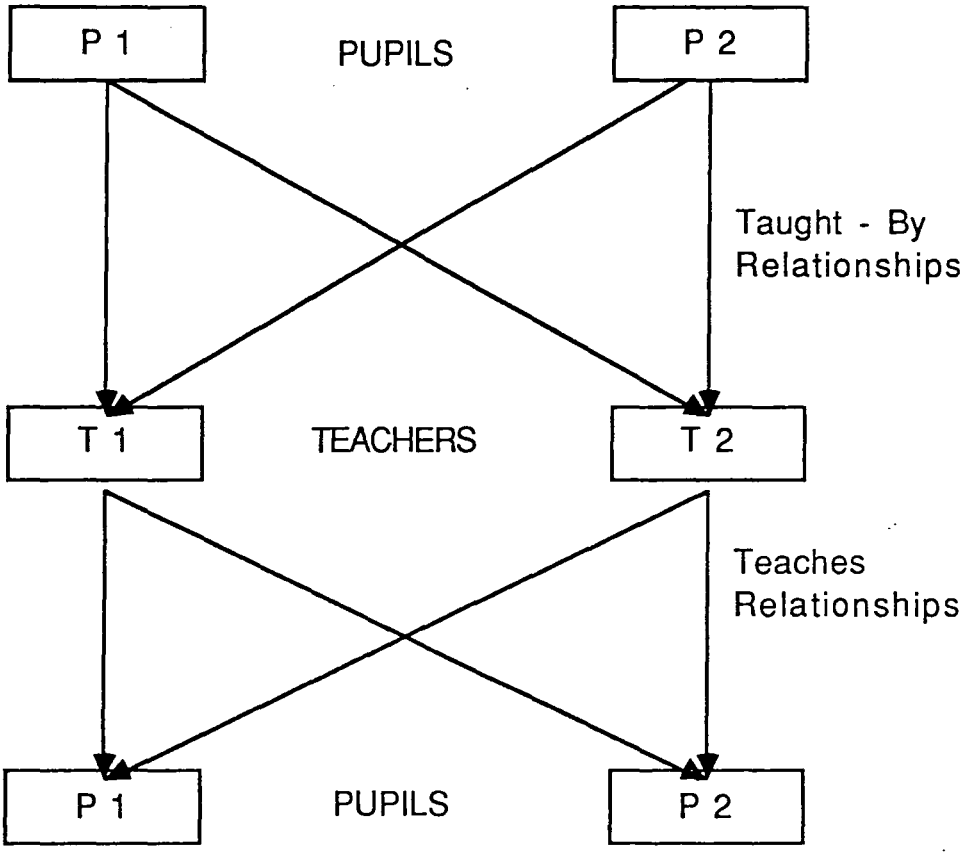


Figure 2.1: Modelling Many to Many Relationships Via Record Duplication

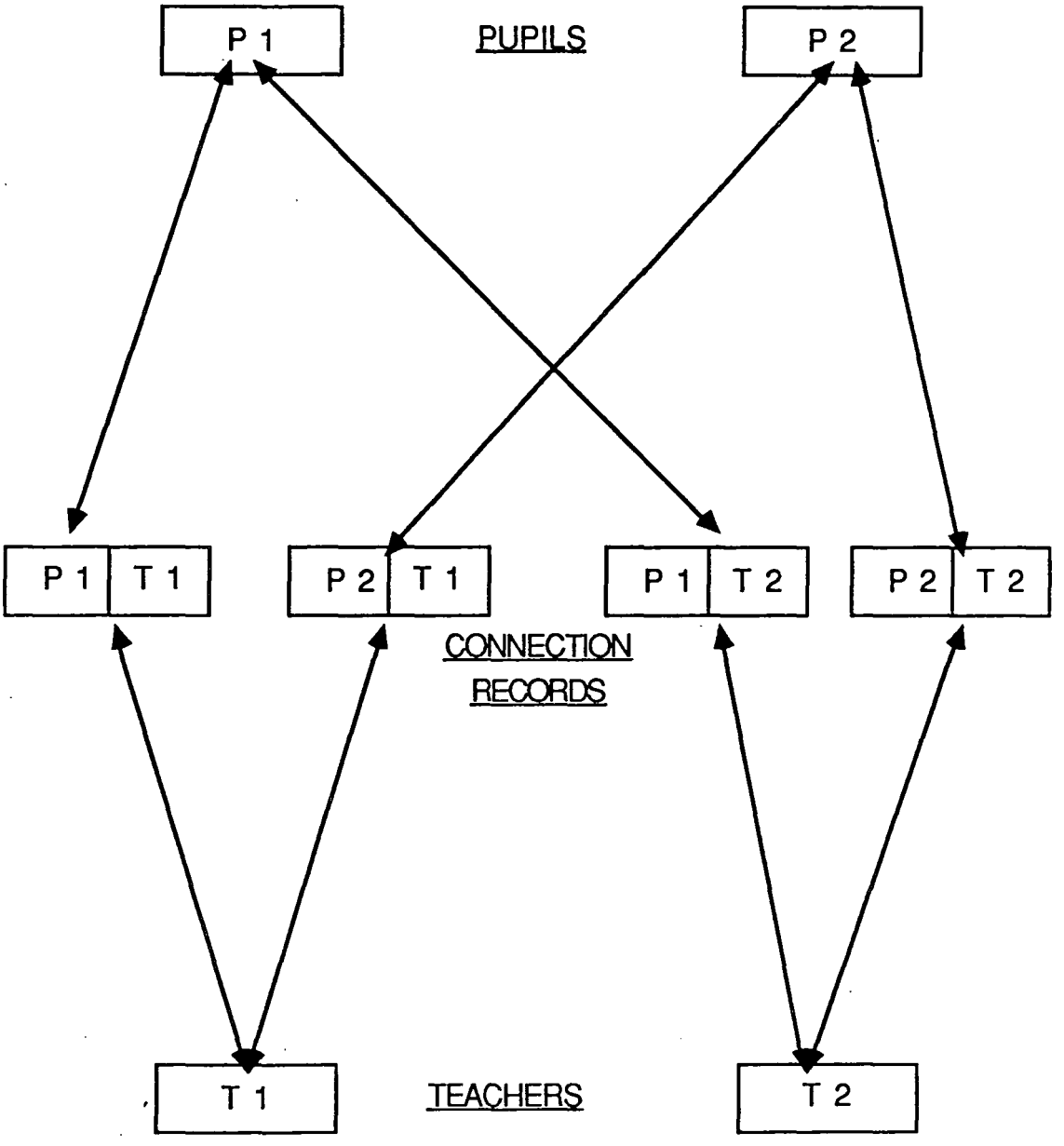


Figure 2.2: Modelling Many to Many Relationships Using Connection Records

the form of the relationship by reducing it to multiple one to many relationships.

Connection records can also be used to represent local cycles between records, such as the *manages* relationship between employee records. In this case a connection record is associated with a record at a certain hierarchical level and it in turn points to other records at that hierarchical level.

By virtue of its organisation, a HDM implementation will favour certain database operations. Consider a manufacturing database where there is a record for each company and subrecords for each product a company produces. Retrieving the names of products a company produces would be trivial, however retrieving the names of companies which manufacture a certain product would be far more time consuming since the structure of the records in the database is organised with respect to retrieval via company names. Queries which involve tracing a path from the root down, can be executed efficiently in the HDM but other queries may take prohibitively long times. Certain implementations provide facilities for speeding time consuming queries such as index files, however each index file represents a storage overhead and they can only be provided for queries which are anticipated by the database administrator. Owing to the correspondence between the efficiency of database manipulation operations and the database schema the user must consider internal implementation details when accessing a HDM database, – a situation which should be avoided wherever possible.

2.2 Network Data Model

The Network Data Model, NDM, is composed of records connected together to form a graph structure, and thus it is a generalisation of the HDM. CODASYL [13, 31] is the most dominant NDM and will be described in this chapter, other implementations include IDMS and ADABAS [47].

In 1959 an organisation called the Conference On DAta Systems Languages, CODASYL, was set up with a brief to develop techniques to aid in data systems analysis, design and implementation. The CODASYL group comprised individuals from many institutions and from this a Data Base Task Group, DBTG, was formed in 1965. The DBTG first published a report in 1969 detailing a Data Description Language, DDL, to define a database schema and a Data Manipulation Language, DML, to manipulate the data contained within the database. This report was further revised and published in 1971 [13] when it was accepted by the Programming Languages Committee of the CODASYL organisation. Since then the CODASYL model has continued to evolve [14] and it will be used to illustrate the salient features of the NDM.

A CODASYL set type consists of a unique set name, an owner record type and one or more member record types. A set occurrence consists of one owner record from the owner record type and zero or more records from the member record types. The rules governing the relationships between CODASYL records may be summarised as follows :-

- A record type may be the owner of one set type and a member in another.
- A record may be a member in more than one set type.
- There is no limit on the number of set types which may be defined between any two record types.
- A database may contain any number of record types and any number of set types.
- Cyclic structuring is permitted, however a record type may not be both the owner and member of a given set type, prohibiting local cycles within a set type.
- A record may not appear more than once in the occurrences of any particular set type.

- The field of a record may contain a set of values and thus repeating groups may be represented.

Unlike in the HDM, records in CODASYL may exist without an owner, – this permits the retention of records in the database which do not have an owner at that instant, but which may be needed, such as in the hypothetical parts database previously described. Set insertion and retention rules which govern the insertion, deletion and disconnection of records may be specified when the record types are declared. The particular rules and their utility for maintaining consistency is discussed in the later chapter on database integrity.

Although the NDM extends the representational power of the HDM, it still suffers from many similar problems. Local cycles cannot be represented, – this problem can be approached in two ways. Firstly the set type can be split up into a hierarchy of set types and these can be linked by relationships or secondly the local cycle can be represented indirectly using connection records as described in the previous section for the HDM. Thus to model the "manages" relationship between employees of a company, one could declare different set types for each level in the hierarchy of employees or define the "manages" relationship so that it points to connection records which would in turn point back to employee records.

By virtue of its owner–members set structure, the NDM implicitly provides one to many modelling, however since a record may not appear more than once in the same set type, many to many relationships cannot be directly modelled. The solution to this problem is the same as previously described with respect to the HDM, – either records can be duplicated or connection records employed. Connection records may also be used to link arbitrary numbers of records together, and this can be used to represent more complex relationships such as the three way relationship between teacher, pupils and classroom.

The physical links between owner and members of a set occurrence are unidirectional, one can only traverse from an owner record to its member records. When a connection record is used to implement a many to many relationship, it acts as the member record in both the set types of the entities involved, and thus one cannot directly navigate between the two entities in either direction. One method which enables navigation is to mark the connection records associated with one entity and then inspect all the connection records associated with all the other entities in order to find the ones which have been marked. This solution is obviously too inefficient in practise and thus the connection records are augmented with fields which contain pointers to their parent entities, further increasing the storage overhead.

Schemas within the NDM are complex, – technical experts are often used to manipulate the database and build application programs for the users due to the difficulty novice users would experience in accessing the database using the query language. The reliance on application programs introduces an extra level, resulting in inflexibility and resistance to change, since a programmer has to be employed to effect any change to the data manipulation operations desired by the user accessing the database.

The complexity of schema evolution in the NDM depends on the particular operation performed. Certain schema evolution operations may be effected without restructuring existing application data, such as the introduction of a new set type, however other operations involve considerably more effort. Consider a set type which relates customers in a shop with the articles that they purchase, if it was desired to extend this with information about the branch at which the articles were purchased, then a connection record would have to be introduced to represent the three way relationship and existing pointer values would have to be updated. Similarly if one desired to change a relationship from one to many into many to many, such as the situation where employees may be assigned to more than one department having previously only been assigned to one department, then connection records would have to be introduced and existing application data restructured.

2.3 Relational Data Model

The Relational Data Model, RDM, [27, 40] is the most recent of the traditional database models and was introduced by Codd [9] in the 1970's, [10, pp397] "to free users from the frustrations of having to deal with the clutter of storage representation details". Commercial relational database systems include Relational Technology's INGRES [21, 23], IBM's QBE [50] and Ashton–Tate's dBase products.

The structure of the RDM, see figure 2.3, is a collection of tables, called relations, – thus unlike the other traditional database models, the user no longer has to consider pointer connections between records. A relation corresponds to an entity in the application domain, the rows, called tuples, represent instances of the entity, and each column represents an attribute of the entity.

Date [11, pp239] defines the term relation as :– "A *relation* on domains D1, D2, ...,Dn (not necessarily all distinct) consists of a *heading* and a *body*.

- The *heading* consists of a fixed set of *attributes* A1, A2, ..., An, such that each attribute Ai corresponds to exactly one of the underlying domains Di (i = 1, 2, ..., n).
- The *body* consists of a time–varying set of *tuples*, where each tuple in turn consists of a set of attribute–value pairs (Ai:vi) (i = 1, 2, ..., n), one such pair for each attribute Ai in the heading. For any given attribute–value pair (Ai:vi), vi is a value from the unique domain Di that is associated with the attribute Ai.

The structural principles of the RDM can be summarised from Mayne and Wood [27, pp19] and Date [11, pp241] as :-

- Each relation must contain only one type of record, each with a fixed number of explicitly named attributes.

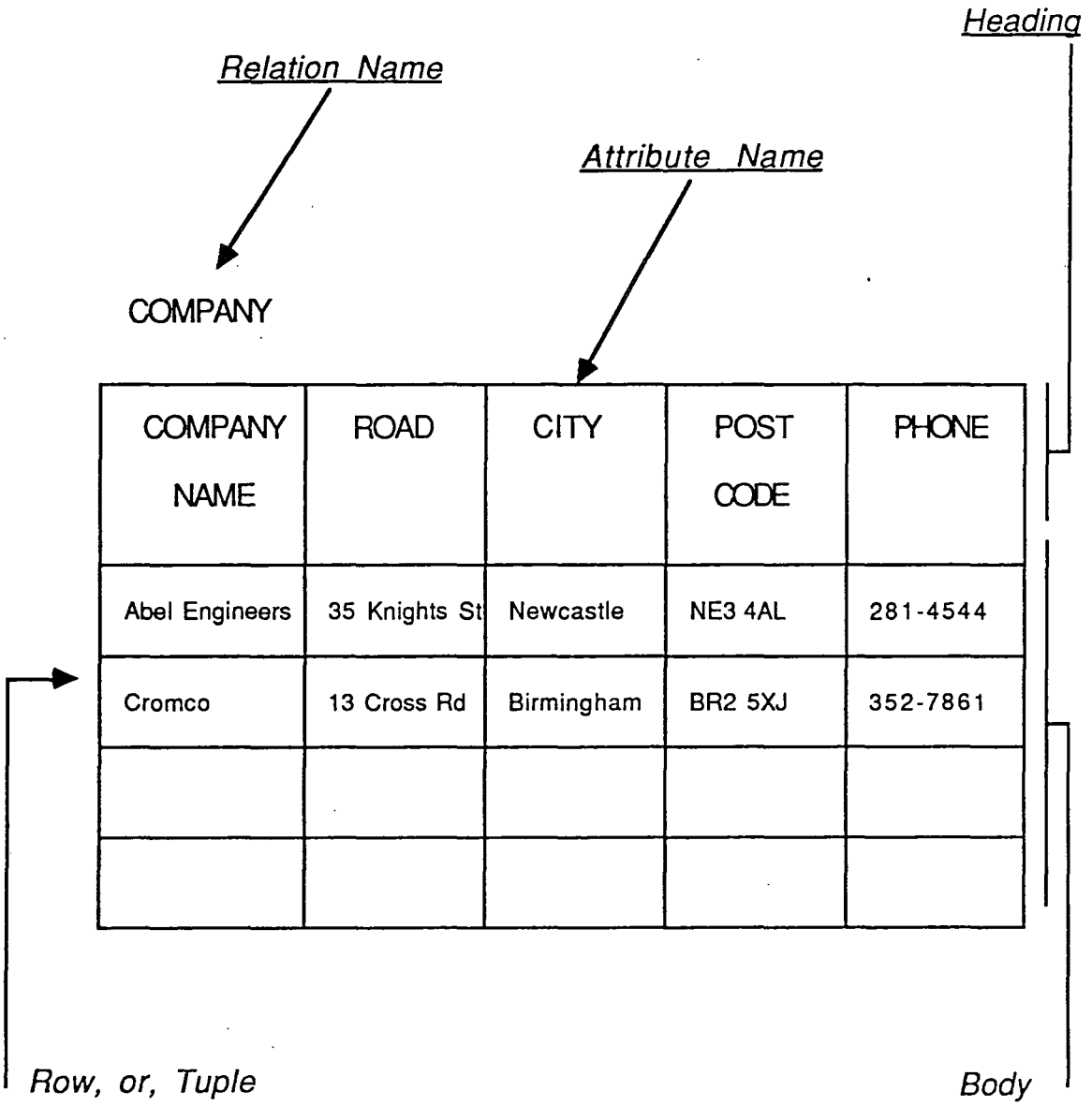


Figure 2.3: Structure of Relations in the Relational Data Model

- Within a relation each attribute must be distinct and repeating groups are not allowed.
- Each tuple in a relation is unique, no duplicates allowed.
- The order of attributes is indeterminate.
- The order of tuples is indeterminate.

The RDM can be manipulated via relational algebra or relational calculus, – both of these methods operate on whole relations. Relational algebra is a high level procedural language consisting of set operators, – union, difference, intersection and special operators such as select, project and join. The select operation is a unary operator which chooses zero or more tuples satisfying a given predicate. The project operation is a unary operator which chooses one or more specified columns. The join operation is a binary operator which is used to combine information from two relations creating a new relation. The join is effected by matching values occurring in a pair of columns one from each relation. A typical relational algebra query which retrieves the names of all employees earning over ten thousand pounds from an employee relation might be :-

SELECT employee–name
FROM employee
WHERE gross–earnings > 10,000

Relational calculus is a non–procedural language, based on first order logic, which merely describes the information required from the database.

The form of a relational calculus query is :-

$\{ t \mid P(t) \}$

- the set of all tuples, t, such that predicate, P, is true for t.
Thus the previous relational algebra query might be rewritten:-

$\{ t \mid t \in employee\ AND\ t[gross\text{-}earnings] > 10,000 \}$

Although hierarchical information may be represented in a relational database using relations for each level of the hierarchy, the query languages provide no explicit constructs for manipulating hierarchical information. For instance there is no construct to directly phrase the query, "Retrieve all the part numbers of parts to be found two levels below the subassembly with part number 675".

The tuples within relations are accessed via a primary key which must be non-null and uniquely identify each entry. There are no direct links or pointers between the relations, instead the matching of attribute values is used as the connection. The ability of tuples to exist in relations completely independently of tuples in other relations can be advantageous in certain situations, like the example previously quoted where we wish to store parts information even if there is no order form directly referencing that part. However, the lack of explicit information relating the attributes of different relations has associated disadvantages. A user may not realise the existence of many relationships between relations or may match values from inappropriate domains, such as matching part numbers in one relation to part quantities in another.

Similarly to the two previous models, the RDM cannot directly represent many to many relationships and this can be overcome by the introduction of an extra relation which contains pairs of relation indexes. Consider the many to many

relationship "teaches" between teachers and pupils, – this would be represented in the relational model by creating an extra relation which held pairs of indexes to the teacher and pupil relations, where each tuple represents one instance of the relationship. The introduction of an extra relation increases the complexity of the model, obscures the form of the relationship for the user, produces a storage overhead and has implications for referential integrity which are described in the later chapter on database consistency.

Normalisation of relational database schemas is a process carried out in the database design stage which ensures freedom from insertion, update and deletion dependencies and facilitates restructuring, – detailed descriptions may be found in many database books such as [26, pp181–215] or [27, pp55–72] .. The process is carried out by analysing the functional dependencies between the application data and forming the structure of the relations with respect to the dependencies.

Consider for example an ordering database which stores order numbers, amounts, customer name and customer address. If one relation was defined to contain all four attributes then it is very likely that a lot of the customer information would be duplicated, since the same company often submits more than one order. Normalisation would result in the formation of two relations, one would hold the customer information indexed on a customer index, the other would hold the order information with a customer index attribute to associate the customer with the orders. In this situation normalisation has ensured that only one copy of the customer name and address would be kept for each customer, reducing the storage overhead and simplifying manipulation of customer information. Although normalisation has many advantages, its disadvantage is the increased number of relations and indices in the system, which obscure the structure of the information from users and increase reliance on the computationally expensive join operation.

2.4 Summary

This section provides a summary of the traditional database models and introduces the concept of the semantic data model.

The HDM is based on the assumption that information associated with the application it is modelling is hierarchical in nature. Random access to information in the resultant tree structure is only efficient if it involves navigation down through the tree, otherwise retrieval of information may take a prohibitively long time. The modelling of non–hierarchical data is possible in the HDM, however it involves either the duplication of information or the introduction of virtual records.

The NDM extends the representational power of the HDM by modelling an application environment in terms of an interconnected graph of entities. The NDM may be used to model most application environments, however this may necessitate the introduction of virtual records. The resultant NDM schemas are often complex, – these must be controlled by the database administrator and accessed via application programs.

The RDM is a conceptually simple model based on tabular information, with no explicit links between the tables. The RDM benefits from being easier to restructure then the other two models and currently is very popular in industry. Although the lack of explicit connections between the information contained in tables has advantages it also enables meaningless joins to be constructed and is associated with referential integrity problems, described fully in the next chapter.

In general, the record based models are able to store efficiently homogeneous units of storage, however Kent [24] describes two types of information homogeneity assumptions which underly them. Horizontal homogeneity is where every record of a given type contains the same fields, for instance where every part in a part inventory might have a part number, quantity on hand and price. Vertical homogeneity is where every field of given record type contains the same kind of information, for

instance a part number is always a six digit integer.

Homogeneity of information is not always borne out in practice and Kent cites book identification numbers as an example of horizontal inhomogeneity, as books may have a Library of Congress number and/or one or more International Standard Book Numbers. The author proposes various solutions which may be adopted to cope with horizontal inhomogeneity in a record based system, – declare all the possible fields and permit null values to be present in some fields or allow a field to have more than one meaning. The first solution necessitates consistency information being built into the application program to ensure that only legal combinations of fields occur, the second introduces problems of vertical inhomogeneity which is discussed below and would only be applicable in this example if a book had *either* a Library of Congress number *or* an International Standard Book Number.

An example of vertical inhomogeneity is proposed by Kent as the situation where a company car may be assigned to an individual or a department. Various solutions were proposed – the car assignment field could be defined to allow identifiers of both groups, more fields could be used such as assignee type, department and employee, two different record types could be defined, one for department car assignments and one for employee car assignments, or employees and departments could all be given a common identifier. The disadvantage of allowing fields to have multiple meanings is that type checking of the field is reduced, – only the users know whether the entity is a department or an employee and thus which record type to search for further information, the different entities might have the same identifiers or the two entities may need different numbers of fields to uniquely identify themselves. Defining more than one field to describe assignments introduces horizontal inhomogeneity which has been discussed above. The use of more than one record type unnecessarily increases the number of record types, potentially all of which may have to be interrogated to extract information about an assignment of a car to a person or department. A data integrity hazard has also been introduced as the same car may be assigned in more than one record and addition of

each new assignment type necessitates the introduction of a new record type. The use of common identifiers necessitates the introduction of one additional identifier for each vertical inhomogeneity, and has the disadvantage that the identifiers' meanings are not often apparent.

In record based systems the more inhomogeneous the information being modelled the less a record based system's structure is able to reflect the inherent structure of the information, as unusual record types and increasing numbers of meaningless identifiers are introduced to overcome the shortfalls of the representation. The reduced correspondence between the internal and external structure complicates a user's understanding of the system and makes maintenance of the system harder.

Kent does not propose a solution to the problems presented in the paper, however he does refer to the superior modelling capabilities of models based on entities and the relationships in which they take part, rather than record structures. These models have been commonly termed 'semantic' models, since they provide richer constructs to model an application environment – the properties of semantic models are described and various models reviewed in the next chapter.

Although the modelling capabilities of the traditional data models have been extended by application independent semantic models, models have also been developed which provide constructs specifically suited to individual applications. Application areas which have been approached include medical information [39], literary texts [7], engineering design information [30], graphics and digitised sound.

Chapter 3

Semantic Oriented Data Models

Semantic oriented data models have been developed to provide constructs which mirror the structure of application domain information in an effort to capture more of the semantics of the application environment than a traditional record oriented model. A brief overview of traditional and semantic models may be found in Potter [36], whereas Hull and King [22] or Peckham and Maryanski [34] review the various models in more depth.

This chapter discusses the term 'semantic model', analyses examples of these models and describes their current applicability. The scope of the models reviewed covers simple extensions to the relational model, to more original models with rich information modelling constructs. Information integrity issues are briefly mentioned where appropriate, – an in depth analysis may be found in the following chapter on database integrity.

Readers should note that in this chapter and throughout the rest of the thesis, entity types are denoted in capital letters and relationships between entities are printed in italics. For example the *order number* relationship might connect the ORDER and ORDER NUMBER entity types.

3.1 Introduction

The term 'semantic model' is commonly used by authors, however it is a nebulous concept since there is no accepted definition of the features of a semantic model and few authors define their usage of the term. Below are a number of relevant descriptions :-

Codd [10, pp397] proposes that the motivation of sematic modelling is to capture more of the meaning of the application environment so that database design may become more systematic and the database system may behave more intelligently.

Hammer and McLeod [20, pp351] state, "This database model (SDM) is designed to capture more of the meaning of an application environment than is possible with contemporary database models". They later state that [20, pp353] " We believe that it is necessary to break with the tradition of record oriented modelling, and to base a database model on structural constructs that are highly user oriented and expressive of the application environment."

Peckham and Maryanski [34, pp153] describe the unifying characteristic of semantic data models as providing more semantic content than the relational data model.

Abiteboul and Hull [1, pp525] describe semantic models as providing "... mechanisms and constructs that mirror the prevalent kinds of relationships naturally arising between data stored in a database."

Abiteboul and Hull [1, pp526], mentioned above, continue to propose four features of semantic data models, – the concepts introduced below will be described in more detail later in the chapter.

- The ability to model relationships between objects directly, – record based implementations introduce unnecessary indirection, since a user must think

in terms of records, pointers and symbolic identifiers.

- Modelling of data as attributes of objects.
- Construction of taxonomies of objects, – this facility is generally provided via the *is–a* relationship.
- The provision of constructor operators, – these enable the description of object types in terms of other object types.

The varied descriptions detailed above, impart a general idea of the concept of semantic models and semantic modelling. If Peckham and Maryanski's description is adopted then almost any simple extension to the relational data model may be described as a semantic data model, such as RM/T, which is described below. Codd developed RM/T as an extension to the relational model, however it is an exception since all the other dominant semantic models are object based in nature. Although RM/T is described in this chapter, it might be considered a hybrid between a traditional and semantic model, particularly since most of the above descriptions of semantic models imply they are object based.

One may summarise the features of a semantic model as providing more expressive constructs than the 'state of the art' databases, enabling a more natural and comprehensive description of the application environment to be constructed. Pure semantic models are object based and currently extend database technology by the inclusion of some or all of the following features, – taxonomies of objects, constructor operators and information about the relationships between objects.

Semantic models are generally used as a database specification facility, – the resulting model is translated into a traditional database implementation. The semantic model specification provides a representation at a suitable conceptual level to form a bridge between the application requirements and the database schema. It also serves as a medium of communication, which is less ambiguous than a prose specification and more readable than a traditional database schema, – this specifi-

cation is at a suitable conceptual level to be referenced by computer experts and non–specialists alike. Novice users of the database may also use the specification as a guide to the structure of the database, although consistency problems may arise between the semantic model database specification and the database itself. Semantic models may be incorporated into the *top down* design methodology, due to their hierarchical information modelling ability, – principle large objects in the application domain may be defined first and then the definition extended to their smaller component objects.

3.2 RM/T

In the late 1970's Codd proposed an extension to the *R*elational *M*odel called RM/T [10], – 'T' stands for Tasmania where the idea was first presented. The motivation for this extension was to increase consistency by eliminating insert and update anomalies as well as increasing the semantic expressiveness of the relational model.

Codd perceived three problems with user–controlled primary keys in the original relational data model. Firstly the user, by definition, must be allowed to change their values and this may result in referential errors. Secondly, two relations may have keys defined on distinct domains which actually refer to the same entities, preventing the use of the join operator. Lastly, information may have to be stored about an entity in situations where it may not have a key value, such as information about a retired employee who no longer has an employee number.

To approach the problems listed above, every entity within RM/T is assigned a single unique system surrogate. E–Relations are unary relations which are employed to represent entity types, – they hold the system surrogate values of all the members of that type. All system surrogates are drawn from the E-domain and any attribute defined over this domain is called an E–attribute. P–Relations hold

the properties of the entity types, – their primary index is the particular type's system surrogate, however they may also contain user–controlled keys, if these are of use to the user.

Consider the following example where RM/T is used to represent the factory ordering database example described in the introduction. An E–Relation representing the entity type, supplier, might be associated with P–Relations describing orders given to the supplier and invoices received. Another E–Relation representing customers of the company, which may include some values present in the E–Relation of suppliers, might be linked with P–Relations representing orders received and invoices raised. Note that in this situation a P–Relation representing address information may contain both supplier and customer addresses.

Another type of relation, Property–Graph, PG–Relations have been defined within RM/T to describe the relationship between the E–Relations and P–Relations in the database. The PG–Relations are used to maintain consistency of the database by directing the actions of the insertion and deletion operations. For instance in this example the deletion of a supplier would cause deletion of all associated entries in the E–Relations describing orders given and invoices received. Note that in this situation, it would only cause deletion of the associated address entry if the supplier was not also a customer.

Instances of relationships may be treated as entities in RM/T and information supplied about them. These entities, called associative entities, are assigned a surrogate value which may be referenced in P–Relations to describe properties of the association, such as it's date or place. For example, an instance of the *buying* relationship between a person and an article could be given a surrogate value and additional information specified for it.

Two types of generalisation may be defined within RM/T, – unconditional and alternative. Unconditional generalisation is used to form the classic type–subtype hierarchy, where every member of the subtype must be a member of the parent type

and all properties of the parent type may be inherited by the subtypes. Multiple inheritance, resulting from an entity taking part in more than one hierarchy, is handled by the naive restriction that no entity can inherit two attributes with the same name, and thus there are no problems of conflict resolution.

Alternative generalisation is used to form subsets of the union of two or more types. For instance technical college students are an alternative generalisation of school–leavers, business–trainees and mature–students. A particular technical college student entered into the database must be either a school–leaver, business–trainee or mature student, however the reverse is not true since a member of the latter three types is not necessarily a technical college student. If a new technical college student was entered into the database, the alternative generalisation relation for these students would be consulted, and additional information would be requested in order to determine which E–Relation of the alternative generalisations the new entity should be placed in.

RM/T is only a theoretical model, it has not been implemented in the ten years since its definition and thus there is no implementation information evaluating the cost of various proposals. Although RM/T has succeeded in improving the referential integrity problems of the relational data model, described more fully in the next chapter, it still suffers from the same representational weakness problems which were elaborated in the previous chapter. Although many rules have been specified to maintain referential integrity within RM/T, similar rules have been defined for the original relational model which have proved to be too expensive to implement in practise. Integrity issues related to RM/T are discussed more fully in the chapter on database integrity.

3.3 Entity–Relationship Models

The Entity–Relationship, ER, model [8] is attributed by Hull and King [22, pp232] as being "one of the first true semantic data models to appear in the literature, although the term "semantic" was not in use at the time". Chen, [8], proposed the Entity–Relationship model as a data model to facilitate high level object centred schema design.

An entity is described by Chen [8, pp10] as "... a 'thing' which may be distinctly identified", such as physical objects or events. The ER model is composed of entities connected to each other via relationships. Entity sets are formed by grouping entities, according to one or more membership predicates, – the resultant entity sets need not be distinct.

Relationships are used to connect entities sets, and have the following properties :–

- A relationship may be defined on a single entity set , – thus the relationship *parent–company* could be defined on the *company* entity set. Thus unlike the traditional data models, a local cycle may be directly represented.
- A relationship may be defined on two or more entity sets, thus the tertiary relationship between students, courses and grades could be represented directly.
- It is possible to define more than one relationship on given entity sets, for instance the entity sets *company* and *name* could be linked by the relationships *company name* and *director name.*
- Relationships may be defined as one to one, one to many or many to many.
- Existence dependencies may be specified for a relationship, – these are described in the following paragraphs.

- Relationship sets may be formed by grouping instances of particular relationships.

Chen distinguishes between relationships which connect entities and attributes which connect entities to printable values. The printable values correspond to objects in the world possessing values which can be used for input and output, – such as characters, character strings or integers. Attributes are viewed as a single fact about an entity, consequently they map from an entity to a single value or tuple of values, – a multivalued attribute requires the use of a relationship. For instance, if humans were to be associated with phone numbers, where one human might have more than one number, a one to many relationship would be declared between *human* and *phone* which would then have a single valued attribute to *phone number*. Relationships themselves may also have attributes, such as the attribute *duration* on the relationship *married* between two members of the HUMAN entity set.

Existence dependencies may be specified in the ER model and these describe the relationship between instances of entity sets. Phone number instances could be declared dependent on human instances in order to ensure that if a human is deleted from the database then the corresponding phone numbers are also deleted.

The Entity–Relationship model has not been incorporated into a DBMS and thus Chen describes how an ER database design model may be represented in terms of a relational database implementation. Information about entities and relationships should be stored separately in entity and relationship relations respectively. Each tuple in an entity relation represents an entity and each column represents a value set from which the individual tuple values will be drawn. Each tuple in a relationship relation is assigned a unique identifier and represents a particular relationship by detailing the specific entities involved in it.

Chen [8, pp25] describes how various manipulation operations on the ER model of the database should be performed within the relational implementation. For in-

stance the deletion of an entity should be performed by deleting the entity tuple, and recursively deleting any entity tuple whose existence depends on it as well as associated relationship tuples. The definition of such existence dependencies is difficult or impossible using the facilities provided by relational databases and thus they must be coded by the database implementor into the application programs accessing the database. A discussion of maintenance of referential integrity within the relational model may be found in the later chapter on database integrity. Chen also does not mention the difficulties associated with modelling many to many relationships within the relational data model since these may not be directly modelled, as discussed in the earlier chapter on traditional data models.

The ER model is less semantically rich compared to the more recent semantic models described below, – it has no grouping constructors and until recently [46] inheritance hierarchies could not be defined. The manual conversion of the Entity–Relationship data model into a relational model implementation is laborious, error–prone and loses semantic information captured in the ER description. The ER model is useful as a database design tool, however its utility would be greatly increased if it was incorporated into a full database management system.

3.4 Semantic Database Model

The Semantic Database Model, SDM [20], was developed by Hammer and McLeod to provide a rich database description representation able to capture more of the semantics of the application environment. Although SDM may only be used as an abstract database model, the authors recognise the utility of incorporating it into a DBMS.

As well as attempting to provide a semantically rich formal database specification mechanism, SDM was created with two related objectives. Firstly to facilitate the creation of a high level user interface to the database in order to aid in the

identification and retrieval of information, and secondly to support the structured design of database applications.

The following principles underly the design of SDM [20, pp355] :-

- To organise the database as a collection of *entities* corresponding to entities in the application environment.
- To group the entities in meaningful collections, called *classes.*
- To relate the classes in the database by means of *interclass connections.*
- *Attributes* may be specified to describe characteristics of entities and classes as well as associating them with each other. Derived attribute values computed from values elsewhere in the database are also supported.

Classes in SDM are formed from homogeneous collections of entities, such as concrete physical objects, events or higher level objects like groupings of classes. SDM enables a distinction to be made between member and set properties of a class. Member attributes are those which apply to each member of a class individually, – such as the attributes *name, telephone number* and *address* of the class PEOPLE, and these inherit from class to subclass. In contrast class attributes apply to a class as a whole, such as the attribute *number of members*, – these attributes do not apply to the individual members of the class and may not be inherited by subclasses. However, SDM does not distinguish between member properties of a class specified for each member of the class individually and those specified once for the class and inherited by all the members. Consequently the attribute *Absolute_legal_top_speed* is specified as a class attribute of OIL_TANKERS [20, pp357], although it is a member attribute which need only be specified once for the class.

Every entity is the member of exactly one *base class* and zero or more *nonbase classes* which are contained in the base classes. The base classes enable an abstraction limit to be defined, otherwise all classes would have to be grouped together

into one root class. Groups of attributes are defined for each base class, which act as the primary key to uniquely identify each member of that class.

Classed in SDM are related via two types of interclass connection, – the *subclass* and *grouping* connections. A subclass connection defines a class which contains a subset of the members of its parent class, – an entity may be the member of more than one subclass, for instance an individual might be a member of the subclasses LAWYER, SQUASH_PLAYER and WEALTHY_PEOPLE of the class HUMAN. Subclasses in SDM are specified by defining a class and a predicate on that class which restricts the membership to form a subclass.

There are four different ways in which this subclass definition may actually be specified :–

1. A predicate may be defined on one of the member attributes of a class to form a subclass. For example the subclass SMALL_BUSINESS might be formed from the class BUSINESS where *number of employees* is less than six.

2. The user may be allowed to specify the members of the subclass. The subclass STUDENT_ON_REPORT might be a specified subclass of STUDENT.

3. A subclass may be defined via set operations specified between other classes. The subclass RAQUETSPORTSPLAYERS is a subclass of SPORTSPLAYERS that contains the union of the classes TENNIS_PLAYERS, BADMINGTON_PLAYERS and SQUASH_PLAYERS.

4. A subclass may be defined as containing members which are currently attribute values of another class. For instance MOTORING_FATALITIES is a subset of PEOPLE satisfying the predicate *fatal victim* of the class ROAD ACCIDENT.

The grouping inter-class connection allows the collection of entities of a similar type into groups of a higher conceptual level, which can themselves be treated as

classes. In a similar manner to the subclass connection there are a variety of ways in which a grouping connection may be specified. A member predicate may be used to group entities together on common value, for instance the predicate *racing status* may be used to group the class OARSMEN into TYPES_OF_OARSMEN. An enumerated grouping class may be formed from a list of classes to be included in the group, where all the named classes are subclasses of one underlying class. For instance the class HAZARDOUS_TYPES_OF_FOOD is a grouping of FOOD-STUFFS consisting of classes FAST_FOOD, POULTRY and EGG_PRODUCTS. The user controlled grouping class is a variation on the previous grouping method where the user specifies the classes to be included in the grouping.

Attributes are used to connect classes with printable values, – meta–information must be supplied about each attribute, the most relevant of which is summarised below :-

- Applicability – whether the attribute is a member or class attribute, discussed above.
- Single or multivalued, – the value of a single valued attribute is a single member of its value class, whereas the value of a multivalued attribute is a subclass of the value class.
- Mandatory – A null value is not allowed for the attribute.
- Not Changeable – Specification of the value is a 'one shot' process, and thus it may not subsequently be changed.
- Exhaustive - Every member of the value class of the attribute must be the value of some entity.
- Nonoverlapping - Each member of the value class is used at most once.

SDM provides many different facilities for defining derived attributes, in order to support data relativism, – the ability to view the same data in many different ways.

Amongst some of the many ways of specifying derived attributes are arithmetic and set operations on member attribute values, as well as cardinality and ordering functions.

SDM is a successful abstract database modelling tool, providing a rich variety of data modelling constructs and attribute domain information. An implementation of SDM would enable a user to interact with the database schema definition and learn about the database, however SDM has not been implemented either as a database model or incorporated into a DBMS. Another benefit of implementing SDM would be the automation of checks on the legality of the schema , such as the prevention of circular sub–class definitions. It must be realised that the richness of the representation cannot be directly translated into a traditional database schema for two reasons. Firstly the domain information specified for attributes cannot be expressed or enforced by the constructs provided by the traditional database systems. Secondly the translation of the model into a traditional database implementation would lose most of the object based form of the information expressed in the semantic model, since a schema in a traditional databases must be expressed in a lower level record oriented manner.

3.5 IFO : A Formal Semantic Database Model

The IFO Semantic Database Model [1] is a more recent model than SDM and provides a similarly rich variety of information modelling constructs. IFO was designed for the investigation of semantic modelling issues, such as the types of objects which may be created by the combination of the modelling constructs provided and the propagational effects of updating data.

The object types within the IFO model can be divided into three *atomic* types, printable, abstract and free, all other object types which are constructed from these are *non–atomic*. The printable objects correspond to concrete objects in the world

which can be used for input and output, as described for the Entity–Relationship model. The abstract objects correspond to objects which have no underlying structure with respect to the database designer or user, – they correspond to base classes in SDM. Abstract objects cannot be referenced directly, but are accessed via their attributes, such as the the attribute *name* of the abstract object 'person'. The free object type corresponds to objects whose structure is inherited via *is–a* relationships and these correspond to non–base classes in SDM. Whether an object is abstract or free depends completely on the database schema in which it takes part, – in a schema consisting of computers and personal computers, the former would be an abstract object and the latter a free object.

Atomic and non–atomic object types may be combined to form non–atomic types via the grouping and aggregation type constructors. The grouping, or finitary power set, operator combines groups of objects of the same type, – for instance the grouping of cars into the object type, car fleet. The aggregation, or Cartesian product, operator combines objects of differing types into one supertype, – such as the combination of the types keyboard, screen and base into the supertype workstation.

IFO distinguishes between two forms of the *is–a* relationship and these may be used to relate the atomic and constructed types to each other. Specialisation defines possible roles for members of a particular type, for instance the type 'tradesman' could be specialised into 'butcher', 'baker' and 'painter'. A member of the supertype could feasibly play the role of any of the specific subtypes without changing its fundamental identity, however if the supertype is labelled disjoint then a member of the supertype may only be a member of one of the subtypes. Generalisation combines distinct types to form new super–types, such as generalising the objects 'tables', 'chairs' and 'stools' to one super–type 'furniture'. A member of the supertype may only be a member of one of the subtypes, however if the supertype is labelled 'covers' then it indicates that the union of the members of the subtypes forms the supertype.

The behaviour of the IFO model under update requests is considered with respect to two types of request, the first is the modification of an attribute value, the second is modification of the members associated with an object type. The former type of modification does not produce any propagational effects if the change does not affect the particular instance whose attribute value is being changed. The latter type of modification has more complex propagational effects when takings into account the type constructors and *is–a* arcs which may be incorporated in the system. The study of update propagation enables a fuller understanding of how a working DBMS based on the IFO representation would operate, however the study has not been completely addressed, since manipulation of the schema has not been taken into account.

IFO has not been incorporated into an actual database management system, it is purely a database schema modelling tool, although the representation has been used in the SNAP [6] schema design system. The work on IFO has specifically concentrated on data representation issues, – future work on IFO is viewed as the incorporation of integrity constraints and the development of an appropriate query language for the model.

3.6 Discussion

The models presented in the chapter have been general purpose data models, however models with features suited for more specific application domains have been designed. Narayanaswamy and Bapa Rao [30] propose a data model suited to use in engineering domains since it has explicit constructs to model the constant revisions of the schema and Su [44] proposes the SAM* model for modelling manufacturing data. The Information Systems Designer, INSYDE [25], was developed specifically for use in the design of office information systems and provides constructs for the definition of the structure of the data and the processes which will

manipulate it. Thus using our factory database example, typical processes would include receipt of an order, order input and invoice production.

The data models presented in this chapter have varied in the complexity of modelling constructs provided from the simplicity of the ER model to the more recent and rich IFO model. Compared with a traditional database schema, the models presented in this chapter have incorporated more of the semantics of the information, enabling the model to more truly reflect the application environment. Only the ER and INSYDE models have been implemented as prototype semantic database models the other models must be constructed manually. All of the semantic models need to be translated into a traditional database implementation as none of them have been incorporated into a database management system. Manual translation into a traditional database implementation is laborious, error–prone and loses a lot of the semantics expressed in the model since the databases cannot represent the semantic richness of the model.

A semantic model may be useful in the initial stages of design of a database to facilitate easier communication of the database structure between the interested parties, however once it has been translated into a traditional implementation its utility is greatly reduced, since potential inconsistencies may arise between the database model and the database itself. Initially the database implementation will not fully reflect the semantic model and subsequently the model may not be updated to reflect changes to the schema of the implementation.

Great benefits would result from the incorporation of semantic models into database management systems. The semantic model is very useful in the design stage for facilitating communication and modelling the application domain. This model would form the underlying representation of the DBMS and thus the problems of conversion between the model and the internal representation would be eliminated.

Chapter 4

Database Integrity

Having presented the traditional and semantic data models in the previous two chapters, the following two chapters consolidate this work by considering two specific issues within these models, – integrity of database data and homogeneity of information representation.

The issue of integrity is very important in databases and concerns the control of data in the database to ensure it agrees with its meta–data and application domain it is modelling. This chapter reviews the different levels in a system at which integrity may be compromised, the different types of integrity hazard and methods of integrity enforcement. Integrity issues are discussed with respect to both traditional and semantic database models.

Security and integrity maintenance subsystems are complementary since the former protects system resources from unauthorised access and manipulation whilst the latter protects resources during authorised access. However, the DBMS proposed in this thesis is a single user prototype system, – security issues are not relevant and thus they are not addressed.

4.1 Introduction

The traditional data models, presented earlier in this thesis, are record oriented, efficient database systems. The semantic database models, described in the previous chapter, enable a more detailed description of an application environment to be formulated than with the traditional data models, by use of a rich set of information constructs. Although many semantic models have been formulated and used in the design of database systems, few have been automated and none have been incorporated into a DBMS.

This thesis concentrates on two problems which have been identified with data stored in databases. The first problem concerns the integrity of the data, – Frost [18, pp24] defines database integrity as "a property which reflects the extent to which the database is an accurate model of that part of the universe which it represents". The approaches taken by the various traditional models to maintain integrity are described and contrasted with the facilities provided by the semantic models. The second problem concerns the homogeneity of information representation in databases. The following chapter concentrates on the issues of the homogeneity of information representation in the two types of model, and describes the consequences of information inhomogeneity. The consideration of integrity and homogeneity issues combined with the knowledge of traditional and semantic models led to the proposal of a new database model, BIRD, described later in this thesis.

4.2 Conceptual Levels of Information in a Database

There are many different conceptual levels at which the integrity of a database system may be compromised :–

System Level Damage sustained to physical blocks on a disk drive may lose part of the database, or a system crash during the execution of a transaction may leave the database in an inconsistent state due to the loss of main memory contents.

DBMS Level Inconsistencies may be introduced in many ways due to software faults in the DBMS, such as incorrect handling of concurrent updates, or mismanagement of pointer values.

User Level The input of erroneous data by the user, such as null values, values out of range or duplicate values. The data may contradict the application independent rules of the data model, or rules derived from the application environment being modelling.

4.3 Database Integrity Constraints

Integrity maintenance at the system and DBMS levels is application independent and is built into the operating system and DBMS. At the user level, explicit integrity constraints may be used to augment the description of the application environment.

Owing to the different representational powers of the data models, a constraint which may need to be specified explicitly in one model may be implicit in another. For instance, referential integrity is implicitly enforced in the HDM and yet it may be defined explicitly in the NDM using the 'connect' and 'disconnect' statements, – this is discussed later.

Explicit integrity constraints may be classified according to whether they govern *structural* or *semantic* aspects of the data and the *granularity* of the data affected.

Structural constraints govern the form of the application data in the database,

such as ensuring that all orders in a database are associated with a date, or all people are associated with one or more phone numbers. For example the "create relation" command within a relational database system is a structural constraint since it provides information about the structure of tuples in a relation of the database. Semantic constraints govern the actual values assigned to items in the database, such as ensuring that amounts of money are greater than zero or that the total budget of a company adds up to the sum of the individual departmental budgets. The granularity of a constraint refers to the level of data structure it governs, – this may vary from individual field values to record types, implying a high to low enforcement cost respectively.

4.4 Treatment of Integrity within Traditional Databases

The facilities provided by the traditional data models for integrity constraint definition and maintenance are compared and contrasted in this section. The relative merits of the different ways of declaring and enforcing integrity constraints are also discussed.

4.4.1 Definition of Explicit Integrity Constraints

The traditional data models provide varying facilities for user definition of explicit integrity constraints. The most rudimentary facilities are provided by the hierarchical data model where integrity constraint definition is limited to the declaration of types for the application data values, such as integer, real or date.

The network data model provides better facilities for integrity maintenance than the hierarchical data model. CODASYL provides a simple range checking

statement, CHECK [31], which may be used to define legal and illegal ranges of values. The form of the CHECK statement is as follows :-

CHECK IS VALUE [NOT] literal-1 [THRU literal-2]

[, literal-3 [THRU literal-4]]

The sparse consistency definition facilities of the CHECK statement may be augmented in CODASYL by the definition of database procedures, – these may be written in any suitable programming language and interface with the DBMS via schema statements which detail their preconditions for execution. Typical preconditions might be the execution of a particular operation or the occurrence of an attribute falling within or outside a specified range.

There are disadvantages to using external procedures which are detailed below :–

- External procedures are not part of the database since they are external to the database possibly written in a variety of programming languages. Database users may neither define, query or alter the constraints which the procedures enforce, consequently a user may not understand why a transaction has been rejected or be able to alter the constraints to reflect a changing application environment.

- Definition and maintenance of external procedures would generally be carried out by the database administrator who understands the programming language in which they are written. The inconvenience of inserting and maintaining the integrity procedures makes it likely that only the most important constraints would be defined, and these may easily become outdated unless sufficient effort is made in maintaining them.

- Constraints expressed in external procedures are not administered by the system and thus there is no possibility of the DBMS checking their consistency, redundancy and completeness.

An alternative to external constraint procedures often used in traditional database applications is to incorporate the constraints into the application programs which access the database, however this approach has the associated disadvantages described below :-

- All the disadvantages of external constraint enforcement procedures, which have been detailed above, apply to the enforcement of integrity through application programs.
- Some users may access the database via the database query language, bypassing the integrity controls.
- The constraints applying to a particular relation must be duplicated in every application program which accesses that relation. Duplication of constraints is laborious and prone to inconsistency, particularly during maintenance, since every occurrence of the same constraint may not be updated to reflect changes in the application environment.

The CHECK statement of the network model is present in a similar form in the relational model, such as the "create integrity" statement in the INGRES 5.0 Relational Database [23]. An example of such a statement, expressed in SQL is shown below :-

CREATE INTEGRITY ON ORDER IS order.amt $\leq$ 4000

The above constraint specifies that the maximum value in the amount attribute of the order relation is four thousand.

Integrity constraints within INGRES are subject to the following restrictions :-

- The integrity expression must only reference attributes from a single relation. This has important ramifications for the maintenance of referential integrity which are discussed later.
- No aggregates are allowed in the integrity expression, - such as SUM, AVERAGE. Aggregates necessitate the retrieval of every tuple in the relation named in the expression, and this would significantly slow the execution of the assertion.
- The COPY command which copies tables into other tables bypasses the integrity constraints specified. This restriction has been made in the interests of speed, since checking constraints for every tuple which is copied would significantly slow the execution of the command over large relations.

The above restrictions limit the scope of the integrity statement, - the integrities section in the INGRES system manuals concludes with the advice that [23, pp5-18] "...it may be simpler and faster to check for valid values from within your program before issuing the query". Checking of values by application programs may be more efficient than the use of general purpose integrity enforcement facilities, however, the many disadvantages of this approach have already been detailed above.

4.4.2 Enforcement of Referential Integrity

Referential integrity is an important issue in databases, since it ensures that an entity exists in the database when information is supplied about it. For instance referential integrity constraints might ensure that a bank account actually exists when a credit is specified for that account. The hierarchical model implicitly enforces referential integrity, since all records in the database must be associated with the parent record type they describe, with the exception of the root record. If a

bank account record were deleted in the hierarchical model, all information associated with that account would also be deleted and no further information could be inserted about that particular account.

It is not necessarily desirable for a model to implicitly enforce referential integrity, since one may wish to retain information records in a database even when a parent record does not exist. Such a situation has already been described in the chapter on traditional data models, – where one might wish to retain information about parts in a factory parts ordering database even though there were no orders currently referencing those parts.

The network data model takes a more flexible approach to referential integrity by means of set insertion and retention specifications, which specify the relationship between instances of set types and their member records. A storage class may be specified as either manual, in which case connection of a record into the set type is left to the user's discretion, or automatic, in which case the new record will always be connected into a set of the appropriate set type. The removal class specifys rules for disconnecting records from their set types, and three modes may be specified, – fixed, mandatory or optional. If the retention is fixed, then member record cannot be disconnected from the set into which it has been inserted without being deleted. If the retention is mandatory, then a member record can only be disconnected from a set if it is reconnected to a set of the same type. If the retention is optional, then a member record can be disconnected from that set occurrence at will.

For the bank account example, mentioned above, the storage class of transaction records for a bank account would be automatic and the removal class would be fixed. Network model implementations take different precautions to ensure that disconnected records cannot remain in the database if there is no means to subsequently access them, – such as enforcing the restriction that a record type must be an automatic member in at least one set type.

The importance and utility of integrity specification facilities is often sadly

underestimated as they are seen as unnecessary infringements, – for instance Olle [31, pp80] in referring to the NDM states that "If the data administrator wishes to allow flexibility, he would prefer the optional (removal class) alternative every time."

Date [11, pp89] defines referential integrity within the relational model as follows."Let D be a primary domain, and let R_1 be a relation with an attribute A that is defined on D. Then, at any given time, each value of A in R_1 must be either (a) null, or (b) equal to V, say, where V is the primary key value of some tuple in some relation R_2 (R_1 and R_2 not necessarily distinct) with primary key defined on D." Here a primary domain is one which has a single attribute primary key defined over it.

The relational model does not implicitly enforce referential integrity, this may possibly be effected using the constraint specification mechanisms or building the information into the application programs which access the database. The disadvantages of building constraints into the application programs have already been elucidated in this chapter and the constraint specification mechanisms may not be powerful enough to enforce referential integrity. Integrity constraints in INGRES 5.0 may not reference more than one relation in the constraint expression which precludes them being used in general referential integrity enforcement, apart from the situation where one wishes to check referential integrity within a single relation. Situations in which one checks referential integrity within a relation are very uncommon, – the process of normalisation increases the number of relations and makes inter–relation referential integrity checks very desirable.

4.4.3 Integrity Maintenance at the System Level

It is important that the integrity of data in a database system is protected against unexpected events, such as a system crash, since a transaction may have

been interrupted in the middle of execution. Mechanisms such as "roll–back" have been introduced which record the operations being performed on a database and attempt to replay the failed operations on the pre–crash version of the database. Such mechanisms are not relevant since the thesis concentrates on issues of integrity in a prototype DBMS. Readers who require more information on integrity maintenance at the system level are referred to standard database texts such as Date [11].

4.5 Treatment of Integrity within Semantic Models

Semantic models attempt to capture as much of the semantics of the application environment as possible and consequently their representation is much richer than the schemas of traditional record oriented databases. The semantic models are able to embody much more consistency information than their traditional model counterparts, without the use of explicit integrity maintenance statements. The integrity facilities provided by the various semantic models discussed in the chapter on semantic data models are presented below. Since *none* of the semantic models have been incorporated into a working DBMS, the viability of the constructs in an implementation has not been evaluated. The integrity facilities of RM/T are discussed separately, since this model is so different to the other semantic models presented.

4.5.1 Integrity Issues within RM/T

RM/T was proposed in an effort to increase the consistency of the relational model, and this was effected by the introduction of system surrogates and the E–Relations, as described in the chapter on semantic data models.

Entity integrity is fundamental to the relational model and dictates that no component of a primary key of a base relation may be null or duplicate to ensure that every tuple in a relation may be uniquely identified. Entity integrity is enforced in RM/T since no E–Relation entry may be null and entries may only be deleted and inserted but not updated.

Rules have been formulated within RM/T to maintain referential integrity, such as the action upon deletion of a surrogate value, or the insertion of a tuple into a P–relation. The information in the PG–relations is crucial for integrity maintenance as it describes the P–relations in which the entities of the E–relations take part. When removing a surrogate value in the case where the entity in question is to be completely removed from the database, all tuples are removed which use the surrogate as the unique identifier, all other tuples have the surrogate identifier replaced by a special value meaning 'entity unknown'. In the case where one type or role for an entity is to be removed only the entries corresponding to that role and entities dependent on that role must be deleted. The above situation might arise in a factory database where a company is both a customer and supplier. If the company was deleted as a supplier, then tuples using the supplier's surrogate as the unique identifier would be deleted, and tuples which relied on the customer's surrogate would remain.

The facilities for referential consistency control are still rudimentary in RM/T, – the set removal and storage classes of CODASYL provide more information about the relationships between entities. Let us consider the factory parts ordering database represented in RM/T, – if the user deleted a tuple representing a particular part, then tuples in the order relations which referenced that part would have the part number replaced by the 'entity unknown' value, since the tuples would be indexed on the order number, not the part number. This action would leave the database in a consistent state with respect to referential integrity but not with respect to semantics, since a tuple detailing that an order refers to a particular quantity of an unknown part is meaningless. Increased expressiveness

in the representation of RM/T is needed to enable the user to specify that in this situation either the tuple referencing the deleted part is removed, or the complete order from the customer is cancelled.

4.5.2 Integrity Issues within other Semantic Models

The expressiveness of semantic modelling constructs enables the modelling of information which would have to be described using integrity constraints in the traditional data models. An example of this is the ability to define derived data in semantic models, – where a data value may be defined via an expression on other data values. For instance an attribute *annual total orders* could be defined as the sum of all the *monthly total orders*. To represent this information in a traditional data model a separate figure would have to be specified for the annual figure and a consistency constraint used to ensure it corresponded correctly with the monthly figures.

The increased modelling capabilities in semantic models avoid duplication of information and the use of explicit consistency constraints. It must be appreciated that derived data may be extremely expensive to implement in practise due to continual recalculation of many values every time one entity value is altered.

The information content of a semantic model is mainly contained in the relationships which are specified between entities. The semantic models allow many domains to be specified for these relationships which qualify how they may be used, enriching the expressiveness of the representation and aiding integrity maintenance.

Typical relationship domain information which may be specified is shown below :–

Name The name of the relationship.

Value class The name of the value class from which the relationship may take its value, such as ORDER_NUMBER, EMPLOYEE or STRING.

Single/Multi–Valued Single–valued relationships have values which are members of the value class, whereas multi–valued relationships have values which are a subset of the value class.

Key Whether the relationship provides a unique identifier for the object.

Mandatory Whether null values for the relationship value are allowed.

Member/Class Whether the relationship is a property of the members of the class or applies to the class as a whole.

One Shot Whether the value can be changed once set.

Exhaustive If an relationship is exhaustive, then its values for the entities in the database exhausts all the values in the value class of the relationship.

To illustrate how the domains might be used in practise let us consider an example of the *order number* relationship of the object ORDER in the hypothetical factory parts ordering database.

Name : OrderNumber

Value class : ORDER_NUMBER

Single/Multi–valued : Single–valued

Key : TRUE

Mandatory : TRUE

Member/Class : Member

One Shot : TRUE

Exhaustive : FALSE

The single/multi–valued and mandatory domains are of crucial importance in maintaining referential integrity since they define the appropriate action to be taken if one of the arguments of the relationship is inserted or deleted. Consider the insertion of an order into the database, – since the *order number* relationship is mandatory and single valued, a single order number would have to be supplied. If that order number were subsequently deleted then the associated order would also be deleted since the *order number* relationship is mandatory.

The semantic model also provides increased power over the definition of value classes which enables more precise control to be exercised over relationship values. The definition of a class in the semantic model might allow the following types of descriptions, (these have already been mentioned with respect to SDM in the chapter on semantic models) :–

Class Type If the class is printable, that is the values of its members can be outputted, then its type may be described as one of the simple input/output types such as string, real, integer or date.

Set Combination A class may be defined via set operators between two or more sets. For instance the class WORKING POPULATION could be defined as the difference between the class of HUMANS and the union of the PENSIONER and CHILDREN classes.

Restricted Class A class may be constructed from the members of another class which fulfill a specified predicate. This type of specification may be used to create sub–ranges of standard value classes, such as ORDER NUMBER is the class INTEGER where value is less than ten thousand. It would also be possible to use restricted classes to form classes whose members may only be computable at runtime, such as the the class SPRINTERS which is the class RUNNERS where physique is large.

Integrity issues specifically related to implementation, such as recovery from

crashes and methods of database restructuring have not been dealt with by any of the models, since none of them have been incorporated into a DBMS.

4.6 Discussion

Integrity maintenance is a facility which has not been well developed within any of the traditional record oriented databases. The network data model allows referential integrity to be defined via the storage and removal class specifications and provides a limited range checking statement. The relational data model enables integrity constraints to be defined via assertion statements, however the representation is limited and implementation overheads may be high. The cost of integrity maintenance is of paramount importance in realistic database applications, as this must be balanced against the value of the data being protected.

The semantic models intrinsically provide a high level of integrity maintenance by virtue of their expressive representations, however the lavishness of the representation may be unrealistic when compared to their benefits in an implementation. The representations embodied in semantic models intrinsically enable a higher level of information integrity than the traditional models even including the use of assertion statements. Advanced representation facilities such as derived data and relationship domain descriptions are extremely useful in integrity maintenance but are likely to prove unrealistic in an implementation. Semantic models need to be incorporated into a DBMS in order to evaluate the cost of the representation.

Chapter 5

Homogeneity of Information Representation in Databases

The previous chapter concentrated upon the issue of integrity in the traditional and semantic data models. This chapter considers the issue of homogeneity of information representation in the same models. The conclusions gained from these two chapters lead to the formulation of the BIRD model, described in the next chapter.

Information entered by users which is stored in database systems comprises the schema and application data. This chapter examines the extent to which this information is stored in a homogeneous manner and the effect which the choice of information representation has on the utility of the database and the user interface.

5.1 Homogeneity of Information in Record Oriented Databases

This section describes how application and schema information is stored in traditional databases, and how this information may be manipulated. The hierarchical data model is not discussed since the features which are examined in this chapter are similar to the network model. Note that the figures in this section which show tabular output from the INGRES relational database have been truncated in length and width in order to fit them onto the page.

5.1.1 Representation and Manipulation of Schema Information

This section initially describes how schema information is represented in the traditional data models. The manipulation of schema information is presented, – integrity information is dealt with separately since it is treated differently to the other schema information.

The schema information comprises a structural description of the application data, possibly augmented with integrity information detailing constraints on the structure or values of the application data. Schema definition statements for the relational and network models are shown in figure 5.1, expressed in the Data Description Language, DDL, of the system. The schema statements show part of the definitions which might be used to define the hypothetical factory parts database. The DDL statements are compiled and stored in the system data dictionary and this description may then be accessed by the system database manager in order to control the application data admitted to the system.

The data dictionary in INGRES 5.0 comprises a number of "system catalogues",

```
SET NAME IS Factory-Order

    OWNER IS Order

    MEMBER IS Order-Item ;


RECORD NAME IS Order-Item

LOCATION MODE IS CALC USING Part-Number

    DUPLICATES ARE NOT ALLOWED ;

02 Part-Number ; PICTURE IS 9( 6) ;

02 Quantity    ; PICTURE is 9( 4) ;

    CHECK IS VALUE NOT 0 ;
```

```
CREATE TABLE Factory-Order( Order-No i4 Part-No i6 , Quan i4) ;

DEFINE INTEGRITY ON Factory-Order IS

    Factory-Order.Quantity > 0 ;
```

Figure 5.1: Schema Definition Statements in the Network Data Model (top) and Relational Data Model (below)

examples of these include the *relation* catalogue which stores a tuple for every relation in the database, and the *attribute* catalogue which stores a tuple for very attribute of every relation in the database. An example of the contents of the *relation* catalogue is shown in figure 5.2

Although a schema may be added to with ease, the process of changing an existing schema, database restructuring, is more complex and may be effected in different ways according to the facilities provided by the DBMS :-

1. No facilities may be provided by the DBMS, in which case a new schema must be constructed and the old database loaded into the new database via an application program. The database will obviously be unavailable to users whilst this process is taking place.

2. A Database Restructuring Language, DRL, may be provided which enables the user to change portions of the schema. In this case the database itself should be automatically altered to reflect the changes in the schema. The database may be unavailable for use whilst this process is taking place, or it may be possible to segment the part of the database which is being restructured and allow users access to the rest.

3. Use of the data description language to effect restructuring. The relational model can best cope with restructuring when compared with the other models due to its lack of explicit pointers, it is also the most convenient since restructuring may take place whilst the database is being used. In order to restructure a relation, a new relation is defined with the appropriate attributes; information is copied from the old to the new relation; information is provided for any new attributes in the new relation; the old relation is deleted and the new relation renamed.

Querying of schema information within the relational model is effected by the use of specific commands. In order to view any of the information in the system

relid	relown	relatt	relwid	relspe	relstat	reltups
abfobjs	a6	11	154	7	1065473	0
indexes	a6	10	33	7	1052691	3
qbfmap	a6	5	40	7	1065473	0
rcommands	a6	8	154	11	1065473	0
relation	a6	25	160	7	1052691	26
integrities	a6	8	33	7	1052691	8
protect	a6	16	52	7	1052691	0
zopt2stat	a6	61	246	7	1064977	0
abfappl	a6	7	136	7	1065473	0
gcommands	a6	9	309	5	1065473	0
iicompfrm	a6	7	1990	7	1064961	0
attribute	a6	8	35	7	1052691	278
fdfields	a6	26	497	7	1065473	0
fdframes	a6	12	40	7	1064961	0
graphs	a6	13	60	7	1064961	0
iiqbfinfo	a6	7	78	7	1064961	0
tree	a6	7	120	7	1052947	24
zopt1stat	a6	10	42	7	1064977	0
invorder	a6	3	12	3	1191944	23
reports	a6	8	29	7	1064961	0
invoice	a6	2	16	3	1191944	11
invindex	a6	2	8	5	1060994	11
fdtrim	a6	5	168	7	1065473	0

intrelid	intrel	inttre	intdomset1	intdomset2	intdomset3
address	a6	0	33554432	0	0
invorder	a6	0	33554432	0	0
address	a6	1	33554432	0	0
invorder	a6	1	33554432	0	0
invorder	a6	2	67108864	0	0
invorder	a6	3	134217728	0	0
invorder	a6	4	67108864	0	0
invoice	a6	0	33554432	0	0

Figure 5.2: Contents of the System Catalogues "relation" (top) and "integrities" (below) in INGRES 5.0 Relational Database

catalogues, the user must use the "HELP" command. To retrieve information about a particular relation in the database, the command "HELP relation–name" would be issued, – figure 5.3 displays the output of the command "HELP relation", which describes the system catalogue named "relation". Although the contents of the data dictionary could all be altered by operating directly on the system catalogues using the query language, this practise is strongly discouraged since it may result in an erroneous system state. Instead the schema information in the data dictionary is indirectly manipulated by operating on the relations it describes using the data description language.

Integrity information may be supplied as part of the schema definition, – see the 'DEFINE' statement in figure 5.1, and this information is also stored in the data dictionary. INGRES stores integrity information in a particular system catalogue named "integrities" and an example of the contents of this relation is shown in figure 5.2. In the case where external procedures are used to perform integrity checking, only the definition of the trigger conditions will be present in the data dictionary. Integrity checking may also be built into the application programs which access the database, and in this case no integrity information at all is stored in the data dictionary.

Integrity information is an exception since it does not govern the structure of the relations and thus cannot be indirectly manipulated by operating on the structure of the relations. Explicit commands are defined for the manipulation of integrity information, – insertion is effected by the "CREATE INTEGRITY" command, deletion by the "DROP INTEGRITY " command and querying by the 'HELP INTEGRITY' command shown in figure 5.2.

```
Name:                   relation
Owner:                  col0
Location:               col0
Type:                   system catalog
Row width:              160
Number of rows:         26
Storage structure:      hash
Number of pages:        7
Overflow data pages:    0
Journaling:             disabled
Optimizer statistics:   none
Column information:

                                          key
 column name     type     length     sequence
 relid           c           12            1
 relowner        c            2
 relatts         integer      2
 relwid          integer      2
 relspec         integer      2
 relstat         integer      4
 reltups         integer      4
```

```
/*      Integrity constraints on invorder are:  */

/*      Integrity constraint 0 -        */

range of i is invorder
define integrity on i is
              (i.invnum >= 0)


/*      Integrity constraint 1 -        */

range of i is invorder
define integrity on i is
              (i.invamt >= 0)
```

Figure 5.3: Sample Output of the "help " Command in INGRES 5.0 Relational Database – "HELP Relation" (top) and "HELP INTEGRITY ON InvOrder" (bottom)

```
SELECT SerialNum, Quantity
FROM   InvOrder , Invoice
WHERE  Invoice.InvNum = InvOrder.InvNum


SELECT   SerialNum, Quantity
FROM     InvOrder
ORDER BY SerialNum


DELETE InvOrder
WHERE  InvNum = 100234


INSERT INTO address
     VALUES ("15 Squires Lane","DURHAM","DH1 3JD")


CREATE VIEW TownInfo AS
    SELECT InvNum, Town
    FROM   Address
```

Figure 5.4: Examples of Standard Query Language, SQL, Commands

5.1.2 Representation and Manipulation of Application Data

The reader is referred to the chapter on traditional data models for a description of how application data is represented and manipulated in the various record-oriented data models. Figure 5.4 shows examples of the usage of a data manipulation language called the Standard Query Language, SQL, to perform insertions, deletions and queries on the relational data model.

5.1.3 Discussion

The inhomogeneity of information representation in the traditional databases necessitates multiple languages to manipulate the different representations of information. All of the traditional databases require the knowledge of a Data Description Language, DDL, to define the database and a Data Manipulation Language, DML, to access the data in the database. In addition there may exist a Database Restructuring Language, DRL, to effect changes to the schema structure, a Data Dictionary Manipulation Language, DDML, to alter information in the data dictionaries and a Data Strategy Description Language, DSDL, [31, pp207] to control storage, buffering and paging in the database system. For example, INGRES 5.0 has sets of commands to create the relational structure, query application data, query the data dictionary and control the storage of the database.

The need to learn multiple languages is laborious and limits the scope to which users may manipulate the information in the database. For instance if there was one common DDL, DML and DDML operating over a common information representation then a user experiencing problems in accessing a database would be able to use the same language to query the schema and discover the structure of the database and any pertinent integrity constraints which have been defined. With different languages, the same user must expend extra effort in order to determine how to manipulate information at a different conceptual level.

The different information representations restrict the information to which each language has access, – a common database information representation would enable statements in the manipulation language to access schema, integrity and data information. An example which demonstrates the utility of this feature is the situation where one wishes to append the contents of relation, A, onto the end of relation, B. With a common DDL, DML and DDML one could form a query which retrieved the constraints on each attribute of relation B and tested the corresponding attributes of the tuples in relation A to see if any constraints were violated,

before copying the tuples over.

With a common information representation, application programs built on top of the database would be easier to write, more maintainable and have the potential to be more powerful. For example if one desired to write an application program which requested the attribute values for all the fields of a particular relation, then the application program could interrogate the schema to determine which attributes exist before requesting the values. The application program would be simpler to write, the code would be reusable for the other relations in the system and maintenance would be enhanced since the relations could be restructured without affecting the application program.

5.2 Homogeneity of Information in Semantic Models

The representation of information within semantic models has already been described in previous chapters. None of the semantic representations have been built into a DBMS and thus it is not possible to comment on the possible inhomogeneity between schema and application data. If semantic models are used in the design of databases and then translated into a traditional implementation, then there will be inhomogeneity between the semantic model, the traditional database schema and the application data.

5.3 Related Work

Gray et al [19] criticise the lack of sufficient database metadata to help users interact effectively with databases, consequently they developed a representation,

MAKR, to capture database metadata. The authors envisage a metadata adviser which is able to give both descriptive and procedural information about the structure of the database and the information within it. The representation MAKR was developed based on artificial intelligence style knowledge nets and this is used to represent schema and meta–schema information. MAKR is a rich representation specifically tailored to the capture of schema and meta–schema information, however it is still being developed and has not been incorporated into a metadata adviser or integrated into a DBMS.

The EDICT model [12] has been proposed by Davis and Bonnell as an enhancement to relational database systems which captures more of the semantics of the application environment and enhances the utility of the data dictionary. The authors propose the storage of information at various levels :–

1. Enterprise Schema – A DBMS independent model of the application domain expressed in a suitable semantic model, such as the ER model.

2. Enterprise Meta–Schema – description of the structure of the enterprise schema, stored in the data dictionary.

3. Conceptual Schema – the relational database schema generated from the enterprise schema.

4. Application Data – the actual enterprise data stored in the database.

5. Internal Schema – description of the underlying storage structures, created and accessed by the DBMS. This describes how the database is physically stored and indexed.

6. Dictionary Schema – contains information about the conceptual schema, such as the number of relations, attributes and integrity information.

7. Dictionary Meta–Schema – information about the data dictionary, such as the names of the relations in the dictionary and the meanings of the attributes.

The authors postulate that since all of the above information apart from the internal and enterprise schemas are stored in relations, they may be accessed using the same query language. Unfortunately, as demonstrated earlier in this chapter, the fact that you can access the data dictionary catalogues using the query language does not mean you can understand the information found therein. System catalogues may be filled with copious amounts of unintelligible information, which is not easy to interpret even with a dictionary meta–schema to help explain. Other problems with this approach is that the relational model is not expressive enough to capture the sort of information discussed. For instance Davis and Bonnell use the ER model as an example of the enterprise schema and state that in this model [12, pp187] "... an attribute is either associated with an entity set or a relationship set , but not both and not neither", – this information would be impossible to capture in the enterprise meta–schema.

The motivation of the EDICT project, to provide enhanced database meta–information and represent information homogeneously is valid, however success has been limited due to the reliance on the relational data model. The project highlights the problem of choosing a suitable representation to store information at varying conceptual levels, – although the relational model may have many benefits at the application data level, the representation is not adequately expressive at the enterprise meta–schema level. The adoption of a much richer representation, such as in the MAKR model, is associated with many implementation problems. To actually implement a representation such as MAKR would be difficult plus storage requirements and application access times may be greatly increased. Davis and Bonnell also do not discuss the manipulation of information at varying levels and how the effects propagate to other levels.

5.4 Summary

Current databases employ different representations for information at differing conceptual levels. This inhomogeneity complicates a user's perception of the database, restricts information sharing and makes application programs less flexible. Although homogeneity of information is desirable, a representation suitable for information at one conceptual level may not be able to efficiently express information at another. The desire for homogeneity of information may be fulfilled by a rich expressive language, however the benefits must be evaluated against the disadvantages. The richer representation will be partially redundant at some levels and there will be an increase in storage costs and access times.

Chapter 6

Formation of the BIRD Model

This chapter draws together all the information contained in the previous chapters, by documenting the formation of a new database model, BIRD. BIRD is a simple semantic model which has been incorporated into a DBMS and approaches the problems of integrity and inhomogeneity elucidated in the previous chapter. The motivation which led to the development of the Binary Data Model, BIRD is described below and this is followed by details of its structure and the operations which may be performed upon it.

6.1 Motivation

The motivation for BIRD was to incorporate a simple semantic data model within a DBMS, stressing integrity and homogeneity of information.

The incorporation of a semantic model within a DBMS has many associated benefits. The design of the database may be effected using the semantic model, with all the advantages described in the previous chapter on semantic models.

The increased representational power provided by a semantic model is expressed in the database schema and there are no problems of manual conversion between the design model and the database schema since they have been combined. The incorporation of the semantic model into a DBMS also enables it's definition to be checked automatically to ensure it is legal according to the rules of the model.

It was desired to represent all information and meta–information in a homogeneous manner in BIRD in order to avoid the disadvantages of data inhomogeneity described in the previous chapter. A simpler user interface results, since the user need only learn a single manipulation language which may be used over all the information levels in the database.

Information integrity was an important motivation due to the rudimentary facilities provided by the traditional models for integrity constraint definition and maintenance. It was desired to provide both structural integrity facilities, such as referential integrity enforcement, as well as semantic integrity facilities, such as range checking.

6.2 Conceptual Structure

A semantic network consisting of labelled binary relations connecting nodes was chosen as the basis for information representation in BIRD. The use of binary relations to represent knowledge was pioneered in the 1960's by Quillian [37, 38] in his work on semantic nets. Quillian used the semantic net as a model of human knowledge, – concepts were represented as nodes and relationships between concepts were represented as named links between the nodes. A lot of semantic net oriented research has taken place since Quillian's original work, – the representation has been used to represent information in many experimental systems [5, 29, 41, 48].

The semantic net has previously been utilised to model real world knowledge in artificial intelligence systems, however it has not been incorporated into a business oriented database system. The form of information in a business database is very different from that of more general knowledge about the world. In the former very few concepts are stored and these are associated with a large number of instantiations, – the information can be perceived as being vertical. In contrast real world knowledge can be perceived as horizontal in nature, – a large interconnected net of concepts with relatively few associated instantiations. For a comprehensive introduction to semantic nets, readers are referred to Brachman [3] which contains a detailed history and discussion of the representation.

6.2.1 Nodes

Two semantic networks are used to represent the information in BIRD, one for the schema information and the other for the application data information, – there are no explicit links between these two networks. The nodes of the semantic net represent different types of object at the different information levels in BIRD. At the schema level the nodes represent object types, such as people, cars or orders, – the object type is defined by its name and the relationships it takes part in at this level. At the application data level, nodes are used to represent instantiations of the object types defined at the schema level. The object type definition details the instantiation type, – which may be printable or abstract. Printable instantiations have a numerical or alphanumerical value, such as the instantiations of order numbers or product names respectively. Abstract instantiations have no associated printable label, but are defined via the relationships they take part in, such as the instantiations of the object types PEOPLE or HOUSE.

6.2.2 Relationships

Relationships, or labelled arcs, are used to connect either the object types or instantiations to each other and may be classified into two types, – application dependent and system defined. Application dependent relationships, such as *order number*, have no actual meaning to the DBMS, – they are declared by the user at the schema level and may be instantiated in the application data level. System defined arcs, such as *value–greater–than*, have their meaning built into the DBMS, – they are specified by the user in order to express structural and semantic integrity constraints and consequently they affect the application data level, but are not instantiated in it.

The specification of application dependent relationships must be augmented by domain information, – this is not necessary for the system defined relationships since their semantics are built into the DBMS. Of the many domains described in the previous chapter on semantic data models, two domains were chosen for their utility in referential integrity maintenance. The *duplication* domain states whether the relationship is single–valued or multi–valued and the *definition* domain states whether the relationship is mandatory or optional. The domain information must be specified with respect to both objects which take part in a relationship, consequently BIRD understands relationships in both directions between any two connected objects in the database.

The duplication domain is of crucial importance since it describes the form of the relationship between object instantiations, – whether it is one to one, one to many, many to one or many to many. The definition domain is fundamental to the maintenance of referential integrity since it enables users to describe the dependencies between the object instantiations in the system. When an insertion or deletion is effected, this information is used to guide any related insertions or deletions respectively.

6.2.3 Levels of Information

The semantic net is used to capture the database schema and application data and this information is captured in two separate nets which have no explicit connections between them.

The structure of information in BIRD can be naturally visualised in terms of stacks of cards, see figure 6.1, where the top card of every stack represents an object description and the cards in the body of each stack represent instantiations of that object description. The top cards in the stacks may be connected to each other, constituting the database schema. The cards in the body of the stack may also be connected to each other constituting the application information. The object instantiations may only possess values and take part in relationships permitted by the corresponding object type description.

BIRD was initially visualised as a structure which could represent information at an arbitrary number of levels, where information at level, n, is governed by the meta–information present at level, $n + 1$. In this manner a meta–schema level could be introduced to describe the meaning and rules of the schema in a similar manner to the EDICT model [12] discussed in the previous chapter. The rules might include construction advice such as the illegality of circular *is–a* loops, – this information could then be queried by the user when constructing the schema.

It was decided to initially design and implement BIRD using two levels of information to ensure project completion within the available time, – the n–level structure is mentioned in the later chapter entitled "Conclusions and Further Work".

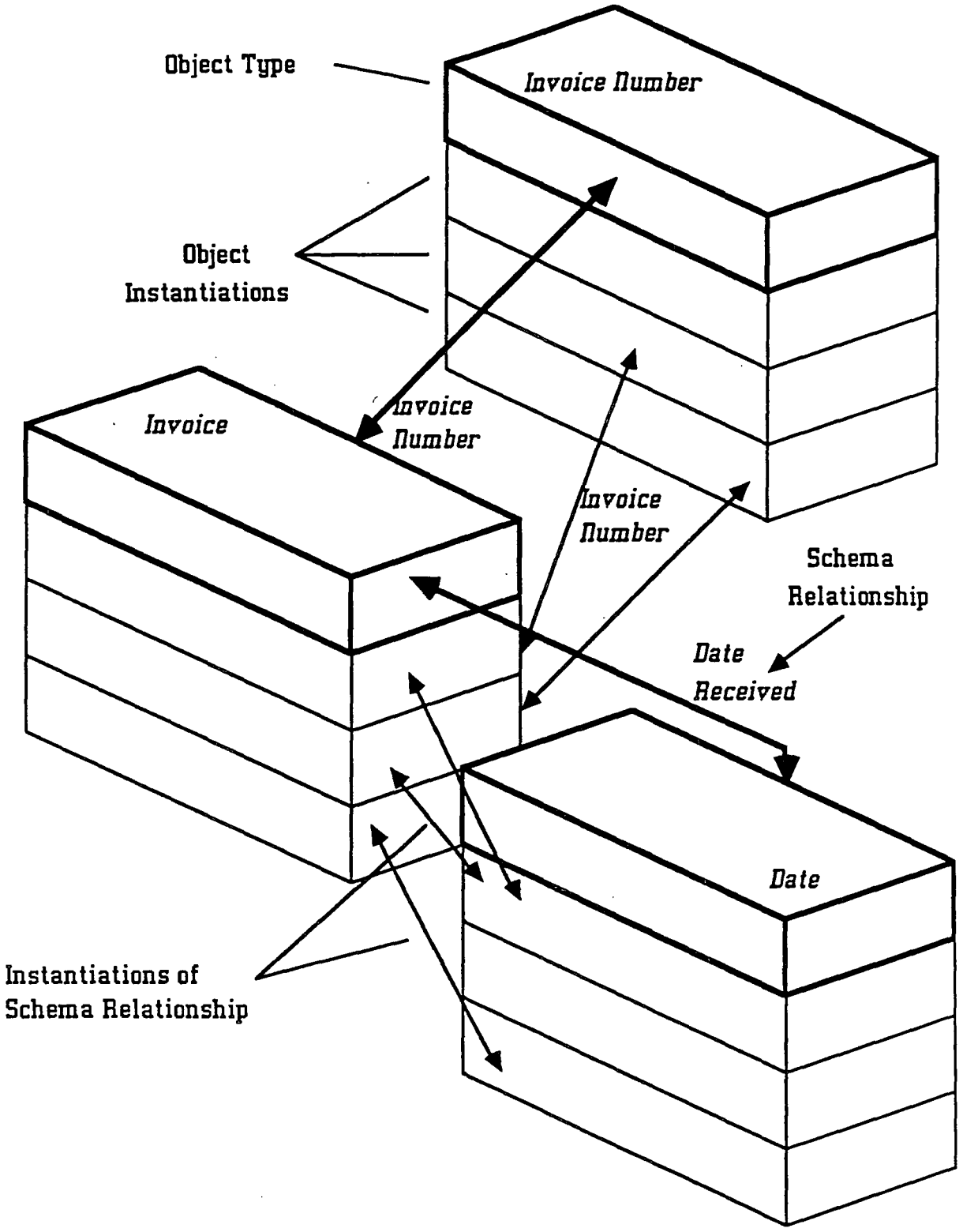


Figure 6.1: Conceptual Structure of BIRD

6.3 Manipulation of Information

It was desired to manipulate information at all levels in the system in the same manner and this was facilitated by the homogeneous representation of both application and schema information. Although information manipulation operations may be formulated in a level independent manner, the effects of these manipulations differs according to the level. The effects of manipulation of schema level information must be propagated through the application data level since the structure of all information at the lower level is governed by the information at the higher level. Manipulation of application data level information may only be carried out with reference to the schema level information, – the manipulation effects propagate through the application data level.

Manipulation of the database will be considered by examining the operations of insertion and deletion over application data and schema information.

6.3.1 Insertion

Insertion of information may take place at the application and schema levels, consequently BIRD was designed to ensure that the domain information guides the operation of all insertions to ensure the database is left in a consistent state after the operation finishes. Insertion of information at the schema level has effects which propagate down the object instances of the object types affected by the insertion.

Insertion of a schema fact incorporating an application dependent relationship affects the definition of the object instances of the two object types which take part in the fact. The effect on the instances of an object type is determined by the value of the definition domain with respect to that object type . If the definition domain of a schema fact has the value *mandatory* with respect to an object type, then that relationship must be specified for all instantiations of that object type.

If the definition domain of a schema fact has the value *optional* with respect to an object type, then the relationship need not be specified for the instantiations of that object type. If a fact must be specified for the instantiations of an object type, then the user must not be allowed to progress to the next database operation until all the appropriate instantiations have had that fact specified.

Integrity constraints may also be inserted at the schema level, – these restrict the legal range of values of object instantiations at the application data level. Subsequent to the insertion of an integrity constraint, object instantiation values should be checked and any illegal values should be changed or deleted.

Instantiation of facts at the application data level is always performed with reference to the associated facts in the structural schema level. Values of the definition domain detail which facts *must* be specified for an object instantiation and values of the duplication domain detail how many times a fact may be specified for a particular instantiation. No fact may be specified for an instantiation unless it is permitted by its associated object type description.

Insertion of an object instantiation is performed in two stages. Firstly the value of the instantiation, if it is printable, should be supplied and then all appropriate facts should be specified for the instantiation. The corresponding object type description details which facts may be specified for an object instance and which of these are mandatory. Insertion of facts or objects at the application data level may lead to associated insertions of objects and therefore facts. Thus the process of information insertion at the application data level may be viewed recursively.

6.3.2 Deletion

Domain information plays an equally important part in the deletion of information as in the insertion of information. The domain information is used to ensure that deletion of information in the database leads to the deletion of all information

which depends on it and thus the database is left in a consistent state.

Consider the deletion of an application data level relationship with respect to one of the object instantiations which it connects. If the relationship is necessary and single–valued with respect to that instantiation, then its deletion will leave the instantiation illegally defined and thus the instantiation itself must also be deleted. If the relationship is necessary and multi–valued with respect to that instantiation, then its deletion will only take place if the relationship has only been specified once with respect to it. If the relationship is optional with respect to the object instantiation, then deletion of the relationship will never cause the associated deletion of the instantiation.

Deletion of object instantiations is performed by multiple fact deletions. All the facts which the instantiation is associated with are deleted, which may lead to the deletion of other instantiations and then the instantiation itself is deleted from the database. Similarly to the insertion of application level information, the deletion of application level information may be viewed as a recursive process.

Deletion of application dependent schema level information affects the information in the application data level. If a schema level fact which incorporates an application dependent relationship is deleted, then all of the instantiations of that fact in the application data level must also be deleted. If a complete object type is deleted, then all the schema level facts in which it takes part must be deleted as well as all of its object instantiations and the facts in which they take part. Deletion of information at the schema level is not a recursive process since it does not lead to further deletions at that level and only directly associated information in the application data level is deleted.

Deletion of integrity information at the schema level does not affect the application data level, since constraints are being relaxed and thus the database is guaranteed to be consistent after the operation if it was consistent before the operation.

Chapter 7

Design of BIRD

Having described the BIRD data model in the previous chapter, this chapter details the development of that model towards an implementation. The chapter addresses the principles behind the design of BIRD and then describes the data structure used to represent the database and the procedures which operate over it. The actual implementation of BIRD is described in the following chapter.

7.1 Principles

The emphasis of this thesis is an investigation of information representation issues in databases. Consequently the design and implementation of BIRD was oriented towards simplicity rather than implementation issues. Little effort was directed towards the development of efficient database manipulation algorithms or approaching commercial implementation issues such as data persistence, security, distribution of data, crash recovery and facilities for multiple users.

7.2 Data Structure

It was desired to represent the conceptual model of BIRD as directly as possible in the design and implementation and thus the three dimensional array shown in figure 7.1 was proposed to hold the database information. Each vertical slice of the array represents one of the stacks of cards, each horizontal level in a slice represents a particular card and the different locations within these horizontal levels hold the relationships between that card and other cards in different stacks. This is a very simple and intuitive representation since object instances of all object types may be found by looking directly below the object type declaration in the array. Similarly the instantiation of any schema level fact is kept directly below it in the application data level.

The schema level is represented in two levels in the data structure, – the structural and semantic schema levels. The structural schema layer describes the structure of the application data, it comprises the application dependent relationships which may be instantiated in the application data level. The semantic schema level is made up of system defined relationships, – these specify integrity constraints over the application data and are not instantiated in the application data level.

Four coordinates, see figure 7.1, are needed in order to identify a single location in the database :–

Object Type Index Selects the particular object type which the operation will access, – in our conceptual model this corresponds to the selection of a stack of cards.

Object Instance Index Selects the particular instantiation of an object type, – this corresponds to the selection of one of the cards within a stack in the conceptual model.

Fact Type Index Selects a fact from the many facts which may be specified for

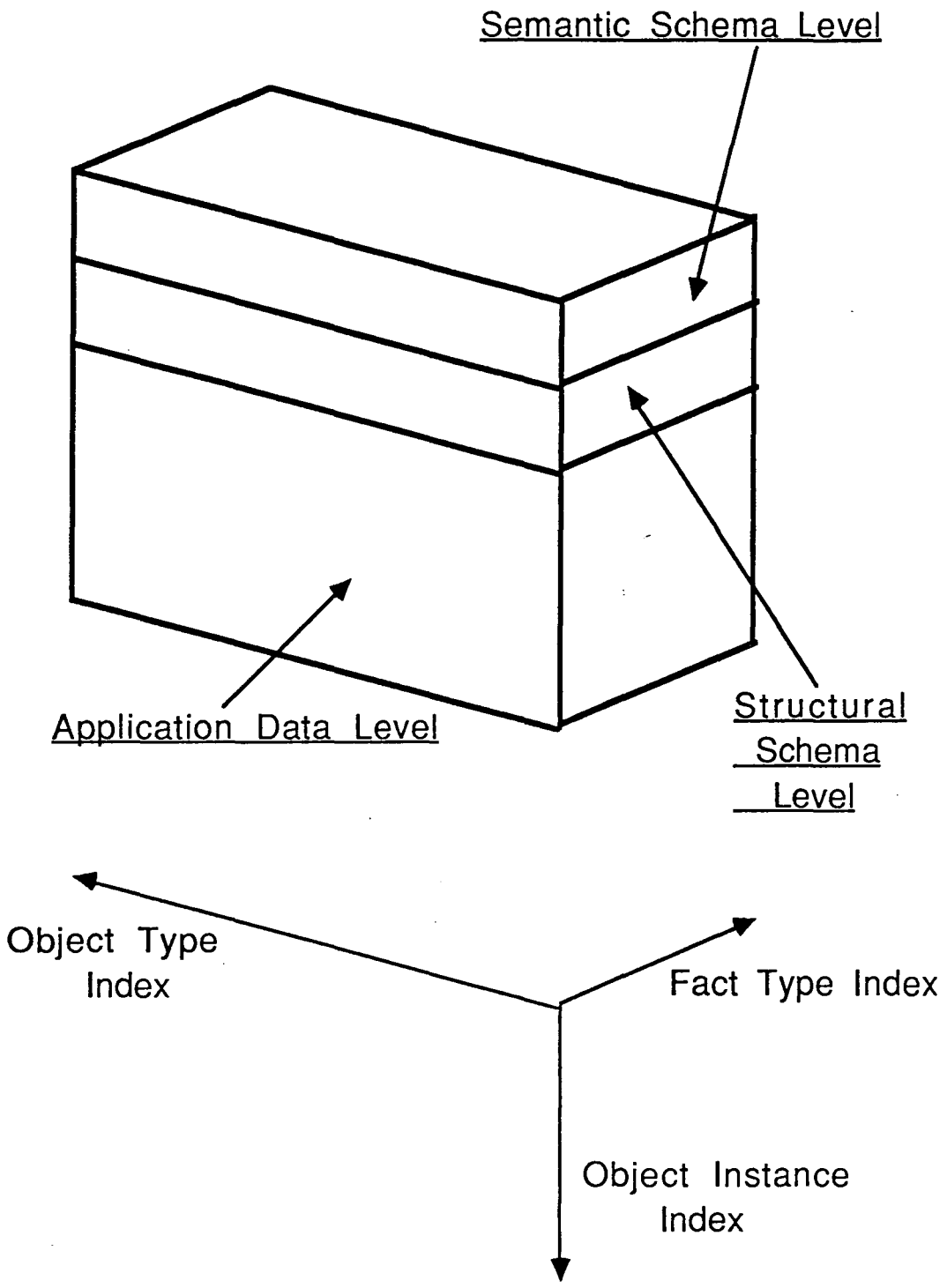


Figure 7.1: Array Structure Underlying BIRD

a card in a stack.

Fact Instance Index Selects an instance of a specified fact type, – this coordinate is necessary since a single fact at the schema level identified by a fact type index value may be instantiated more than once, according to the domain information, in the application data level. Thus the fact instance index is used to select a particular fact instantiation at the application data level.

The database array is a three dimensional array of arrays of facts, – although every location in the array could potentially hold a fact, the utilisation of the locations differs according to the conceptual level of information. At the structural and semantic schema levels, only a single fact may be stored at each location identified by a fact type index value, – the fact instance index is redundant.

Although facts are expressed in a homogeneous manner irrespective of conceptual level, different types of facts are stored at different levels. Information at the semantic schema level only comprises integrity constraint defining facts which incorporate system defined relationships. At the structural schema level, object types are declared with their associated facts and domain information. At the application data level, the information comprises the values of object instantiations and their associated facts.

7.3 Relationships

It was decided to initially define a basic set of system defined relationships with a view to extending them later if time permitted. The system defined relationships employed are described below :–

Max – Specifies the maximum value of integer instantiations of a particular object type.

Min – Specifies the minimum value of integer instantiations of a particular object type.

MaxLen – Specifies the maximum string length of alphanumeric instantiations of a particular object type.

MinLen – Specifies the minimum string length of alphanumeric instantiations of a particular object type.

There is one other system defined relationship not mentioned above, – the *is–a* relationship which is probably the most famous relationship in knowledge network research, – it has existed since the early days of semantic nets and due to its property of inheritance it has been widely studied [4]. The *is–a* relationship is utilised to provide structural information by stating that one object type in the database is a sub–object type of another object type, consequently it is an exception since it is a system defined relationship which resides at the structural schema level.

Inheritance within BIRD is rudimentary in nature, – a sub–object type inherits all the facts specified for the corresponding object type and multiple inheritance is not allowed. More sophisticated knowledge representation networks provide facilities to handle clashes arising from multiple inheritance and only allow set member properties to inherit over the link. Set member and class properties were defined in the section describing SDM in the chapter on the semantic data models. BIRD provides no facilities for the specification of class properties since their occurrence is so rare in practise, – it is very hard to think of any useful class properties except for stating the number of members in a set.

7.4 Modular Structure and Levels of Procedures

The software engineering principles of modular design and information hiding were employed in the design of BIRD, – the benefits of this approach are described in the following chapter on the implementation of BIRD. Procedures in the system were grouped into a hierarchy of five levels, see figure 7.2 where the procedures in each level may only access the procedures in the level below. Each level hides part of the underlying data structure from the level above it and incorporates more semantics into the operations than is present at the level below it.

The levels into which procedures are grouped are described below, – readers seeking a full description of all the procedures at the various levels are referred to the appendix at the back of this thesis.

In order to demonstrate how commands are built up incrementally through the levels, the insertion operation will be described at each level. In BIRD the name of each procedure is suffixed with the level at which it operates, for example the procedure to retrieve a fact *CheckFactL0* works at level zero.

7.4.1 Level Zero

Level zero is the only level at which procedures may directly access the array representing the database. Little of the semantics of the database are built into the procedures at this level, – they are passed the four dimensional coordinate of a location in the database and must perform the specified action on that location. Each procedure at this level performs one simple low level operation, such as the insertion of an object type name or setting the value of a particular domain.

If one considers the insertion of information, there are many procedures which

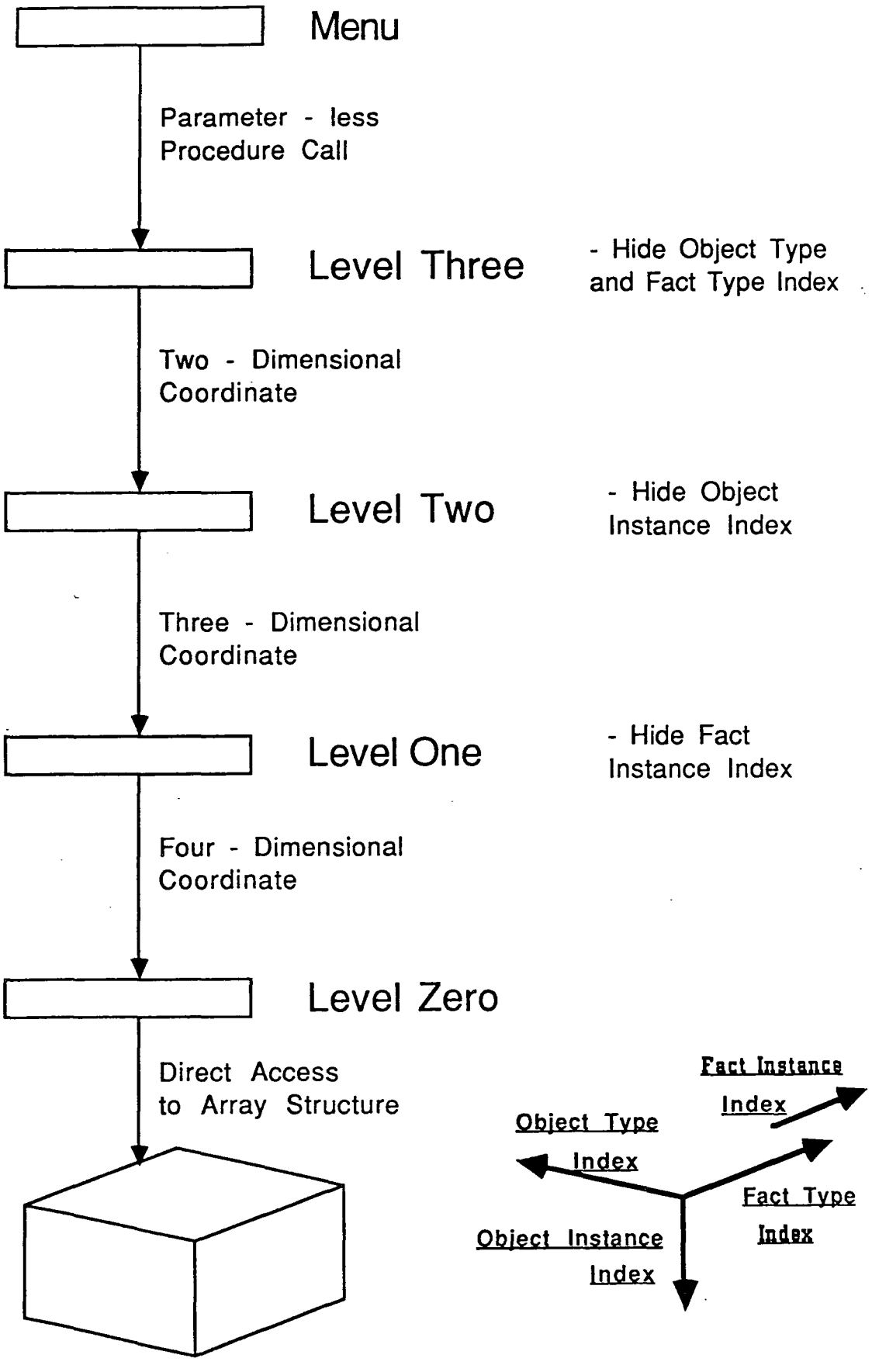


Figure 7.2: Constituent Levels of BIRD

effect this function. The *InsertFactL0* procedure operates at every conceptual level of the data structure and inserts the fact supplied at the location specified. The *InsertNameL0* and *InsertNumberL0* procedures insert the names and numerical values of object types and instantiations. There are also the procedures *MandatoryFlagL0* and *DuplicationFlagL0* which set the relevant domains to the value supplied.

7.4.2 Level One

Level one procedures manipulate the database data via the procedures provided at level zero. Level one builds on the procedures provided at level zero and hides the fact instance coordinate from level two. Level one procedures which insert facts in the application data level must call level zero procedures to determine the fact instance cooordinate value of the first free location for that fact. Level one procedures which delete facts in the application data level are supplied a three dimensional coordinate and the particular fact, – they must use the level zero procedures to search through the fact instances at that three dimensional coordinate to find the fact instance coordinate value of the fact supplied before it may be deleted. Procedures which operate at the semantic and structural schema levels implicitly hide the fact instance coordinate from the level above since only one fact may be stored in each location identified by the fact type index at this level.

Considering insertion of information at level one, the *InsertFactInstL1* procedure operates over the application data level and inserts the fact provided. The procedure uses the level zero procedures to inspect consecutive fact instance locations until it finds one which is free, – the fact is inserted at this location. The *InsertFactTypeL1* procedure operates at the structural and semantic schema levels. The procedure is passed a three dimensional coordinate and a fact, – it checks the location specified is free and then inserts the fact. The *InsertObjectInstL1* procedure instantiates an object type, – this is effected by setting the instance's value

at the location specified to the name or number provided. The *InsertObjectTypeL1* increments the value of the object type index until it finds at free location, – it then inserts the object type by setting the location to the value supplied. The insertion of domain information is split up into four procedures at this level, – each procedure either sets or clears one of the domain values for a specified fact location in the structural schema level.

7.4.3 Level Two

Level two hides the object instance coordinate from the level above, – instead of passing a coordinate containing an object instance coordinate value, the procedures at level three must specify the object type and the actual value of its object instance. The level two procedures use this object instance value to determine the particular object instance coordinate. More semantics are built into this level, – insertion procedures check that the information has not already been inserted, – in this way duplicate data facts, object types or object instances cannot exist in the database. All procedures at this level operate on pairs of facts. For instance if a fact is to be inserted in the application data level then it is inserted into the fact lists of both the object instance arguments of the fact.

Let us consider the insertion of information at level two. The *InsertObjectInstL2* procedure inserts object instance values in the application data level at the first free position below the specified object type, having first checked that there are no object instances with identical values for the object type specified. The *InsertObjectTypeL2* procedure operates over the structural schema level, it firstly checks for identical object types and then inserts the object type specified. The *InsertDataFactL2* procedure operates over the application data level, – it inserts the data fact in the fact list of both object instances which take part in the fact, having firstly checked that the fact does not already exist. The *InsertSchemaFactL2* procedure operates at the structural schema level, – it inserts the fact and domain

information supplied in the fact lists of both of the object types involved, having first checked for fact duplication. The *InsertConsistencyFactL2* procedure operates at the semantic schema level and inserts the fact provided in the fact lists of both of the object types involved. As with all the other procedures at this level, a check is made to ensure the information has not already been inserted before the operation takes place.

7.4.4 Level Three

Procedures at level three form the dividing line between the database proper and the user interface which is built on top. The procedures are called from the level above without parameters and thus all the remaining semantics of the operations must be built into this level. Since the procedures are called without parameters they perform all input and output necessary to gain the information necessary for execution.

Structural and semantic integrity of information is stressed at this layer, – object instance values are checked for their semantic legality and structural integrity is checked after insertion or deletion operations. Insertion or deletion of information at the schema level affects the corresponding object instances in the application data level, – the procedures at this level take this into account. The behaviour of the *is–a* relationship is also taken into account at this layer, effects of insertions or deletions of schema level facts must be propagated to any sub–object types of the object types in the facts. Further information on integrity maintenance and the incorporation of the *is–a* relationship may be found in the following chapter describing the implementation and operation of BIRD.

The *InsertObjectTypeL3* procedure requests all the necessary information from the user and then inserts the object type into the database. This procedure is an exception since it's execution cannot compromise the integrity of the database,

– an object type has been inserted but it has no associated instantiations whose definition may be inconsistent.

The *InsertConsistencyFactL3* procedure requests from the user the details of a semantic integrity fact, this is checked to ensure it is a legal fact before it is inserted into the database. One of the arguments of an integrity fact is a constant, – the RemedyObjectTypeDefinitionViaInsertionL3 procedure is called to propagate the effect of the insertion to the object instances of the single object type named in the fact. The operation of the RemedyObjectTypeDefinitionViaInsertionL3 procedure is described below.

The *InsertSchemaFactL3* procedure operates in a similar way to *InsertConsistencyFactL3*. Firstly the details of the structural schema fact including the domain information is requested from the user and it is then inserted into the database. The fact is propagated to all sub–object types of both the object types which take part in the fact and then the procedure *RemedyObjectTypeDefinitionViaInsertionL3* is called for every object type which has been affected by the insertion. BIRD tests for the special case where the structural schema fact contains the *is–a* relationship. In this case the structural schema fact itself is not propagated down the *is–a* hierarchy, instead all structural and semantic schema facts associated with the super–object type are propagated to all the sub–object types.

The *InsertObjectInstL3* procedure requests from the user the object type to be instantiated and the object instance value. The object instance value is checked against the semantic schema information before the object instance is inserted and the procedure *RemedyObjectInstDefinitionViaInsertionL3* is then called. The operation of the *RemedyObjectInstDefinitionViaInsertionL3* procedure is described below.

The *InsertDataFactL3* procedure establishes from the user the particular structural schema fact which is to be instantiated. The user must specify the object instances which form the subject and object of the application data level fact, if

the domain information allows the specification of the fact for these arguments then it is inserted. It is possible that the arguments of the data fact are new object instances and thus *RemedyObjectInstDefinitionViaInsertionL3* is called for each argument.

The procedure *RemedyObjectInstDefinitionViaInsertionL3* inspects the application data facts which have been specified for an object instance and compares them against the structural schema information for the corresponding object type. If it is found that any structural schema facts which are necessary have not been instantiated for an object instance then the user is invited to instantiate a structural schema facts. The user must continue instantiating structural schema facts until the definition of the object instance is correct with respect to the schema information. If any new object instances have been created in this process then *RemedyObjectInstViaInsertionL3* checks their definition as well and thus it operates recursively. *RemedyObjectInstViaInsertionL3* also checks the values of the object instances against any appropriate integrity constraints. If an object instance value contravenes an integrity constraint the user is invited to supply a new value.

The procedure *RemedyObjectTypeViaInsertion* checks the definitions of all the object instances associated with an object type. *RemedyObjectTypeDefinitionViaInsertion* steps through all the object instances of the specified object type and passes them to *RemedyObjectInstDefinitionViaInsertion.*

7.4.5 Menu

The menu level forms the highest level of procedures in BIRD and provides the interface of the database with the user, it is described more fully in the following chapter on the implementation of BIRD.

BIRD is menu driven, – two levels of menu are presented to the user. The first level of menu allows the user to perform various housekeeping functions, such as

storing or retrieving permanent copies of a database. The second level provides the functions which actually operate on the database. The user must choose a function, – *insert, delete* or *query*, a level, – *semantic schema, structural schema* or *application data* and a granularity of operation, – *object* or *fact.* Having selected the operation from the menu, the operation is executed by calling the appropriate level three procedure. The level three procedures request and check the information they need to execute and thus the user is lead through the execution of commands in BIRD.

Chapter 8

Implementation of BIRD

The previous chapters have described the motivation which led to the formation and subsequent design of the BIRD model. This chapter describes the implementation of BIRD using Modula–2. The system was implemented in levels of modules, – each level is discussed together with a description of relevant implementation decisions made whilst implementing that level.

Modula–2 was chosen as the implementation language due to its modular software development facilities which enabled efficient software engineering of the project, as described later in this chapter.

As mentioned in the previous chapter BIRD was proposed as an experimental prototype DBMS, addressing representational rather than implementation issues. Consequently, implementation decisions which were made during the development of the system stressed elegance and simplicity rather than optimisation of resources.

8.1 Data Structure

The database data structure is defined as a three dimensional array of records. The record stored in a single location in the array may take the form of either of three variants. The record variant which is chosen is determined by the conceptual level of information which the cell represents, see figure 8.1, – either semantic schema, structural schema or application information. A semantic schema variant consists only of a fact, in contrast a structural schema variant consists of a fact with domain information plus an optional description of the object type that cell represents and the type of its instantiations. An application data variant consists of an array of data facts plus an optional description of the object instance that cell represents.

Within the planes identified by the object type index and fact type index all cells are of the same variant record. However the utilisation of the fields of these variants changes according to the value of the fact type index. Only cells selected by the first value of the fact type index may store a name or value as well as a fact. Thus for a particular object type index value, the record identified by the first value of the fact type index in the structural schema layer will hold the name of the object type. By keeping the object type and fact type index values constant whilst varying the object instance index value, the values of the instances of that object type may be inspected. In the structural and semantic schema layers, facts associated with the object types may be selected by individual fact type index values. In the application data layer, the fact type index is used to select a particular array of data facts which is further indexed by the fact instance index.

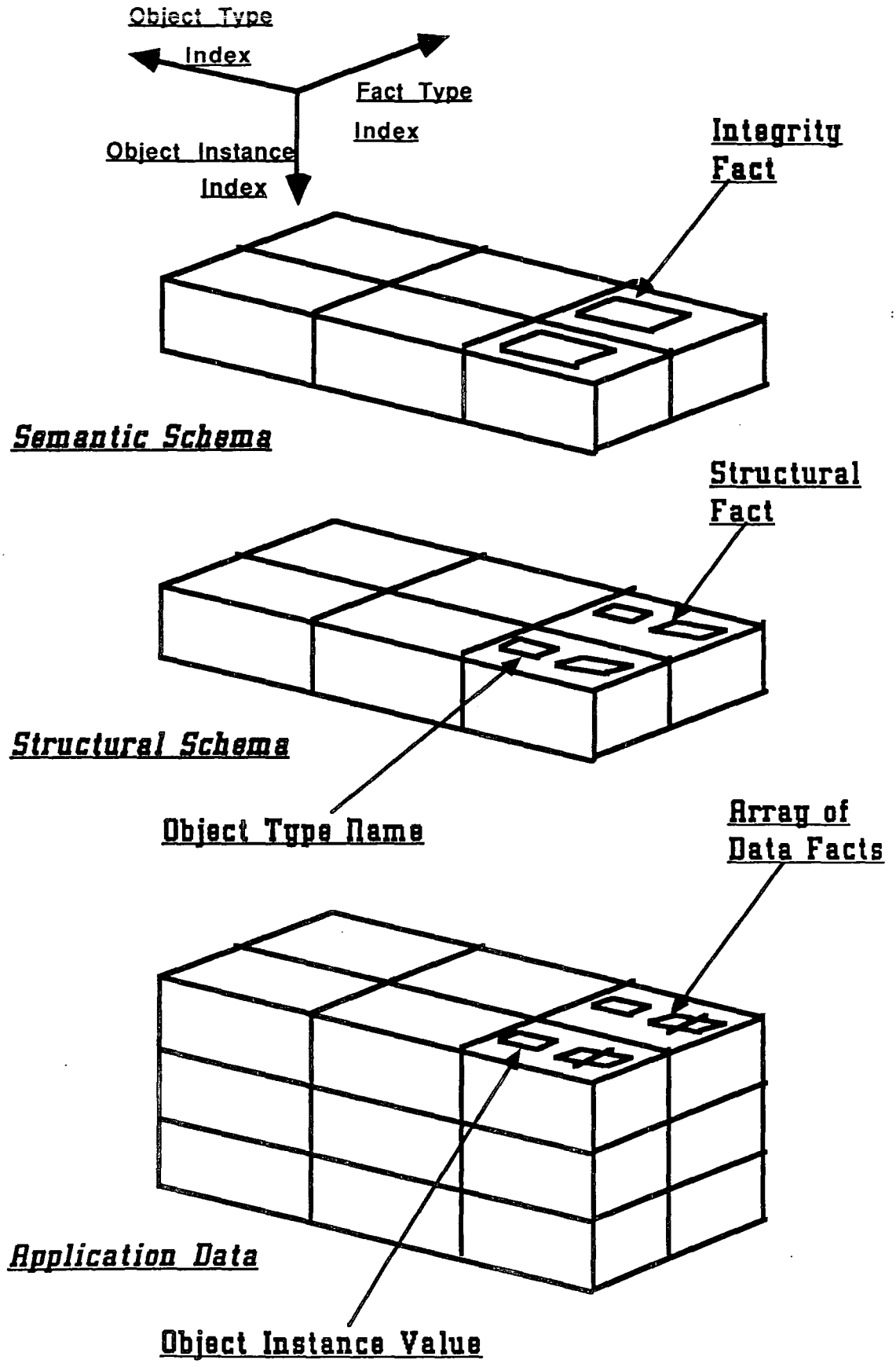


Figure 8.1: BIRD Database Array Cell Structure at Different Conceptual Levels

8.2 Modular Structure

The BIRD system was constructed using the modular facilities provided by the Modula–2 programming language. A module is composed of data structures and procedures which are associated with an external interface which defines what data structures and procedures other modules may access. The modular structure may be used to enforce data hiding, – in this manner a data structure is declared locally to a module and may only be manipulated via the procedures in that module. Thus procedures in other modules may only access the data structure indirectly via the authorised procedures. Data hiding ensures that erroneous system states resulting from direct accesses to a data structure by procedures distributed throughout a system may not occur. The modular structure also facilitates independent compilation, – after a change to a module only *that* module and modules which are dependent on it need to be recompiled as opposed to the whole system.

The BIRD system was designed in hierarchical layers of procedures, where a particular layer may be represented in multiple modules according to its size. Each level of procedures in the hierarchy implemented the same three basic database operations, – retrieval, insertion and deletion, however the sophistication of the operations was built up through the layers as described in the previous chapter on the design of BIRD.

The completion of each level in the hierarchy provided a convenient and logical stage at which to test the system. This eased the task of testing and debugging since it was effected in small manageable stages, as well as promoting the efficient software engineering of the system.

8.3 Implementation of Levels

This section describes the implementation of each level of procedures which constitute the BIRD system, including a description of important decisions made at each stage. All implementation decisions were made according to the principles detailed at the beginning of this chapter.

8.3.1 Database

A module called 'database' was created to contain definitions which would be needed at multiple levels of the system, such as the definitions of names and co-ordinates. The modules which constitute the levels of the system may all import definitions from this single module. The 'database' module was a cosmetic exercise which ensured that each level of modules implementing the database functions only imported definitions from the level below. The definitions contained in the module 'database' could equally have been placed at level zero and imported into multiple levels.

The 'database' module also included a single procedure called 'halt' which was useful to all levels of procedures in the system. The procedure 'halt' is called by any procedure which encounters an unexpected or erroneous system state, such as inadequate space in the database array. The error handling procedure displays the error message passed to it and then induces a system error in order to abnormally terminate the program. In this manner the tracing facilities of Modula–2 are invoked and a call graph of the procedure calls leading to the abnormal termination is displayed, thus aiding the debugging process.

8.3.2 Level Zero

Level zero is the only level at which procedures may directly access the array containing the database. Part of the declaration of the database array is shown in figure 8.2, – this declaration is local to level zero and thus invisible to all other modules.

The operations performed at level zero are all very fundamental in nature, – the procedures are passed a four dimensional coordinate exactly identifying an array location and information is inserted, deleted or retrieved at this location.

8.3.3 Level One

Level one procedures build on the low level procedures at level zero in order to include more semantics. It was decided at this level that information in the array would not be restricted to occupying successive locations in any one dimension. For example if a fact is deleted, then remaining facts will not be shuffled up one location to fill the gap, thus information may be distributed in patches throughout the array. This decision simplifies the operations of insertion and deletion, – insertion is performed at the first free appropriate location and deletion is effected by simply deleting the information. This method introduces an overhead for queries, since encountering an empty location does not imply that subsequent locations inspected will be empty.

Similarly in order to simplify programming, object types were not stored in successive locations or in any particular order in the array. Alternatives include the use of hashing routines to locate object types or simply ordering them alphabetically.

Level one procedures introduce some simple semantics into the database manipulation operations. The value of the fact instance index is hidden from the level

```
TYPE
    DataFactType = ARRAY FactInstIndexType OF DataFactInstType ;

    DBCellType   = RECORD
                       CASE CellIdentity  : LevelType OF
                         SemanticSchema :
                           ConsisFact     : SchemaConsisFactInstType|
                         StructuralSchema:
                           ObjectTypeDesc : ObjectTypeDescType ;
                           SchemaFact     : SchemaConsisFactInstType;
                           Domain         : DomainInformationType |
                         Data:
                           ObjectInstDesc : ObjectInstDescType ;
                           DataFacts      : DataFactType ;
                       END ;
                   END ;

    DatabaseType = ARRAY ObjectTypeIndexType, FactTypeIndexType,
                       ObjectInstIndexType OF DatabaseCellType ;

VAR
    database     : DatabaseType ;
```

Figure 8.2: Part of the Modula-2 Definition of the BIRD Database Array

above and thus insertion procedures must inspect the locations of the application data fact arrays to determine where to insert a fact and deletion procedures must search the locations of the specified application data array to find the fact to be deleted.

8.3.4 Level Two

At level two the additional complexity of the operations necessitated the creation of multiple modules to handle the large volume of code. Modules were defined for the separate database functions of insertion, deletion and retrieval as well as modules for user input, output and database integrity checking.

In BIRD since all facts are relationships between two entities, the facts have to be stored twice, once in the fact lists of both entities concerned. At level two the database manipulation procedures take this into account by operating on pairs of facts. Procedures which insert information at level two also ensure that the information does not already exist before inserting it.

The modules which constitute level two each perform a cohesive set of functions and have no need to reference procedures in other modules at the same level. Consequently updating the code of any module at level two does not necessitate recompilation of any other level two module.

An output module was written to provide basic procedures for outputting character strings, numbers and facts. An input module was written providing procedures for eliciting domain information, object type names, object instance values and facts from a user. An integrity module was also written which provided procedures for testing the integrity of object instances with respect to the schema level information and testing if insertion of specified information would compromise the integrity of the database.

8.3.5 Level Three

Level three constitutes the interface between the database manipulation operations and the user interface above. All remaining semantics were built into the procedures at level three, – these procedures were separated into three modules by declaring one each for insertion, deletion and retrieval of information. As with level two, the procedures in the level three modules do not reference each other.

All procedures at level three perform their own input and output where necessary. The insertion and deletion procedures ensure that the integrity of the information in the database is intact after the operation has finished and take into account the semantics of the *is–a* relationship. For instance, if an instance of an order is inserted into the database then all information necessary to the definition of that order instance will be requested from the user before the insertion operation terminates, – if this leads to the insertion of other object instances then these will also be correctly defined. Extensive examples of the operation of the integrity maintenance system are given in the next chapter on operation and evaluation as well as a full description of the incorporation of the *is–a* relationship.

Since the level three procedures include all remaining semantics, they may be called from higher levels without parameters. Any procedure at level three will ensure it elicits all necessary information for its execution and that the database integrity is intact before terminating.

8.3.6 Menu

The menu level formed the interface between the database proper and the end user. Since the thrust of the BIRD project is towards representation issues, the construction of a complex user interface was not deemed appropriate. The brief of the user interface was to allow a user to insert and delete information with

ease, coupled with a basic information query facility. Owing to the emphasis on homogeneity it was desired to enable the user to manipulate information at different conceptual levels in the same manner.

A menu driven interface was implemented which enables the user to build up simple commands by the selection of options from three lists. The first list expresses the function, – *insert, remove* or *query*; the second list expresses the conceptual level of the information, – semantic schema (*consistency*), structural schema (*schema*) or application data (*data*) and the last list expresses the granularity of the operation, *object* or *fact*. Selection of "object" granularity is assumed to mean object type at the schema levels and object instance at the application data level. Thus examples of selections might be "insert structural schema level fact" or "delete application data level fact", – the former is shown in figure 8.3. To execute this selection the user presses the "RETURN" key with a blank command line. To change the selection the user enters the first letter of the appropriate list entries, – thus to select the deletion of an application data fact the letters, 'rdf' would be entered. Having pressed the "RETURN" key, the menu would change to reflect the selection, pressing the "RETURN" key again would execute the command.

The BIRD system is "active" by nature, – once the desired command has been selected, the relevant parameters will be requested and then all necessary measures will be taken to ensure that database integrity is maintained. BIRD provides the user with the choice to insert or delete any of the information at any level in the database without restriction and then ensures that any related insertions or deletions, respectively, are performed.

During experimentation with the system it was deemed desirable to provide a facility for storing the database permanently, – since at that time the database was lost as soon as the program was terminated. Storage of the database could have been effected in a number of ways of varying complexity, such as retrieving all the facts and depositing them in a file or keeping all the facts permanently in a file to replace the use of the internal array. The simplest option was chosen, – to directly

```
-------------------------------------------------------
|    OPERATION     |      LEVEL       |   GRANULARITY  |
-------------------------------------------------------
|      QUERY       |   CONSISTENCY    |      OBJECT    |
| #    INSERT      | #    SCHEMA      | #    FACT      |
|      REMOVE      |      DATA        |                |
-------------------------------------------------------

To change the command executed, enter the first letter of the
selection you wish to make or enter 'u' to exit to the upper
menu level.  Press RETURN to execute the command.
```

Figure 8.3: Database Manipulation Menu in BIRD

```
--------------------------------
|          OPERATION           |
--------------------------------
|    CHANGE SNAPSHOT LIBRARY   |
|    DELETE DATABASE SNAPSHOT  |
|    LIST DATABASE SNAPSHOTS   |
| #  RESTORE DATABASE SNAPSHOT |
|    SAVE DATABASE             |
|    NEXT MENU DOWN            |
|    QUIT PROGRAM              |
--------------------------------

To change the command executed, enter the first letter of the
selection you wish to make.  Press RETURN to execute the command.
```

Figure 8.4: High Level Menu in BIRD

write the whole array into a file using the file input/output facilities provided by the Modula–2 library procedures. This concept was named the database 'snapshot' since facilities were provided to freeze and retrieve complete pictures of the database, however these snapshots could not be mixed in any way since retrieval of a snapshot completely overwrites the current array contents.

To write the array into a file necessitated a procedure at level zero in order to have access to the array. Consequently the menu level directly accesses procedures at level zero in order to effect database snapshot storage and retrieval. Although the menu level should only reference procedures at level three, there was no scope for building up the complexity of the snapshot commands through the levels, since they were complete at level zero and thus they were accessed directly.

The introduction of snapshots necessitated the construction of a higher level menu, shown in figure 8.4, consisting of snapshot manipulation options plus options to descend to the database manipulation menu or to quit to the operating system.

8.4 Conclusion

The BIRD system was successfully implemented in ten thousand lines of source code which compiled into a one hundred and forty five thousand byte executable file. During experimentation with the system the dimensions chosen for the array were as follows:–

- Object Type Dimension : 40
- Object Instance Dimension : 20
- Fact Type Dimension : 20
- Fact Instance Dimension : 10

When this array was stored as a snapshot it occupied 2.5 megabytes of memory. Commonly, most of the locations of the array were not utilised, – the use of the UNIX "compress" facility yielded a storage reduction by about ninety percent.

The structure of the system is summarised in a call graph, figure 8.5, which displays the modules comprising the system and highlights the hierarchical nature of the system effectively. Note the complete lack of interaction between modules at any particular level. A table summarising information on the size of each level of procedures is shown in figure 8.6

The ability to develop and test the system in stages greatly facilitated its implementation. The utility of comprehensive testing of the procedures at one level before implementation was started on the next level was proven since errors originating in procedures buried in lower levels were always hard to find. One error involving a reversed array subscript in a procedure contained in a module three levels below the one currently being implemented took nearly a day to find.

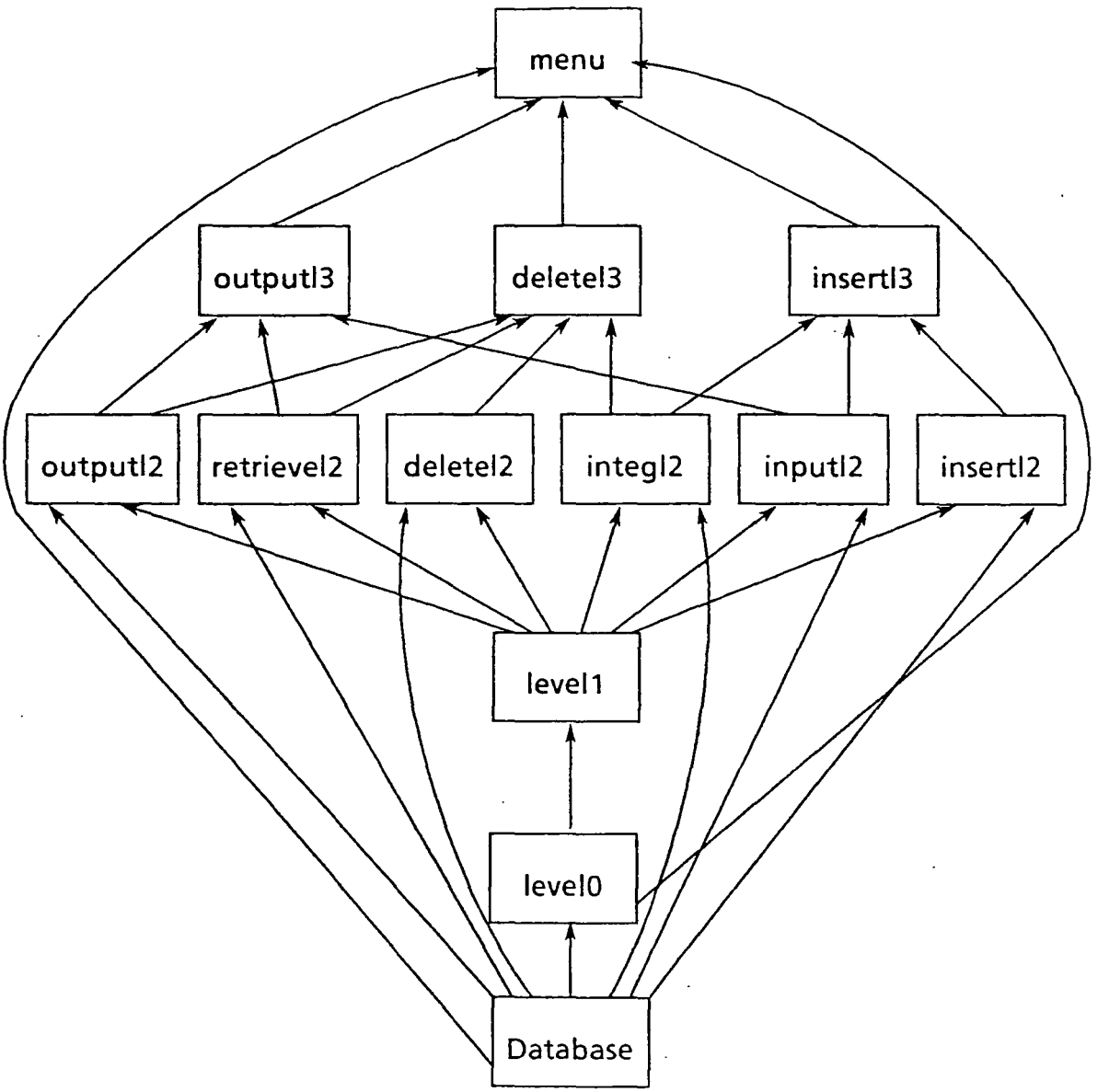
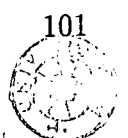


Figure 8.5: Structure of BIRD Showing Inter–Module Interaction

Level Name	Number of Modules	Size/ kBytes	Size/ lines
Database	1	6	135
Level Zero	1	30	788
Level One	1	55	1434
Level Two	6	298	6243
Level Three	3	96	2358
Menu	1	26	702

Figure 8.6: Summary of BIRD Implementation Details



Chapter 9

Operation and Evaluation of BIRD

This chapter details the operation of the BIRD database by describing the execution of database manipulation operations. Attention is focussed on the internal processes which ensure that the integrity of the database is intact before each operation terminates. The interaction between the *is–a* relationship and the database manipulation operations is discussed in a separate section. A section is present at the end of the chapter which contrasts the BIRD approach with a record oriented database example.

The user selections necessary to perform specified operations are briefly described where appropriate, however this chapter is not intended to act as a "user–manual" for BIRD. The user menus used for specifying the database manipulation operations may be seen in figure 8.3 and figure 8.4 in the previous chapter. The hypothetical factory parts ordering database mentioned in the introduction of this thesis is used as an example throughout this chapter and is extended for the purposes of exposition. The same factory parts ordering database was implemented

using the INGRES 5.0 relational database and salient differences between the two approaches are discussed.

The methods which BIRD employs to maintain database integrity may only be effectively demonstrated if there is already information in the database which will be affected by the manipulations. Consequently it is assumed that the schema shown in figure 9.1 has already been entered into BIRD as well as associated application data. Figure 9.1 details the object types present in the database, such as ORDER and ORDER ITEM as well as describing the relationships between them and the types of instantiations they support. For example the ORDER object type has abstract instantiations and is related to the ORDER ITEM object type by the *order number* relationship which is mandatory and single–valued with respect to ORDER and mandatory and multi–valued with respect to ORDER ITEM.

9.1 Information Insertion

This section describes the insertion of information at the semantic schema, structural schema and application data levels. Attention is directed at the internal processes which ensure all information related to the insertion is supplied in order to maintain database integrity.

Let us consider the situation where an order has been received to be entered into the hypothetical factory parts ordering database. The user would be free to start describing the order to BIRD by instantiating any of the relevant object types, such as ORDER NUMBER, ORDER or ORDER ITEM. To instantiate an object type the *insert, application* and *object* options would be selected from the user menu, the user would then be prompted for the name of the object type and for the purposes of exposition let us assume that the ORDER ITEM object type was chosen to be instantiated. An ORDER ITEM instantiation would be created, – a value need not be supplied for the instantiation since it is an abstract instantiation.

Entity ORDER

Instantiations Abstract

Relationships

Name Order Number
Definition Mandatory
Duplication Single–Valued
Argument SERIAL #

Name Address
Definition Mandatory
Duplication Single–Valued
Argument ADDRESS

Name Order Item
Definition Mandatory
Duplication Multi–Valued
Argument ORDER ITEM

Entity ORDER ITEM

Instantiations Abstract

Relationships

Name Order Item
Definition Mandatory
Duplication Multi–Valued
Argument ORDER

Name Quantity
Definition Mandatory
Duplication Single–Valued
Argument QUANTITY

Name Part Number
Definition Mandatory
Duplication Single–Valued
Argument PART NUMBER

Entity QUANTITY

Instantiations Integer

Relationships

Name Quantity
Definition Mandatory
Duplication Multi–Valued
Argument ORDER ITEM

Constraints

Name Greater Than
Argument 0

Entity SERIAL #

Instantiations Integer

Relationships

Name Order Number
Definition Mandatory
Duplication Single–Valued
Argument ORDER

Constraints

Name Less Than
Argument 20,000

Figure 9.1: Factory Parts Ordering Database Schema

The structural schema level information would then be consulted by BIRD to ascertain what further information should be supplied for the new instantiation. The user would be prompted to choose to instantiate one of the *order item, quantity* or *order item* relationships. If the *order item* relationship was chosen first then the user would specify whether the ORDER ITEM instantiation was to be associated with a new or existing ORDER instantiation. In this case a new ORDER instantiation would be selected, since information is being entered on a new order and this would lead to the specification of address and order number information associated with the order instantiation. The insertion would not be finished once the ORDER instantiation had been defined since the *quantity* and *part number* relationships from the ORDER ITEM instantiation would have yet to be instantiated since the definition domain states they are mandatory. Once these relationships and the associated instantiations had been correctly defined the insertion operation would terminate.

When instantiating a relationship the domain information of that relationship with respect to the destination object instantiation plays a crucial role. In the above example, the *order item* relationship instantiation could be connected to any existing ORDER instantiation or a new ORDER instantiation, since this relationship is multi–valued with respect to the ORDER object type which indicates that any ORDER instantiation may be connected to many ORDER ITEM instantiation. If a relationship was single–valued with respect to an object type, then it might only be connected to an existing instantiation for which the relationship had not already been specified or a new instantiation. For example if the *married to* relationship was declared single–valued with respect to the WOMEN object type, then an instantiation of the relationship would have to be connected to an existing WOMEN instantiation for whom it had not already been specified or a new WOMEN instantiation could be entered into the database. If a relationship was single–valued and mandatory with respect to an object type then it must already have been specified for all existing instantiations of that object type. For instance the relationship *date of birth* might be declared single–valued and mandatory with

respect to all instantiations of the HUMAN object type, – thus if a new date of birth was entered into the database it would have to be associated with a new instantiation of the HUMAN object type.

Let us consider the insertion of information at the structural schema level. The ability to easily change a database schema is an important feature of a DBMS since databases model fluid real world environments. BIRD enables a user to insert new schema information easily and brings the application data up to date using the newly specified domain information as a guide, – this is demonstrated below.

For the purposes of exposition let us assume that a corporate decision was made enforcing the storage of the date of receipt of every order in the hypothetical parts database. In this situation a new object type, DATE, would be defined with character string instantiations. A relationship called *date of receipt* could then be defined between the ORDER and DATE object types. This relationship would be single–valued and mandatory with respect to ORDER and multi–valued and mandatory with respect to DATE. These domain values would be appropriate since all orders *must* be associated with a *single* date and every date *must* be associated with at least one order but may be associated with more than one.

Having inserted the new structural schema fact, BIRD would inspect the instantiations of the fact's object type arguments to determine whether any inconsistency had been introduced into the database, taking into account the relevant domain information. There would not be any instantiations of the DATE object type in the database, since the object type has just been declared and thus there cannot be any inconsistency between the definition of the object type and its instantiations. However the definitions of the existing ORDER instantiations would all be inadequate since each must be associated with a date of receipt. Each instantiation would be taken in turn and associated with a date of receipt, – the insertion operation would only terminate once every ORDER instantiation had been correctly defined.

It is appreciated that when changing a schema, the user may not possess the relevant information to bring the existing application data in the database up to date. For example, the user may not have the dates of receipt of all past orders in the hypothetical factory parts ordering database.

This problem may be approached in two ways. Firstly, if the domain information of new relationships could always be specified as optional, thus they would not have to be supplied for existing instantiations. This is an unsatisfactory solution since relationships which are mandatory will not be correctly defined and may their specification may be avoided in future instantiations. A more satisfactory solution would involve development of specific facilities in BIRD to handle schema updates. In this way application data would always be associated with the schema which was active when it was entered. Explicit relationships could then be used to link together the versions of the schema and facilities built into the DBMS to interpret the situation. Narayanaswamy and Bapa Roa, [30], have carried out related work in modelling schema evolution in engineering environments.

Let us consider the insertion of information at the semantic schema level, this may be selected by the *insert, consistency* and *fact* options on the user menu. Information at this level expresses restrictions on the values of the object instances in the application data level. The values of all object type instantiations are always checked against the semantic schema information before insertion into the database to ensure erroneous values may not be introduced into the database. In the factory parts database example, figure 9.1 shows that a fact has already been declared at this level which limits the maximum value of serial numbers to twenty thousand. Let us assume it was desired to enter the fact that serial numbers must be greater than zero, – having selected the appropriate options on the user menu, the object type which is the subject of the fact, SERIAL #, would be specified, then the appropriate system defined relationship, *Min* and lastly the minimum integer value.

Having supplied the new semantic schema level fact, BIRD would check existing instantiations of the SERIAL # object type to ensure there is no inconsistency

between their values and the new fact. If any existing values of instantiations contravened the new semantic schema information then the user would be requested to enter a legal replacement value.

9.2 Information Deletion

Deletion of information in BIRD is performed with the same emphasis on consistency as insertion of information. Once the deletion operation has been given, the specified information is deleted plus any information which relies upon it for its definition. Since BIRD is only a prototype, deletion of associated information is carried out without consultation with the user and thus a user may unwittingly lose more information than expected.

Let us first consider the deletion of information at the application data level. Considering the hypothetical factory parts ordering database it might be desired to remove an erroneous order item from a particular order. The user should specify the removal of the relationship between the appropriate ORDER and ORDER ITEM instances, – one should not request the deletion of the actual ORDER ITEM instance since it may be associated with other ORDER instances.

Having selected the *delete, application* and *fact* options from the user menu, the relevant fact would be specified and BIRD would commence the deletion. Having deleted the fact, BIRD would check the definitions of the two object instantiations the fact linked. The domain information details that all order instantiations must be associated with at least one order item, if it is assumed that this order instantiation was associated with another order item instantiation then it would not be deleted. Similarly since the *order item* relationship is mandatory and multi–valued with respect to the ORDER ITEM object type, all instantiations of this object type must be associated with at least one order. For the purposes of exposition, let us assume that the relevant ORDER ITEM instantiation was not connected to

any other ORDER instantiations, – in which case it must be deleted.

In order to delete an object type instantiation, BIRD deletes all facts in which that instantiation takes part before deleting the instantiation itself. Thus in this example the deletion of the ORDER ITEM instantiation leads to the deletion of the associated *quantity* and *part number* relationships which may then lead to the associated deletion of the QUANTITY and PART NUMBER instantiations. The related QUANTITY and PART NUMBER instantiations would only be deleted if they were not connected to other ORDER ITEM instantiations. Deletion of application information in BIRD can be programmed recursively since deletion of facts may lead to deletion of instantiations which may lead to the deletion of more facts. A single deletion of a fact or instantiation in BIRD may cause a wave of deletions to spread through the application data, – this wave only subsides once the database has reached a consistent state.

Deletion of information at the structural schema level has entirely different propagational effects. Instead of deletions spreading horizontally through the level, they spread vertically into the application data. Let us consider the situation where it is no longer desired to associate orders with order numbers. In this case the concept of order numbers would be deleted from the database and thus the *delete, schema* and *object* options would be selected from the user menu. Note that deletion of only the relationship *order number* between ORDER and ORDER NUMBER would leave the concept of ORDER NUMBER in the database with all of its instantiations.

To effect the deletion of an object type, BIRD firstly deletes all the schema level facts in which it takes part. This has ramifications in the application data level, causing the deletion of all instantiations of these facts. The object type itself is then deleted causing the deletion of all associated instantiations in the application data level. Note that deletion of information at the structural schema layer does not initiate a self propagating wave of deletions, – the deletion of structural schema information leads only to the deletion of directly associated application

data information.

Deletion of information at the semantic data level is trivial since it has no propagational effects in that level or any other level. Deletion of this information signifies a relaxation of restrictions since integrity constraint facts are being deleted and thus deletion at this level cannot cause associated deletions at other levels.

9.3 Information Retrieval

As mentioned earlier in this chapter, a sophisticated query language has not been developed for BIRD and instead a simple menu driven user interface has been constructed. Specifying a retrieval request is performed in the same manner as deletion and insertion requests, – the operation, conceptual level and granularity of operation must be provided.

Selection of the *query, schema* and *object* options causes the display of all objects types at the structural schema level. If the 'fact' option is selected instead of 'object' then, having specified an object type, BIRD displays all the structural schema facts in which that object type takes part. Selection of the *query, consistency* and *fact* options similarly allows the user to inspect the semantic schema facts for a named object type. At the application data level, selection of the *query, data* and *object* options will display the values of all instantiations of a named object type. Selection of the *query, data* and *fact* options from the menu will display the facts in which a named instantiation of a named object type takes part.

9.4 Is–a Relationships

The *is–a* relationship is an exception to other relationships in BIRD since it is the only system defined relationship present at the structural schema level and may not be instantiated in the application data level. The *is–a* relationship is used in BIRD to express the fact that an object type is a sub–object type of another object type. Consequently the sub–object type inherits all the schema level facts specified for the object type.

In BIRD the *is–a* link can be used to form taxonomies of object types. The ability to specify a taxonomy is advantageous in many ways. Taxonomies facilitate economy of expression since common features of object types are specified once in the taxonomy and inherited by the appropriate object types. As well as being an elegant and natural way to model application environments, taxonomies also facilitate the process of schema restructuring, as will be shown below.

By considering an extension to the factory parts ordering database example, the utility of *is–a* taxonomies may be demonstrated. Let us assume that the manufacturing company wishes to extend the database to include information on all suppliers and customers of the firm. Suppliers to the firm receive SUPPLIER ORDERs and return SUPPLIER INVOICEs, whereas customers send CUSTOMER ORDERs and receive CUSTOMER INVOICEs. The four different object types can be elegantly organised into a taxonomy shown in figure 9.2.

The COMPANY DOCUMENT object type is the most general and covers all documents received and sent by that company. All company documents are deemed to possess a name and address. ORDER object types are COMPANY DOCUMENTS with *order number, part number* and *quantity* attributes. INVOICE object types are COMPANY DOCUMENTS with *invoice number, part number, quantity, unit cost* and *total cost* attributes. The ORDER and INVOICE object types are then further divided into sub–object types, as shown in figure 9.2.

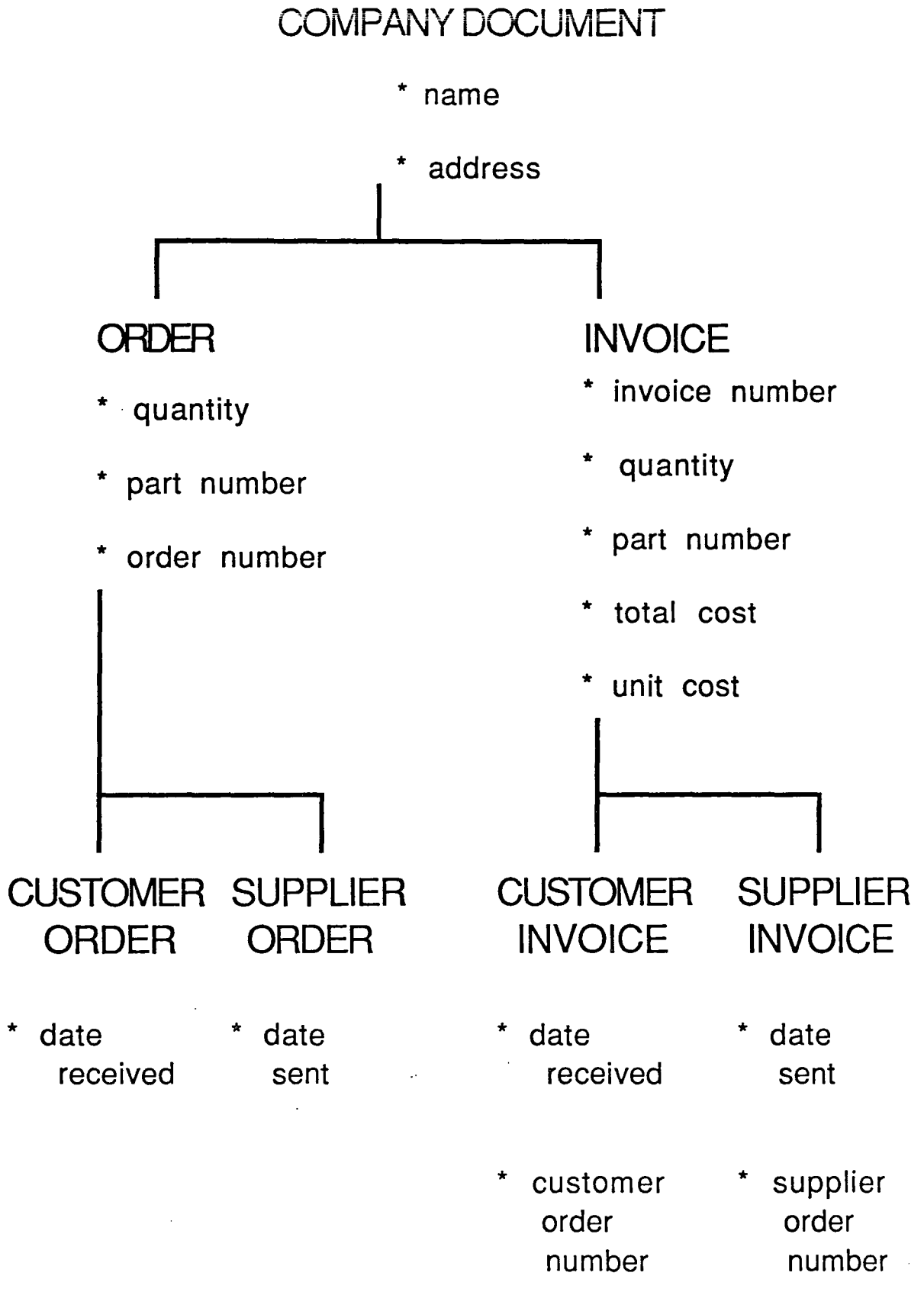


Figure 9.2: Extended Factory Parts Ordering Database Taxonomy

In BIRD such an *is–a* taxonomy may be declared since the insertion and deletion operators take into account the semantics of the *is–a* relationship. If a structural schema level fact containing an *is–a* relationship is inserted, BIRD propagates all relationships specified for the super–object type to all object types below it in the taxonomy. Depending on the domain information of the facts associated with the super–object, this may cause many associated insertions in the application data level. Whenever a fact is inserted at the structural schema level, BIRD checks both of the object types which are referenced in the fact, to determine if either are part of an *is–a* taxonomy. If either, or both, of the object types are part of a hierarchy, then BIRD ensures that the newly inserted fact is correctly inherited.

The operation of BIRD upon deletion of information at the structural schema level is similar to that upon insertion of information. If a fact is deleted containing a user–specified relationship then BIRD ensures that fact is deleted for every object type lower in the taxonomy. Note that a user may not specify the deletion of an inherited fact, – all schema facts must be deleted at the object type where they were originally specified. If a fact containing an *is–a* relationship is deleted then all facts inherited over that link are deleted from all the object types below in the taxonomy.

Integrity constraints declared in the semantic schema level are deemed to inherit over *is–a* links. This may be used to declare general forms or ranges for object type instantiations which may be further restricted for the instantiations of the sub–object types.

9.5 Comparison of BIRD with Record Oriented Implementation

The same hypothetical factory parts ordering database was entered into the INGRES 5.0 relational database and interesting contrasts were noted between the two systems. It is appreciated that INGRES is vastly superior than BIRD if compared on implementation criteria such as speed, data storage or the query language. However this section intends to elucidate the differences between the systems which arise as a cause of their different underlying representations. This section assumes the reader is aware of the information relevant to relational databases contained in the chapter describing the traditional data models.

9.5.1 Schema Definition

The relations shown in figure 9.3 were constructed to hold the information for the hypothetical parts database. Note the introduction of the COMPANY NUMBER domain which acts as a link between the INVOICE and ADDRESS relations, – the INVOICE NUMBER domain cannot be used as the index to the ADDRESS relation since the same company may submit multiple invoices. The CONNECTION relation was introduced in order to model the many to many relationship between orders and their constituent order items. To find out the order items associated with an invoice, the user must consult the CONNECTION relation to determine order item numbers associated with the order number, – these would then be used to access the ORDER ITEM relation to find out the actual items ordered and their quantities. The use of the CONNECTION relation was not mandatory, – the ORDER ITEM relation could have been accessed by order numbers, – this would reduce the relationship to one to many and might result in some duplication of information in the ORDER ITEM relation, since different orders may reference the same order items.

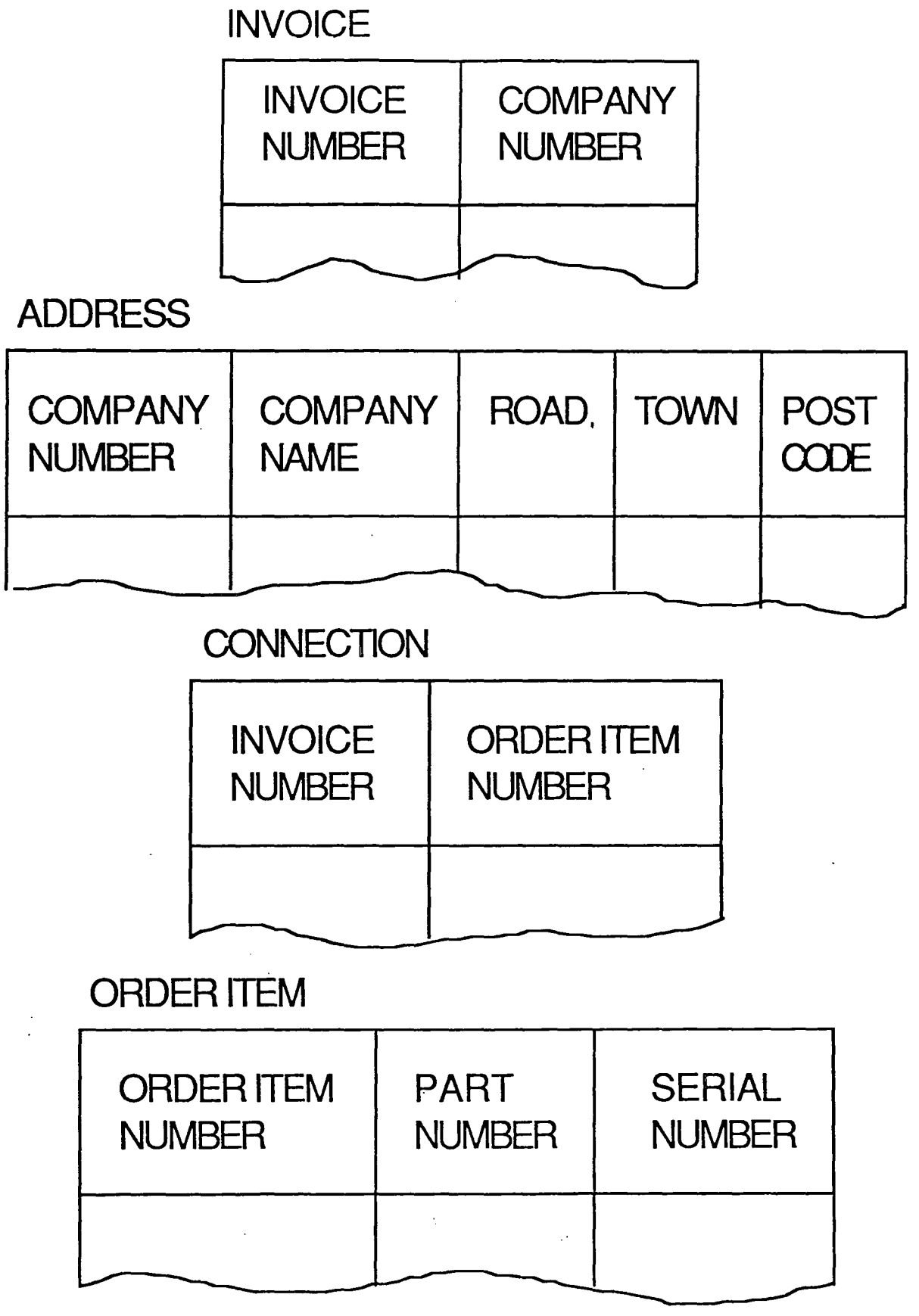


Figure 9.3: Relational Structure Declared for the Hypothetical Factory Ordering Database

A user of the relational database must discover the relationship between the information contained in the various relations since there are no explicit links between them. In BIRD all the relationships between objects in the database are stored explicitly, thus a user may navigate around the database, following links, to discover the structure of the database. Since the relationships between objects in BIRD are explicit there is no need for the definition of attributes, such as COMPANY NUMBER or the CONNECTION relation, to act as links between information.

9.5.2 Information Insertion

The lack of information describing the relationship between the domains in the various relations enables inconsistencies to be readily introduced when accessing the database using the query language. For instance entries could be made in the INVOICE relation without associated entries in the ADDRESS relation. Individual tuples in relations may also be inadequately declared, – for instance a tuple could be introduced into the INVOICE ORDER relation with a null value for the quantity.

Changes may be made to the schema by redeclaring relations and using the block copy facilities provided by INGRES. As mentioned in the earlier chapter entitled "Database Integrity", the block copy ignores any integrity constraints which have been declared for the destination relation, – providing a simple potential source of inconsistency. When new attributes are specified for relations, there is no domain information to specify its relationship with the rest of the information in the database and thus one must rely on the user to update existing tuples in the relation appropriately.

In contrast, the simple domain information supplied for BIRD relationships enables a much higher level of integrity to be maintained, since there is a specification of the dependency of object instantiations on other object instantiations.

9.5.3 Information Deletion

INGRES permits the deletion of information almost at will without regard for the consequences or semantics of the remaining information. Similar situations may arise as described for the insertion of information, – incomplete tuples and tuples referencing non–existent tuples in other relations. In contrast the BIRD system uses the domain information to ensure that referential integrity is restored after any deletion operation.

9.5.4 Information Retrieval

INGRES is a highly developed system and naturally provides advanced facilities for the retrieval of information via query languages such as the Standard Query Language, SQL and QUEL, a query language developed specifically for INGRES. Sophisticated queries may be built up which access information from multiple relations and perform arithmetic operations.

BIRD was developed as a prototype database management system and thus possesses a very simple menu driven user interface. Although the query facilities are very basic, they are also very simple to operate and are presented in a homogeneous manner to the other types of operations, – having selected the query function the user selects the conceptual level and granularity of information. In contrast the style of query used to access information in the relational data model depends upon its conceptual level, – a separate set of "help" commands must be used to access the schema level information.

9.5.5 Discussion of Comparison

It is realised that many features of INGRES which might be criticised are present in the interests of efficiency. For instance it was mentioned that the block copy instruction does not check any integrity constraints when copying data from one relation into another, but to perform this checking on a large copy of hundreds of thousands of tuples would significantly slow execution. BIRD can afford to perform extravagant integrity checks since the implementation was not oriented towards efficiency and thus it is unfair to directly compare two very different systems.

The lack of domain information inter–relating the information in a relational database allows an inconsistent database to be created easily, when accessing it via the query language. Although the necessary semantics may be built into the application programs accessing the database, the disadvantages of this approach have already been described in the chapter on database integrity. It is up to the user of a relational database to be aware of all the relations which exist and the implicit relationships between them. However in BIRD, all schema information is stored as database data and may be easily inspected. BIRD itself ensures that database integrity is maintained and leads the user through the database manipulation operations. Great freedom is afforded to the user in the method of database information and deletion, since any information at any level may be inserted or deleted without any danger of compromising integrity.

9.6 Conclusion

BIRD possesses a simple menu driven user interface, which enables the specification of database manipulation commands in a homogeneous manner over different conceptual levels. The active nature of BIRD adds to the simplicity of the user

interface, since the user is prompted to supply information necessary for the operation to be performed and the database restored to a consistent state. BIRD ensures that integrity of information is maintained at all times and can handle the definition of information taxonomies.

Owing to the fact that BIRD is a prototype, the user interface is rather basic. The deletion command needs to be developed, since at present the user may find that a small deletion has produced an unexpected amount of related deletions. Thus it would be desirable to extend the dialogue between BIRD and the user as well as specifying a comprehensive query language.

Chapter 10

Conclusions and Further Work

This chapter presents the conclusions of the work undertaken for the thesis, including suggestions of applicable further research.

The work for this thesis started with an analysis of record oriented database models and succeeded in identifying two particular problem areas, – the lack of integrity enforcement and inhomogeneity of information representation. This analysis led to the development of a new model, BIRD, whose design, implementation and evaluation are described in the thesis.

The BIRD model succeeded in fulfilling its objectives detailed in the chapter entitled "Formation of the BIRD Model". Namely, a DBMS was implemented based on a simple semantic model representation, meta–information was incorporated homogeneously as database data and emphasis was placed on integrity issues. The system was augmented with a simple menu driven interface and the representation was extended to include simple inheritance hierarchies.

The BIRD system proved very easy to use, due to the homogeneous representation of information, the advanced integrity maintenance facilities and the "active"

nature of the system. The homogeneous representation of information allows a user to view all information and meta–information as one body of database data which may be manipulated with the same set of manipulation operations. The integrity maintenance facilities ensure that there is no inconsistency between the application data and the structural and semantic schema information. BIRD affords the user the freedom to manipulate any of the data in the database, via the user interface, without fear of compromising the integrity of the database. BIRD is an "active" system making it easy for use by non–experts, – it leads the user through the database operations, prompting for information where necessary. The representation of information using binary relations is very easy for a user to perceive and the introduction of the *is–a* relationship increased the information modelling capabilities of the system. The ability to model information in hierarchies is both natural and very effective, as described in the previous chapter.

Further work may be viewed in two categories, – theoretical work on the representation of information in databases and the development of implementation issues. The binary relations used to represent information in BIRD were adequate for simple information modelling, however the limitations of the representation were more apparent at higher conceptual levels. Simple integrity constraints, expressing ranges of values or string lengths, were easily specified, however many common constraints involve more than two entities and these could not be specified. For example a useful simple constraint would specify that an annual total must be the sum of monthly totals, however this cannot be expressed using binary relations.

The representation formulated might also be extended to include nested binary relations, – this would enable representations between more than one entity to be expressed. For example if a total, A, was the sum of B and C then this could be expressed as :–

equal(A, sum(B , C))

In addition many of the modelling constructs present in the semantic models de-

scribed earlier in this thesis, such as the aggregation and grouping constructors, would provide useful extensions to the system.

As mentioned earlier in the chapter entitled "Formation of the BIRD model", it would be very interesting to extend the database to include information at higher conceptual levels than currently implemented, – such as the semantics of the database operations or rules to guide the construction of the schema. However the higher the conceptual knowledge incorporated into the database, the richer the representation that will be needed to express it. If homogeneity of information representation at all levels is to be maintained then the enrichment of the representation will be unnecessary for the representation of information at the lower conceptual levels. The implementation implications of further representational extensions must be considered, – the benefits of an extended representation must be balanced against the increased storage and processing costs.

Implementation issues in BIRD also need to be approached, particularly storage and efficiency issues. The most important development of the current system, would be to eliminate the usage of an array to store the database data, since it is an inflexible and wasteful method of information storage. The easiest modification would be to store the information in linked lists of facts, – this would economise the storage requirements and cope with varying size of databases.

The above solution does not solve the problem of persistence of information, since the linked list would still be main memory bound. Although it would be possible to transfer the contents of the linked list structure to permanent storage when it is not needed, the solution has many associated problems. The overheads of the transfer increase with the database size and would prove prohibitive with a very large application, – assuming the machine had enough main memory to hold the database. Since the database is contained in main memory, the data would be very vulnerable to system crashes and thus logging facilities would have to be developed to ensure the database could be reconstructed following a system failure.

The most realistic solution to the information storage problem is to store the information in records in files, in the same manner as standard database technology. The same problems would then have to be approached as in standard database technology, – namely the optimum organisation of the records to facilitate the expected database operations. To store the information associated with the complex modelling constructs of a semantic model would require sophisticated storage structures and would be associated with storage overheads and a reduction in access time. In the future it is possible that current work on the processing of knowledge networks may come to fruition. Fahlman [17] describes the implementation of knowledge networks using parallel architectures and Bic [2] proposes the use of dataflow graph architectures.

However the data storage and persistence problems are solved, future versions of BIRD will never be as economical with storage space or as fast as the current database technologies, since more information is stored for a given application. BIRD offers a simple user interface, advanced integrity facilities and a simple intuitive representation, – the cost of these features must be balanced against their utility for the proposed application.

Appendix A

Bibliography

[1] Abiteboul, S. and R. Hull, "IFO : A Formal Semantic Database Model," *ACM Transactions On Database Systems*, vol. 12, No.4, pp. 525–565, ACM, December 1987.

[2] Bic, L., "Processing of Semantic Nets on Dataflow Architectures," *Artificial Intelligence*, vol. 27, pp. 219–227, Elsevier Science Publishers B. V., North Holland, 1985.

[3] Brachman, R. J., "On The Epistemological Status Of Semantic Networks," *Associative Networks – Representation And Use Of Knowledge By Computers*, pp. 3–46, Academic Press, Inc, New York, 1979.

[4] Brachman, R. J., "What IS–A Is And Isn't : An Analysis of Taxonomic Links in Semantic Networks" *IEEE Computer*, vol. 16, No.10, IEEE, New York, 1983.

[5] Brown, J. S. and R. R. Burton, "Multiple Representations Of Knowledge For Tutorial Reasoning," *Representation And Understanding – Studies In Cognitive Science*, pp. 311–349, Academic Press, Inc, New York, 1975.

[6] Bryce, D. and R. Hull, "SNAP : A Graphics–Based Schema Manager," *Proceedings of the Second International Conference on Data Engineering*, pp. 151–164, IEEE, New York, February 1986.

[7] Cercone, N. and R. Goebel, "Data Bases And Knowledge Representation For Literary And Linguistic Studies," *Computers and the Humanities*, vol. 17, pp. 121-137, Elsevier Science Publishers. B.V. North Holland, 1983.

[8] Chen, P. P., "The Entity–Relational Model – Toward a Unified View of Data," *ACM Transactions on Database Systems*, vol. 1, No.1, pp. 9–36, March 1976.

[9] Codd, E. F., "A Relational Model for Large Shared Data Banks," *Communications of the ACM*, vol. 13, No.6, pp. 377–387, June 1970.

[10] Codd, E. F., "Extending the Database Relational Model to Capture More Meaning," *ACM Transactions on Database Systems*, vol. 4, No.4, pp. 397–434, December 1979.

[11] Date, C. J., *An Introduction to Database Systems*, Addison–Wesley, 1986.

[12] Davis, J. P. and R. D. Bonnell , "EDICT : An Enhanced Relational Data Dictionary : Architecture and Example," *Proceedings of the Fourth International Conference on Data Engineering*, pp. 184–191, IEEE, Los Angeles, February 1–5 1988.

[13] *CODASYL Data Base Task Group April 71 Report*, ACM, New York, 1971.

[14] *CODASYL Data Description Language Journal of Development*, Material Data Management Branch, Department of Supply and Services, Ottawa, Ontario, 1978.

[15] Dinerstein, N. T., *Database and File Management Systems for the Microcomputer*, Scott, Foresman & Co, London, 1985.

[16] Etherington, D. W. and R. Reiter, "On Inheritance Hierarchies With Exceptions," *Proceedings American Association for Artificial Intelligence*, pp. 104–108, Washington. D.C, 1983.

[17] Fahlman, S. E. and G. E. Hinton, "Connectionist Architectures for Artificial Intelligence," *Computer*, vol. 2, no. 1, pp. 100–109, IEEE, January 1987.

[18] Frost, R. A., *Database Management Systems*, Granada, London, 1984.

[19] Gray, P. M. D., G. E. Storrs and J. B. H. du Boulay, "Knowledge Representations for Database Metadata," *Artificial Intelligence Review*, Blackwell Scientific Publications vol. 2, No.1, pp. 3–30, 1988.

[20] Hammer, M. and D. McLeod, "Database Description With SDM : A Semantic Database Model," *ACM Transactions On Database Systems*, vol. 6, No.3, pp. 351–386, September 1981.

[21] Held, C. D., M. R. Stonebraker, and E. Wong, "INGRES : A Relational Database System," *Proc. ACM Pacific 75 Regional Conference*, pp. 409–416, May 1975.

[22] Hull, R. and R. King, "Semantic Data Modelling : Survey, Applications and Research Issues," *ACM Computing Surveys*, vol. 19, No.3, pp. 201–260, September 1987.

[23] *INGRES Release 5.0 VAX/VMS System Manual*, Relational Technology Inc, California, 1986.

[24] Kent, W., "Limitations of Record-Based Information Models," *ACM Transactions on Database Systems*, vol. 4, No.1, pp. 107–131, March 1979.

[25] King, R. and D. McLeod, "A Database Design Methodology and Tool for Information Systems," *ACM Transactions on Office Systems*, vol. 3, No.1, pp. 2–21, January 1985.

[26] Korth, H. F. and A. Silberschatz, *Database System Concepts*, McGraw-Hill Inc, 1986.

[27] Mayne, A. and M. B. Wood, *Introducing Relational Database*, NCC Publications, England, 1983.

[28] McGee, W., "The Information Management System IMS/VS Part 1 : General Structure and Operation," *IBM Systems Journal*, vol. 16, No.2, pp. 84–168, June 1977.

[29] McSkimin, J. R. and J. Minker, "The Use of a Semantic Net in a Deductive Question Answering System," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pp. 50–58, Cambridge, Massachusetts, August 22–25 1977.

[30] Narayanaswamy, K. and K. V. Bapa Rao, "An Incremental Mechanism for Schema Evolution in Engineering Domains," *Proceedings of the Fourth International Conference on Data Engineering*, pp. 294–301, IEEE, Los Angeles, USA, February 1 - 5, 1988.

[31] Olle, T. W., *The CODASYL Approach to Database Management*, John Wiley & Sons, New York, 1978.

[32] Orman, L., "Functional Database Constraints," *Computer Journal*, vol. 31. No.4, pp. 336–343, Cambridge University Press, 1988.

[33] Ozkarahan, E., *Database Machines and Database Management*, Prentice–Hall Inc, New Jersey, 1986.

[34] Peckham, J. and F. Maryanski, "Semantic Data Models," *ACM Computing Surveys*, vol. 20 No.3, pp. 153–189, September 1988.

[35] Piercy, R. M. and A. J. Slade, "Binary Relation Database : Issues of Representation and Implementation," *Proceedings of the IXth Conference of the Latin American Computer Society*, Santiago, Chile, pp. 146–157, July 10 - 14, 1989.

[36] Potter, W. D. and R. P. Trueblood, "Traditional, Semantic and Hyper–Semantic Approaches to Data Modelling," *IEEE Computer*, vol. 21, No.6, pp. 53–63, June 1988.

[37] Quillian, M. R., *Semantic Memory - Ph.D. Dissertation*, Carnegie Institute Of Technology, Pittsburgh, Pa, 1966.

[38] Quillian, M. R., "Semantic Memory," *Semantic Information Processing*, pp. 227–270, MIT Press, Cambridge, Massachusetts, 1968.

[39] Rishe, N., "On Representation of Medical Knowledge by a Binary Data Model," *Proceedings of the Fifth International Conference on Mathematical Modelling*, 1986.

[40] Schmidt, J. W. and M. L. Brodie eds, *Relational Database Systems : Analysis and Comparison*, Springer–Verlag, Berlin, 1983.

[41] Schwarcz, R. M., J. F. Burger, and R. F. Simmons, "A Deductive Question-Answerer for Natural Language Inference," *Communications of the ACM*, vol. 13, No.3, pp. 167–183, ACM, March 1970.

[42] Shapiro, S. C., "The SNePS Semantic Network Processing System," *Associative Networks – Representation and Use of Knowledge by Computers*, pp. 179–203, Academic Press, Inc, New York, 1979.

[43] Stocker, P. M., P. M. D. Gray, and M. P. Atkinson eds, *Databases – Role and Structure*, Cambridge University Press, 1984.

[44] Su, S. Y. W., "Modeling integrated manufacturing data with SAM*," *IEEE Computer Magazine*, pp. 34–49, January 1986.

[45] Sundgren, B., *Data Bases and Data Models*, Studentlitteratur, Sweden, 1985.

[46] Teory, T. J., D. Yang, and J. P. Fry, "A Logical Design Methodology for Relational Database using the Extended Entity–Relationship Model," *ACM Computing Surveys*, vol. 18, No.2, pp. 197–222, June 1986.

[47] Tsichritzis, D. C. and F. H. Lochovsky, *Hierarchical Data Base Management Systems*, Academic Press, New York, 1977.

[48] Wenger, E., *Artificial Intelligence and Tutoring Systems*, Morgan Kaufmann, California, 1987.

[49] Woods, W. A., "What's in a Link : Foundations for Semantic Networks," *Representation and Understanding – Studies in Cognitive Science*, pp. 35–82, Academic Press, Inc, New York, 1975.

[50] Zloof, M. M., "Query–By–Example," *Proc. AFIPS Conference*, pp. 431–438, 1975.

Appendix B

Constituent Functions of BIRD

B.1 BIRD DATA TYPES

DataFactInstType – contains coordinates of relation , subject and object comprising the data fact. The relation is an object type coordinate, the other two arguments are specified by object instance coordinates.

DefinitionDomainType – a flag indicating a schema fact is mandatory or optional.

DuplicationDomainType – a flag indicating a schema fact is single or multi valued.

FileNameType – the name of a file holding a database 'snapshot'.

Files – a file handle used by a modula–2 program to refer to a file.

NameType – the name of an object instance or object type.

NumberType – the numerical value of an object instance or object type.

ObjectInstContentType – describes the contents at each object instance node, – empty, name, number, abstract.

ObjectInstDescType – an object instance descriptor comprising three fields, ObjectInstContentType, NameType and NumberType.

ObjectInstFactCoordType – 3 dimensional coordinate, made up of object type index, fact type index and object instance index .

ObjectInstFactInstCoordType – 4 dimensional coordinate, made up of object type index, fact type index, object instance index and fact instance index.

ObjectTypeContentType – describes the contents at each object type node, – unused, relation, nameobject, constant.

ObjectTypeCoordType – 1 dimensional coordinate, made up of the object type index.

ObjectTypeDescType – an object type descriptor comprising three fields, ObjectTypeContentType, NameType and NumberType.

ObjectTypeFactCoordType – 2 dimensional coordinate, made up of object type index and fact type index.

SchemaConsisFactInstType – contains coordinates of relation, subject and object which comprise a schema or consistency fact. All the constituents of this type are object type coordinates.

B.2 Level 0 – L0

InsertFactL0

accept FICoord : ObjectInstFactInstCoordType ;
DataFactInst : DataFactInstType ;
SchConFactInst : SchemaConsisFactInstType ;

return NULL

action Insert either data, schema or consistency fact at the location specified.

DeleteFactL0

accept FICoord : ObjectInstFactInstCoordType ;

return NULL

action Delete either data, schema or consistency fact at the location specified.

CheckFactL0

accept FICoord : ObjectInstFactInstCoordType ;

action DataFactInst : DataFactInstType ;
SchConFactInst : SchemaConsisFactInstType ;
Present : Boolean ;

action Return the data, schema or consistency fact at the location specified, set the boolean variable if a fact is present.

InsertNameL0

accept FICoord : ObjectInstFactInstCoordType ;
Name : NameType ;

return NULL

action Insert name at the object instance or object type location specified.

InsertNumberL0

accept FICoord : ObjectInstFactInstCoordType ;
Number : NumberType ;

return NULL

action Insert number at the object instance or object type location specified.

SetCellContentL0

accept FICoord : ObjectInstFactInstCoordType ;
DataContent : ObjectInstContentType ;
SchemaContent : ObjectTypeContentType ;

return NULL

action Set the object instance or object type content flag at the location specified to the value given, describing the type of that object instance or type.

RetrieveCellContentL0

accept FICoord : ObjectInstFactInstCoordType ;

return DataContent : ObjectInstContentType ;
SchemaContent : ObjectTypeContentType ;

action Retrieve the object instance or object type content flag at the location specified.

SetObjectInstTypeL0

accept FICoord : ObjectInstFactInstCoordType ;
ObjectInstType : ObjectInstContentType ;

return NULL

action Sets the object instance type descriptor of object types, describing the type of object instance which may be instantiated.

CheckObjectInstTypeL0

accept FICoord : ObjectInstFactInstCoordType ;

return ObjectInstType : ObjectInstContentType ;

action Returns the object instance type descriptor at the object type specified.

DeleteNameL0

accept FICoord : ObjectInstFactInstCoordType ;

return NULL

action Delete the object instance or object type name at the location specified.

DeleteNumberL0

accept FICoord : ObjectInstFactInstCoordType ;

return NULL

action Delete the object instance or object type number at the location specified.

RetrieveNameL0

accept FICoord : ObjectInstFactInstCoordType ;

return Name : NameType ;

action Return the object instance or object type name at the location specified.

RetrieveNumberL0

accept FICoord : ObjectInstFactInstCoordType ;

return Number : NumberType ;

action Return the object instance or object type number at the location specified.

DuplicationFlag

accept FICoord : ObjectInstFactInstCoordType ;
DuplicationFlag : DuplicationDomainType ;

return NULL ;

action Set the duplication flag of the schema fact specified to the value provided.

CheckDuplicationFlag

accept FICoord : ObjectInstFactInstCoordType ;

return DuplicationFlag : DuplicationDomainType ;

action Return the duplication flag of the schema fact specified.

MandatoryFlag

accept FICoord : ObjectInstFactInstCoordType ; DefinitionFlag : DefinitionDomainType ;

return NULL ;

action Set the definition flag of the schema fact specified to the value provided.

CheckMandatoryFlag

accept FICoord : ObjectInstFactInstCoordType ;

return DefinitionFlag : DefinitionDomainType ;

action Return the definition flag of the schema fact specified.

OpenFileForStorage

accept StoreFileName : FileNameType ;

return StoreFile : Files ;
AllOk : BOOLEAN ;

action Open named file for writing, return file handle and boolean indicating the success of the operation.

OpenFileForRead

accept ReadFileName : FileNameType ;

return ReadFile : Files ;
AllOk : BOOLEAN ;

action Open named file for reading, return file handle and boolean indicating the success of the operation.

DeleteFile

accept DeleteFileName : FileNameType ;

return AllOk : BOOLEAN ;

action Delete named file and return boolean indicating the success of the operation.

StoreDB

accept StoreFile : Files ;

return AllOk : BOOLEAN ;

action Store a complete database snapshot in the named file and return boolean indicating the success of the operation.

RestoreDB

accept RestoreFile : Files ;

return AllOk : BOOLEAN ;

action Restore complete database snapshot from the named file and return boolean indicating the success of the operation.

B.3 Level 1 – L1

InsertFactInstL1

accept ObjectInstFact : ObjectInstFactCoordType ;
DataFact : DatFactInstType ;

return NULL

action Insert data fact at the first free fact instance position in the object instance fact type location specified.

DeleteFactInstL1

accept ObjectInstFact : ObjectInstFactCoordType ;
DeleteDataFact : DataFactInstType ;

return Success : Boolean

action Look through the data fact instances at the object instance fact type location specified. If the data fact specified if found then delete it, return boolean variable indicating the success of the operation.

RetrieveFirstFactInstL1

accept ObjectInstFact : ObjectInstFactCoordType ;

return DataFact : DataFactInstType ;
Success : Boolean

action Return the first data fact at the object instance fact type location specified, return boolean variable indicating the success of the operation.

RetrieveNextFactInstL1

accept ObjectInstFact : ObjectInstFactCoordType ;
CurrentDataFact : DataFactInstType ;

return NextDataFact : DataFactInstType ; Success : Boolean

action Return the first data fact after the data fact provided at the object instance fact type location specified, return boolean variable indicating the success of the operation.

InsertFactTypeL1

accept ObjectInst : ObjectInstCoordType ;
SchConFact : SchemaConsisFactInstType ;

return ObjectInstFact : ObjectInstFactCoordType ;

action Insert the schema or consistency fact at the first free fact type location for the object type specified, return the coordinate of the fact.

DeleteFactTypeL1

accept ObjectInst : ObjectInstCoordType ;

return Success : BOOLEAN ;

action Delete the schema or consistency fact at the fact type location specified, return a boolean variable indicating the success of the operation.

CheckFactTypeL1

accept ObjectInst : ObjectInstCoordType ;

return SchConFact : SchemaConsisFactInstType ; Success : BOOLEAN ;

action Return the schema or consistency fact at the fact type location specified, return a boolean variable indicating the success of the operation.

InsertObjectInstL1

accept ObjectType : ObjectTypeCoordType ;
ObjectInstDesc : ObjectInstDescType ;

return ObjectInst : ObjectInstCoordType ;

action Create an object instance of the object type specified, instantiating it with the details provided in the object instance descriptor, return the coordinate of the new object instance.

DeleteObjectInstL1

accept ObjectInst : ObjectInstCoordType ;

return NULL ;

action Reset the object instance descriptor at the location specified.

CheckObjectInstL1

accept ObjectInst : ObjectInstCoordType ;

return ObjectInstDesc : ObjectInstDescType ;
Success : BOOLEAN ;

action Return the object instance descriptor at the location specified,. and a boolean variable indicating whether an object instance was found.

ReplaceObjectInstDescL1

accept ObjectTypeCoord : ObjectTypeCoordType ;
ObjectInstDesc : ObjectInstDescType ;
NewObjectInstDesc : ObjectInstDescType ;

return NULL

action Replace the object instance of the object type specified with the new object instance descriptor provided.

InsertObjectTypeL1

accept ObjectTypeDesc : ObjectTypeDescType ;
ObjectInstType : ObjectInstContentType ;

return ObjectType : ObjectTypeCoordType ;

action Insert a new object type at the first free object type location, assigning it the object type descriptor and description of the type of its object instances.

DeleteObjectTypeL1

accept ObjectType : ObjectTypeCoordType ;

return NULL ;

action Reset the object type descriptor and object instance type descriptor at the location specified.

CheckObjectTypeL1

accept ObjectType : ObjectTypeCoordType ;

return ObjectTypeDesc : ObjectTypeDescType ;
ObjectInstType : ObjectInstContentType ;
Success : BOOLEAN ;

action Return the object type descriptor and object instance descriptor of at the object type location specified.

SetFactMandatoryL1

accept ObjectTypeFact : ObjectTypeFactCoordType ;

return NULL

action Set definition flag of specified schema fact to mandatory.

SetFactOptionalL1

accept ObjectTypeFact : ObjectTypeFactCoordType ;

return NULL

action Set definition flag of specified schema fact to optional.

IsFactMandatoryL1

accept ObjectTypeFact : ObjectTypeFactCoordType ;

return FactMandatory : Boolean ;

action Check definition flag of specified schema fact, return status.

SetFactSingleL1

accept ObjectTypeFact : ObjectTypeFactCoordType ;

return NULL

action Set singlevalue flag of specified schema fact to single–valued.

SetFactMultiL1

accept ObjectTypeFact : ObjectTypeFactCoordType ;

return NULL

action Set singlevalue flag of specified schema fact to multi-valued.

IsFactSingleL1

accept ObjectTypeFact : ObjectTypeFactCoordType ;

return FactSingle : Boolean ;

action Check singlevalue flag of specified fact, return status.

B.4 Level 2 – L2

B.4.1 Level 2 Deletion – DEL

DeleteObjectInstL2DEL

accept ObjectType : ObjectTypeCoordType ;
ObjectInstDesc : ObjectInstDescType ;

return Success : BOOLEAN ;

action Delete the object instance of the object type specified, returning a boolean variable to indicate if the operation was successful.

DeleteObjectTypeL2DEL

accept ObjectType : ObjectTypeCoordType ;

return Success : BOOLEAN ;

action Delete the object type at the object type coordinate specified, return a boolean variable indicating if the object type was found and successfully deleted.

DeleteDataFactL2DEL

accept DataFact : DataFactInstType ;
ObjectType1Fact : ObjectTypeFactCoordType ;
ObjectInst1Desc : ObjectInstDescType ;

return ObjectType2Fact : ObjectTypeFactCoordType ;
ObjectInst2Desc : ObjectInstDescType ;
Success : BOOLEAN ;

action Delete the data fact for the object type and object instance provided. Delete the data fact entry for the other object instance in the fact, and return the object type and object instance descriptor of the other argument. Return a boolean variable to indicate whether the deletion operation succeeded.

DeleteSchemaFactL2DEL

accept SchemaFact : SchemaConsisFactInstType ;

return Success : BOOLEAN ;

action Delete the schema fact for both the argument object types of the fact.

DeleteConsistencyFactL2DEL

accept ConsistencyFact : SchemaConsisFactInstType ;

return Success : BOOLEAN ;

action Delete the consistency fact for both the argument object types of the fact.

B.4.2 Level 2 Input – IP

SpecifyObjectTypeL2IP

accept NewObjectTypeAllowed : BOOLEAN ;
NoObjectTypeAllowed : BOOLEAN ;
ObjectTypeSpecified : BOOLEAN ;

return ObjectTypeDesc : ObjectTypeDescType ;
ObjectInstType : ObjectInstContentType ;
Success : BOOLEAN ;

action Permits the user to specify an object type, according to various restrictions. NewObjectTypeAllowed specifys whether the user may enter details on a new object type. NoObjectTypeAllowed specifys whether the user may refuse to specify an object type. ObjectTypeSpecified indicates whether the ObjectTypeContentType field of the ObjectTypeDesc has already been set, restricting the type of object type which may be selected. The ObjectTypeDesc and ObjectInstType of the object type selected is returned with a boolean variable to indicate whether an object type was successfully selected.

SpecifyObjectInstL2IP

accept NewObjectInstAllowed : BOOLEAN ;
NoObjectInstAllowed : BOOLEAN ;
ObjectType : ObjectTypeCoordType ;

return ObjectInstDesc : ObjectInstDescType ;
Success : BOOLEAN ;

action Permits the user to specify an object instance of the object type specified, according to various restrictions. NewObjectInstAllowed specifys whether the user may enter details on a new object inst. NoObjectInstAllowed specifys whether the user may refuse to specify an object instance. The ObjectInstDesc of the object instance selected is returned with a boolean variable to indicate whether an object instance was successfully selected.

SpecifySchemaFactL2IP

accept ObjectType : ObjectTypeCoordType ;

return SchemaFact : SchemaConsisFactInstType ;
Success : BOOLEAN ;

action Select a schema fact of the object type specified, return the schema fact plus a boolean variable to indicate whether the schema fact was successfully selected.

SpecifyConsistencyFactL2IP

accept ObjectType : ObjectTypeCoordType ;

return ConsistencyFact : SchemaConsisFactInstType ;
Success : BOOLEAN ;

action Select a consistency fact of the object type specified, return the consistency fact plus a boolean variable to indicate whether the consistency fact was successfully selected.

SpecifyDataFactL2IP

accept ObjectTypeFact : ObjectTypeFactCoordType ;
ObjectInstDesc : ObjectInstDesc ;

return CurrentDataFact : DataFactInstType ;
Success : BOOLEAN ;

action Select a data fact from the fact type location of the object instance specified plus a boolean variable to indicate whether a data fact was successfully selected.

GetDomainInfoL2IP

accept NULL ;

return Mandatory : Boolean ;
Single : Boolean ;

action Ask the user to answer y/n to questions on the two domains.

B.4.3 Level 2 Insertion – IN

InsertObjectInstL2IN

accept ObjectType : ObjectTypeCoordType ;
ObjectInstDesc : ObjectInstDescType ;

return NULL ;

action Insert the specified object instance at the first free position, having checked that the object instance does not already exist.

ReplaceObjectInstDescL2IN

accept ObjectTypeCoord : ObjectTypeCoordType ;
ObjectInstDesc : ObjectInstDescType ;
NewObjectInstDesc : ObjectInstDescType ;

return NULL ;

action Overwrite the ObjectInstDesc of the ObjectTypeCoord with the new ObjectInstDesc value specified.

InsertObjectTypeL2IN

accept ObjectTypeDesc1 : ObjectTypeDescType ;
ObjectInstType1 : ObjectInstContentType ;

return ObjectType : ObjectTypeCoordType ;
Success : BOOLEAN ;

action Check the object type does not already exist, insert it in the first available location. Return the object type's coordinate and a boolean variable to indicate whether the operation was performed successfully.

InsertDataFactL2IN

accept SchemaFact : SchemaConsisFactInstType ;
SubjectOInstDesc1 : ObjectInstDescType ;
ObjectOInstDesc1 : ObjectInstDescType ;

return NULL ;

action Instantiate the schema fact for the two object instances specified, thus this procedure inserts two data facts.

InsertSchemaFactL2IN

accept SchemaFact : SchemaConsisFactInstType ;
SubDomain : DomainInformationType ;
ObjDomain : DomainInformationType ;

return NULL ;

action Insert the schema fact for the subject and object of the fact and insert the domain information provided. This procedure checks against duplication of facts.

InsertConsistencyFactL2IN

accept ConsistencyFact : SchemaConsisFactInstType ;

return NULL ;

action Insert the consistency fact for the two argument object types of the consistency fact. This procedure checks against duplication of facts.

B.4.4 Level 2 Database Integrity DI

InsertFactInstDomainOkL2DI

accept SchemaFact : ObjectTypeFactCoordType ;
ObjectInstDesc : ObjectInstDescType ;

return InsertOk : BOOLEAN ;

action Checks whether the specified schema fact may be instantiated for the object instance described, – this relies on the value of the duplication domain of the schema fact and whether the schema fact has already been instantiated. A boolean variable is returned to indicate if the insertion may proceed.

ObjectInstIntegrityOkL2DI

accept ObjectInstDesc : ObjectInstDescType ;
ObjectType : ObjectTypeCoordType ;

return IntegrityOk : Boolean ;

action Consult the consistency facts for the ObjectType specified, checking the restrictions against the value of the ObjectInstDesc supplied. A boolean variable is returned to indicate if the object instance description contradicts consistency information.

ObjectInstFactTypeDomainOkL2DI

accept SchemaFact : ObjectTypeFactCoordType ;
ObjectInstDesc : ObjectinstDescType ;

return DomainOk : Boolean ;

action Checks that the data facts in the fact type location of the object instance specified do not conflict with the domain information of the schema fact, this involves checking the duplication and definition values for the schema fact.

AllObjectInstFactsL2DI

accept ObjectInstDesc : ObjectinstDescType ;

return DomainOk : Boolean ;

action For every fact type of the specified object instance call ObjectInstFactTypeDomainOkL2DI, return a boolean variable to indicate whether all the object instance facts are in agreement with the schema fact domain information.

ConsistencyArcL2DI

accept ObjectTypeDesc : ObjectTypeDescType ;

return LegalConsistencyArc : BOOLEAN ;

action Tests the object type descriptor and sets the boolean variable value according to whether the object type is a legal consistency arc.

SchemaFactPresentL2DI

accept SchemaFactCoord : ObjectTypeFactCoordType ;

return FactPresent : Boolean ;

action Sets the boolean variable according to whether a schema fact exists at that location.

SchemaFactInstantiatedL2DI

accept SchemaFactCoord : ObjectTypeFactCoordType ; ObjectInstDesc : ObjectInstDescType ;

return Instantiated : Boolean ;

action Sets the boolean variable according to whether that schema fact has been instantiated for that particular object instance.

SchemaFactMandatoryL2DI

accept SchemaFactCoord : ObjectTypeFactCoordType ;

return FactMandatory : Boolean ;

action Sets the boolean variable according to whether the specified schema fact is mandatory.

SchemaFactUniqueL2DI

accept SchemaFactCoord : ObjectTypeFactCoordType ;

return FactUnique : Boolean ;

action Sets the boolean variable according to whether the specified schema fact is unique.

B.4.5 Level 2 Output – OP

OneSchemaFactL2OP

accept SchemaFactLocn : ObjectTypeFactCoordType ;

action Output the schema fact at the location specified.

SchemaFactsL2OP

accept ObjectType : ObjectTypeCoord ;

action Output all the schema facts of the object type specified.

ObjectInst2L2OP

accept ObjectInstDesc : ObjectInstDescType ;

action Display the value of the object instance descriptor.

ObjectInstFactsL2OP

accept ObjectTypeLocn : ObjectTypeCoordType ;
ObjectInstDesc : ObjectInstDescType ;

action Output all the data facts which the specified object instance takes part in.

AllObjectInstsL2OP

accept ObjectType : ObjectTypeCoordType ;

action Output the object instance descriptors for every object instance of the specified object type .

AllObjectInstsFactsL2OP

accept ObjectType : ObjectTypeCoordType ;

action Output the data facts for every object instance of the specified object type .

ObjectType2L2OP

accept ObjectTypeCoord ; ObjectTypeCoordType ;

action Output the object type descriptor of the object type specified.

AllObjectTypeL2OP

accept NULL

action Output the object type descriptor of every object type.

ConsistencyFactsL2OP

accept ObjectTypeCoord : ObjectTypeCoordType ;

action Output all the consistency facts for the object type specified.

ConsistencyRelationsL2OP

accept NULL ;

action Output the names of all the legal consistency relations.

RelationObjectTypeL2OP

accept NULL ;

action Output all the object types which represent relations.

ConstantObjectTypeL2OP

accept NULL ;

action Output all the object types which represent constants.

NameObjectTypeL2OP

accept NULL ;

action Output all the object types which represent names.

B.4.6 Level 2 Retrieve – RET

RetrieveFirstFactInstL2RET

accept ObjectInstDesc : ObjectInstDescType ;
ObjectTypeFact : ObjectTypeFactCoordType ;

return DataFact : DataFactInstType ;
Success : Boolean

action Locate the object instance described and return the first data fact in the appropriate fact type location supplied. A boolean variable is also returned indicating whether a data fact was retrieved.

RetrieveNextFactInstL2RET

accept ObjectInstDesc : ObjectInstDescType ;
ObjectTypeFact : ObjectTypeFactCoordType ;
NextDataFact : DataFactInstType ;

return DataFact : DataFactInstType ;
Success : Boolean

action Locate the data fact supplied in the fact type location of the object instance described and return the next data fact. A boolean variable is also returned indicating whether a data fact was retrieved.

RetrieveFirstObjectInstL2RET

accept ObjectTypeCoord : ObjectTypeCoordType ;

return ObjectInstDesc : ObjectInstDescType ;
Success : Boolean

action Return the first object instance found for the object type supplied. A boolean variable is also returned indicating whether an object instance was retrieved.

RetrieveNextObjectInstL2RET

accept ObjectTypeCoord : ObjectTypeCoordType ;

return ObjectInstDesc : ObjectInstDescType ;
NextObjectInstDesc : ObjectInstDescType ;
Success : BOOLEAN ;

action Locate the object instance for the object type supplied and return the object instance which follows it. A boolean variable is also returned indicating whether an object instance was retrieved.

RetrieveConsistencyFactL2RET

accept ObjectTypeFactCoord : ObjectTypeFactCoordType ;

return ConsistencyFact : SchemaConsisFactInstType ;
Success : BOOLEAN ;

action Retrieve the consistency fact at the location supplied. A boolean variable is also returned indicating whether consistency fact was retrieved.

RetrieveSchemaFactL2RET

accept ObjectTypeFactCoord : ObjectTypeFactCoordType ;

return SchemaFact : SchemaConsisFactInstType ; Success : BOOLEAN ;

action Retrieve the schema fact at the location supplied. A boolean variable is also returned indicating whether schema fact was retrieved.

RetrieveSchemaFactCoordL2RET

accept ObjectType : ObjectTypeCoordType ;
SchemaFact : SchemaConsisFactInstType ;

return ObjectTypeFactCoord : ObjectTypeFactCoordType ;
Success : BOOLEAN ;

action Retrieve the schema fact coordinate for the object type specified. A boolean variable is also returned indicating whether the schema fact was found.

RetrieveConsistencyFactCoordL2RET

accept ObjectType : ObjectTypeCoordType ;
ConsistencyFact : SchemaConsisFactInstType ;

return ObjectTypeFactCoord : ObjectTypeFactCoordType ;
Success : BOOLEAN ;

action Retrieve the consistency fact coordinate for the object type specified. A boolean variable is also returned indicating whether the consistency fact was found.

RetrieveObjectTypeCoordL2RET

accept ObjectTypeDesc : ObjectTypeDescType ;

return ObjectType : ObjectTypeCoordType ;
Success : BOOLEAN ;

action Retrieve the object type coordinate for the object type specified. A boolean variable is also returned indicating whether the object type was found.

RetrieveObjectTypeDescL2RET

accept ObjectType : ObjectTypeCoordType ;

return ObjectTypeDesc : ObjectTypeDescType ;
Success : BOOLEAN ;

action Retrieve the object type descriptor for the object type at the coordinate specified. A boolean variable is also returned indicating whether the object type was found.

B.5 Level 3 – L3

None of the procedures at this level have any parameters.

B.5.1 Level 3 Deletion – DEL

DeleteDataFactL3DEL

action User chooses a data fact which is then deleted. The definition of object instances which form the subject and object of the data fact are checked to ensure they are still legal, if they are not legal then the object instances themselves are deleted with all their associated data facts, and the process stops when the database is left in a consistent state with respect to the domain information.

DeleteObjectInstL3DEL

action User chooses an object instance which is deleted. All the data facts associated with that object instance are deleted and the object instances which are involved in those data facts are checked for legality with respect to the domain information which may lead to additional object instance deletions. The process stops when the database is left in a consistent state with respect to the domain information.

DeleteSchemaFactL3DEL

action User chooses a schema fact which is deleted. The corresponding data facts for all the object instances of the subject and object of the schema fact are also deleted, however the wave of deletions will not spread any further than this.

DeleteConsistencyFactL3DEL

action User chooses a consistency fact which is deleted. This has no further effect on the data base as it is relaxing a restriction and thus the database is guaranteed to be consistent after the deletion if it was consistent before the deletion.

DeleteObjectTypeL3DEL

action User chooses an object type which is deleted. The schema facts of the object type, all the object instances and their associated data facts are deleted.

B.5.2 Level 3 Insertion – IN

InsertObjectInstL3IN

action User supplies details of an object instance which is inserted and the system ensures that all appropriate information is supplied for it to make it legal with respect to its domain and consistency information.

InsertObjectTypeL3IN

action User supplies details of an object type which is inserted.

InsertSchemaFactL3IN

action User specifies details of a schema fact which is inserted, – the system then checks all object instances of arguments of the schema fact to ensure that they are legal with respect to its domain information.

InsertConsistencyFactL3IN

action User specifies a consistency fact, the system then checks all instances of the subject of the fact to ensure that they are legal with respect to the new consistency information. The object of a consistency fact is always a constant, and thus it has no instances which need to be checked.

InsertDataFactL3IN

action User supplies details of a data fact which is inserted if it does not compromise the domain information of the schema fact of which it is an instantiation.

B.5.3 Level 3 Output – OP

OutputObjectTypesL3OP

action All the object types in the database are displayed.

OutputSchemaFactsL3OP

action An object type is selected by the user, all of its schema facts are displayed.

OutputConsistencyFactsL3OP

action An object type is selected by the user, all of its consistency facts are displayed.

OutputObjectInstL3OP

action An object type is selected by the user, all of its object instances are displayed.

OutputObjectInstFacts

action An object instance is selected by the user, all of its data facts are displayed.

B.6 Level 4 Menu

Menu

action Uses the level 3 procedures to display information to the user and permit the user to perform the various database manipulation operations.