A Collection Model for Data Management in Object-Oriented Systems

Moira C. Norrie

A thesis submitted for the degree of Doctor of Philosophy,

to the

Faculty of Science, University of Glasgow

December 1992.

© Moira C. Norrie

ProQuest Number: 13818579

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 13818579

Published by ProQuest LLC (2018). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Thesis
9381
Copy 1

GLASGOW
UNIVERSITY
LIBRARY

Abstract

This thesis addresses the question of how to provide data management services in object-oriented systems with reliable persistent object stores. It proposes an object data model, called the collection model, which serves as a foundation for the construction of such services. The collection model is general in that it is independent of any particular implementation platform. In part, this independence is achieved through the separation of the data model from the underlying type model.

There are two components of the collection model - a structural model, BROOM, and an operational model based on an algebra of collections. The structural model is semantically rich and exhibits properties of both the entity-relationship and semantic data models. Unary collections are used to represent entity categories and binary collections to represent relationships between entities. Classification structures are based on the notion of a collection family which represents various forms of conceptual dependencies among the collections of a family.

The requirements for supporting the various forms of evolution in object-oriented database systems are presented. An extension to the collection model is proposed to support object evolution whereby objects can migrate within classification structures.

Two existing realisations of the collection model are described. One is a prototype, single-user system implemented in Prolog. The other forms the basis of the Object Data Management Services of the Comandos platform for distributed, object-oriented applications.

A general approach to object data model design, specification and realisation is advocated. In particular, a metacircular description of the collection model is used as an intermediate form of data model specification. This metacircular description is then transformed into a formal specification in the Z language.

Acknowledgements

Much of the work reported in this thesis forms part of the Esprit project 2071, Comandos. I acknowledge the contributions of the rest of the Comandos project group at Glasgow, particularly those of the ODMS group: they are Steve Blott, Colin Dunlop, David Harper, Arkadi Kosmynin and Andrew Walker. Special thanks are due to David Harper for taking on the responsibility for the ODMS group and the supervision of this thesis. His contributions to the design of the collection model were invaluable. The implementation of the ODMS was undertaken by Steve Blott and Andrew Walker and they, along with David, should take the credit for turning a dream into reality. Colin Dunlop and Arkadi Kosmynin were responsible for the development of database tools for use with the collection model. Further thanks are due to Steve for his comments on drafts of this thesis and to Colin for his considerable help with the diagrams.

Finally, I give a special mention to four very good friends: Les Jeeves, Terttu Orci, Michael Rosier and Jozef Swiatycki. I owe it all to them.

Contents

1 Introduction 1

1.1 Database System Requirements 1

1.1.1 Effectual Data Management 3

1.1.2 Expedient Data Management 6

1.1.3 Efficient Data Management 11

1.2 The Collection Model 14

1.3 Structure of Thesis 17

2 Foundations of Data Models 19

2.1 Philosophical Foundations 20

2.2 A General Framework 25

2.3 Data Models 29

2.4 Object-Oriented Database Management Systems 35

3 The Structural Model: BROOM 40

3.1 An Overview of BROOM 42

3.2 Collections and Collection Families 49

3.2.1 Collection Families 51

3.3 A Metacircular Description of BROOM 55

3.4 A Z Specification of BROOM 64

4 Semantic Modelling in BROOM **80**

4.1 Relationships 82

4.2 Aggregation, Association and Generalisation 87

5 The Operational Model **97**

5.1 Operational Levels 98

5.2 A Collection Algebra 102

5.2.1 Operations on Collections 104

5.2.2 Operations on Binary Collections 107

5.2.3 Operations on Ordered Collections 109

5.2.4 Operations on Sets 110

5.2.5 Operations on Bags 111

5.2.6 Specifying Collection Operations in Z 113

5.3 Example Queries 115

5.4 Properties of the Collection Algebra 119

6 Evolution **123**

6.1 Object Creation and Deletion 124

6.2 Object Evolution 133

6.3 Schema Evolution 141

7 Realising the Collection Model 144

7.1 COLLEEN 145

7.2 The Comandos ODMS 149

8 Conclusions 155

8.1 Contributions 155

8.2 Further Work 159

Chapter 1

Introduction

With the advent of persistent programming systems and persistent object stores, some of the functionality of a database management system (DBMS) is provided by a general persistent system. We are forced to ask ourselves questions such as: "What functionality of a DBMS is not supported by a persistent store?", "What distinguishes a database system from a persistent system?".

To answer these questions, we must first examine the basic requirements of a database system. This enables us to establish a clear characterisation of both database systems and persistent systems and thereby assert the additional functionality required of a database management system. We can then address the issue of how to provide this functionality of a database management system in an object-oriented system with a persistent object store.

1.1 Database System Requirements

A database system is a software system which supports a data-intensive application. In the 60's and 70's, such applications tended to be relatively simple and straightforward in that the structure and use of data was regular and well-known. For example, database systems were developed for stock control, payroll and patient record applications.

As database technology advanced, so did the ambitions of the developers of database application systems. This is not to say that there was no longer a requirement for the more traditional applications, but rather that support was sought for more general and complex application systems such as Office Information Systems, Computer

Aided Design and Computer Assisted Software Engineering. These application systems model complex human processing systems and support a wide range of activities. For example, an Office Information System encompasses support for all office activities rather than supporting a single activity such as the maintenance of personnel records.

To provide computer support for an activity, we must first be able to describe that activity and, in the case of complex human activities such as those of designing a ship or producing a large software system, it is an intricate task to produce such a description. One of the aims of the designers of a database management system is to provide the database application system designer with a set of concepts that assist the construction of such an application model.

The main characteristic of all data-intensive applications is the representation of a large number of interrelated entities of the application domain. Each entity will be represented in the database system by a value or an association of values. Here, the term 'value' is used in its most general sense to mean any valid data item of a particular system. A value may be a simple value, such as the integer 3 or the string `john`, or a complex record or object value.

Instead of an entity being represented directly by a single value, it may be the case that it is represented by an association of values. For example, a particular person might be represented by the association of the string value `john` and the string value `24 High Street`. Associations of values are also used to represent relationships between entities. If a particular person entity is represented by the value p, and a particular department entity is represented by the value d, then the fact of that person being employed by that department may be represented by the association (p, d).

The form of representation of an entity, or a relationship between entities, will depend on the forms of values and associations of values supported in a particular system. An object-oriented system can represent entities directly by means of object values whereas a relational database system only supports simple values and generally entities of the application domain have to be represented through associations of values held as tuples of relations. Further, for a given system, the database designer may have a choice as to the form of representation of particular entities and relationships.

It follows that a database system is concerned with the management of a large number of values and associations between values. The general requirements of such a system can be expressed in terms of 'the four e's'. If a database system is to be *effective*, the management of values must be *effectual*, *expedient* and *efficient*.

In order that the management of values be effectual, the operation of the database system must be valid. This implies that the values of the database be correct, current and consistent. Expediency implies convenience for both a designer and a user of the

system whether that user be a database administrator, application programmer or end-user. For a system to be effective, it must also be efficient in that operations must be performed within an acceptable time period.

Note that these three requirements of database systems are not independent. Further, some features may contribute to more than one requirement. For example, support for data independence which divorces the logical and physical descriptions of data has benefits in terms of both user convenience and efficiency of both system operations and programmer productivity.

Each of these three requirements is now examined in some detail.

1.1.1 Effectual Data Management

If a database system is to satisfy its intended purpose, it is fundamental that it ensures the validity of the data values that it manages. This means that the system must protect the data against loss or corruption due either to failure or user abuse. Furthermore, the data values must be current in that the database must reflect the effects of successfully completed operations on the database. Thus, there must be some form of long-term, reliable storage of data values in order that the database will persist between application program executions and beyond machine shutdowns or failures.

For values to persist beyond program execution, there must be some form of persistent address space. In conventional systems, the logical unit of persistence supported by the operating system is the file and the persistent address space is controlled by the file manager.

Most general purpose programming language systems adopted this basic model of persistence by having a single form of persistent value - the file. Usually, a file corresponds to a sequence of uniformly formatted records. It was recognised that there are two major problems with such a system. Firstly, all data values which the programmer wishes to persist must be converted to an appropriate file format. Secondly, since the persistent data is effectively stored in raw format, there are no mechanisms to ensure that the data is used 'safely'.

To overcome these problems, there arose the notion of a persistent programming language in which any form of data value could persist. Two of the first persistent programming languages were developed independently in the East and West by Zamulin [Zam75, Zam78, Zam89] and Atkinson [ACC81, ABC+83], respectively.

It was estimated that of the order of one third of commercial application program code had been concerned with the mapping of program data structures to persistent

file structures and vice versa. Hence, the use of persistent programming languages for data-intensive applications reduced both programming effort and program code. Further, since these systems store type information along with the data values in the persistent store, they provide support for ensuring the safety of data use, i.e. that it is used in a correct and meaningful way.

To ensure the resilience of the system to machine failures, the persistent values must be held on non-volatile storage. Magnetic disks are the most widely used non-volatile medium for the storage of large quantities of current, updatable data values. We distinguish current data values from archive or back-up data which may be held on magnetic tape. Also, we distinguish between databases which may be updated from those that are a read-only information store and may be held on a read-only medium such as CD ROM. We note that with the development of systems with large non-volatile RAM, there are a number of research projects investigating main memory database systems in which all persistent values are held in RAM.

Therefore, typically the persistent values of a data-intensive application system will be held on magnetic disk. The persistent values associated with an application system are termed, collectively, the database of the system. In some persistent programming languages, it is possible to access more than one database in a program and therefore the persistent values of an application system may be stored as one or more databases. Those persistent values accessed by an application will be mapped into the program's virtual address space as required.

As stated previously, the system must ensure that the effects of successfully completed operations are reflected in the database. On the other hand, the system must prevent operations which fail to complete successfully from leaving the database in an inconsistent state. It is therefore necessary to introduce the notion of a logical unit of processing - the transaction. A transaction is an atomic operation on the database in that either it completes successfully and all of its effects are reflected in the database or it fails to complete and none of its effects are reflected in the database.

A successful transaction completes with a *commit* operation. All persistent values which have been created or updated in the transaction's virtual address space will be mapped into the persistent address space. If the machine were to fail while the persistent values are being written to the non-volatile storage, some of these updates could be lost and the database left in an inconsistent state. To prevent this, a log of the transaction's processing and outcome is maintained and, at the point of commit, this log is written onto stable storage. In the event of machine failure before the updates are written to the database, examination of the log record during recovery shows that the decision was taken to commit and the new database state can be constructed from the information in the log.

A transaction which fails to complete successfully will terminate with an *abort* operation. The decision to abort will be recorded in the log and the persistent values in

the transaction's virtual address space will not be mapped into the persistent address space. In this way, the database is unaffected by any processing performed by the transaction before abortion.

One of the early motivations for database systems was the separation of data from programs, thereby allowing a central data repository which could be shared by a number of application programs. This notion can be generalised to the separation of knowledge from the application of knowledge. Thus, there can be a shared repository of knowledge about some application domain and a particular application program can utilise whichever parts of this knowledge repository it requires and apply this knowledge in whatever way it chooses.

This generalisation is introduced here to emphasise that the same principles of persistence apply whatever forms of values may persist. Thus, some persistent programming languages, such as PS-algol [ACC81], [ABC+83] and Napier [MBCD89], have introduced procedures as first-class objects and one form of persistent value is the procedure. Similarly, a deductive database system is based on the logic programming paradigm in which knowledge about the application domain is recorded as a set of facts and a set of general rules. Therefore, the persistent store of a deductive database system must store both values which are facts and values which are rules. In object-oriented systems, an object has associated methods and the implementations of these methods must also persist.

Whatever the forms of persistent values supported by a system, it may be a requirement of the system that these values can be shared by a number of application programs. There must be some mechanism to ensure that two or more application programs accessing the same persistent store do not interfere with each other's operation. An application program may incorporate one or more transactions where a transaction is an atomic operation on the database. Hence, what is really required is some form of control to prevent conflict among concurrent transactions which operate on the same persistent store. This means that the effects on the database and the output of the transactions must be the same as if these transactions had run in isolation.

The system must support some form of concurrency control mechanism which either restricts access to the persistent values or restricts the commit operation of transactions in such a way that the effect of processing a set of transactions concurrently is equivalent to the effect of having processed these transactions one after the other in some specified order, i.e. the transactions are serializable.

Access to persistent values may be restricted by some form of locking scheme whereby a transaction may be delayed until it can obtain an appropriate lock on a persistent value (or the group of values that include the required value). Such a concurrency control scheme is termed pessimistic in that it prevents conflict at the cost of delaying the processing of transactions. An alternative optimistic approach is to detect conflict

when it arises and recover from it by the rather costly process of aborting transactions. This can be achieved by having transactions operate on their own local copies of persistent values and then checking before the commit operation if their operation is in conflict with that of any transactions that have committed during that transaction's processing. In the case of conflict, the commit operation is not allowed to proceed. In general, the pessimistic approach is preferred where the likelihood of conflict is high and the optimistic approach is preferred where the likelihood of conflict is low.

In summary, effectual data management requires some form of persistent store and transaction management. The transaction management should at least provide some form of recovery mechanism and, if the persistent store is to be sharable, then it must also support some form of concurrency control mechanism. Many variants of mechanisms for recovery and concurrency control have been proposed and here only a very brief description of their requirements has been provided. A good comprehensive description of the various issues and proposed mechanisms is given in the book by Bernstein, Hadzilacos and Goodman [BHG87].

In addition to the above, there should also be security mechanisms to prevent unauthorised access to data. The two basic forms of access control mechanisms are list-based or token-based. List-based mechanisms associate with a persistent value (or group of values), a list of authorised users and their access privileges. A token-based system permits access to a persistent value (or group of values) by users with the appropriate token. The second kind of access control is the basis of the capability systems [LS76].

Distributed systems require further extensions to the persistent store and transaction mechanisms. The persistent address space spans several nodes and a transaction might access persistent values across the nodes of this address space. Persistent values might be replicated at different nodes to increase the levels of locality of access and availability of data. Consequently, the recovery, concurrency control and security mechanisms have to be extended to take this into account. For example, some variant of the two-phase commit protocol might be used to ensure that the effects of a transaction are committed at all participating nodes or none of the participating nodes.

1.1.2 Expedient Data Management

A database is a representation of the corresponding application reality. A user of the database system can make enquiries about the application reality through this representation by the retrieval and processing of values of the database. An important part of the user activity is the maintenance of the representation in an effort to ensure that it is both a current and an accurate representation of the application reality.

To assist the users in their maintenance and retrieval activities, it is vital that they be presented with a conceptual model of the application reality. This conceptual model is a metalevel description in terms of the general concepts of interest in the application domain. It is the task of the database designer to construct the conceptual model.

The set of general concepts will be based upon the recognition of the existence of similarities among entities and their associations. An entity has a set of properties each of which may be either structural or operational. From now on, when we refer to the properties of an entity, it is assumed that we refer only to those properties of interest in a particular application system rather than all properties exhibited by the entity. A structural property is referred to as an attribute of the entity and it has an associated value. For example, an entity might have a name attribute with the value john. The set of structural properties, or attributes, of an entity determine the form of the entity. An operational property specifies an operation that the entity can perform. The set of operational properties of an entity characterises its behaviour.

If a set of entities have similar form and behaviour, then a general description of the properties of these entities may be introduced and these entities will have a common representation in the database. In this way, there is a move from the particular to the general and the introduction of a metalevel description of the representation of these entities referred to as a type. A type gives a general description of a value in terms of the properties that value must hold and we say that a value with those properties is an instance of that type.

A system may provide up to four basic kinds of types. Firstly, the system may support a number of primitive types for which the structural and operational properties are predefined. For example, commonly the primitive types **integer**, and **boolean** are available. Secondly, the system may support a number of structured types for which the operational properties are predefined but the structural properties are not. For example, record and enumerated types are structured types; they have fixed operations such as selectors on records - but the structure of the records is specified by the user. Thirdly, the system may support operational types in which the structure is fixed and operational properties are specified by the user. Examples of these are procedure or function types. Fourthly, the system may support abstract types for which both structural and operational properties have to be specified by the user. These are abstract data types or object types.

Note that just as an entity may be a particular of several general concepts, its representation value may have properties additional to those of a given type and it is therefore possible for a value to be considered as an instance of many types - provided that it has the properties of each of those types.

Similar entities may be classified according to their roles in the application domain. This classification is based on semantic groupings of entities which reflect not only the properties of those entities, but also their associations with other entities and external

operations on entities. An operation is external to an entity if it is not a property of that entity but rather a property of an encompassing entity. For example, in a library system the operation of borrowing a book may be viewed - not as an operation of a borrower or of a book - but rather as an operation of the encompassing library entity; then the borrow operation would be external to book entities. These external operations characterise the envisaged 'usage' of an entity. We adopt the term *category* to refer to such entity groupings.

The classification of entities is represented in the database by named collections of values representing the entity categories. A metalevel description of a collection is given by a *collection scheme* (cf. relation scheme) which specifies the name and nature of the collection and the type of its members. The nature of a collection determines whether a collection may contain more than one occurrence of any value and whether the member values are ordered; it corresponds to a particular form of bulk data type such as a set, ordered set or bag.

These categories of entities are not independent since they form part of a classification structure for the application domain. Classification structures are represented in the conceptual model by the specification of conceptual dependencies among the collections. Further, certain categories of entities will be related through the sorts of relationships in which their members may participate and this is also represented in the conceptual model by conceptual dependencies among collections.

A conceptual model may therefore be regarded as having a three-level structure. At the lowest level there are the general descriptions of individual entities of the application domain in terms of *types*. These entities are grouped together into categories according to their roles in the application domain and this is modelled in the conceptual model by *collection schemes* which describe the collections of values representing these categories. Finally, the categories are related to each other both through their participation in classification structures, and, in terms of the relationships that may exist between their members; this is described in the conceptual model as constraints on collections. This structuring of the basic notions of conceptual models is illustrated in figure 1.1.

The application reality contains person entities and house entities. These entities are grouped into categories of persons and homes, respectively. Then we wish to represent the relationships between persons and the homes they live in.

The application entities are modelled by the types `person` and `house` which give the properties exhibited by the corresponding entities. Categories are modelled by collection schemes. There are three collection schemes which specify: `Persons` is a collection of values of type `person`; `Homes` is a collection of values of type `house`; and, `Lives` is a collection that associates values of type `person` to values of type `house`. All three collections have set behaviour. The single constraint specifies that `Lives` maps members of collection `Persons` to members of collection `Homes`.

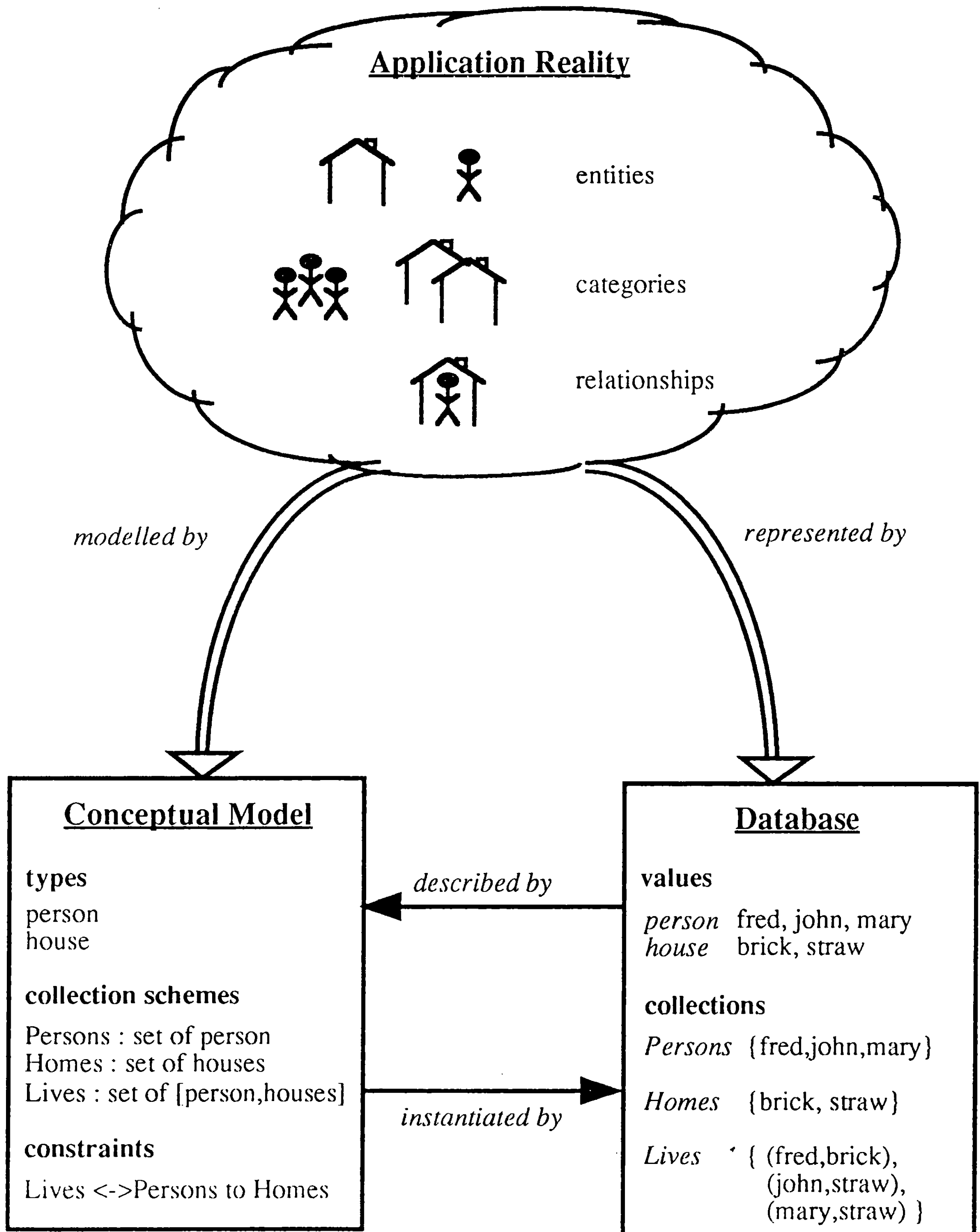

Figure 1.1: Conceptual Models and Databases

Then the database is an instantiation of the conceptual model in that it represents particular entities of the application reality, their roles and the relationships between them. Here it is assumed that a person entity is represented by a simple name value e.g. `john`, and similarly, for a house entity e.g. `brick`. Then the collection `Persons` is the set {`fred,john,mary`}, and the collection `Homes` is the set {`brick,straw`}. The relationship between persons and the homes they live in is represented by collection `Lives` which is a mapping from `Persons` to `Homes`. For example, `(fred,brick)` is a member of `Lives` and this represents the fact that fred lives in the brick house.

For a given database to be an instance of a conceptual model, the collections of the database must be as specified in the collection schemes and, must satisfy the constraints of the model.

The conceptual model is constructed by the database designer and is the basis for the use of the application system. The database designer specifies the conceptual model in terms of a conceptual modelling language. To assist the database designer in their task and also to ensure that the resulting conceptual model is understood by the users, this language must provide a number of general constructs that are adequate for the description of data-intensive application systems. The key to making such a language effective is that it should be simple: this means that it should be based on a small number of easily understood concepts and should be orthogonal in the application of these concepts.

Since the conceptual model imposes a structure on the values of the database, the conceptual modelling language is commonly referred to as a data modelling language - and the constructs on which it is based as a 'data model'. Thus, a data model could be regarded as a database designer's toolkit in that it provides the basic components for constructing a conceptual model. The term 'data model' is somewhat confusing since it is not a model but rather a theory for which models may be constructed. However, the term is in common usage and therefore will be adopted here. But we would like to emphasise the distinction that will be used between the terms 'data model' and 'conceptual model'. Here, we use 'data model' to mean the set of basic constructs and the term 'conceptual model' to mean the model of a particular application reality that is expressed in terms of these constructs. The conceptual model of a database is often referred as the database schema.

For expedient data management, it is therefore necessary to provide some means of representing a conceptual model for an application reality and the ability to view and manipulate the database in terms of this model. This means that the user should be able to specify operations not only on individual values - but also on collections of values and indeed on the entire database.

1.1.3 Efficient Data Management

An application system could be designed from scratch and thereby tailored for the intended applications to obtain optimal performance. However, in general, this would be a very expensive solution and, in the long term, may prove very inefficient as it does not cater for system evolution. A database system will evolve over time in terms of its use, its requirements, its structures and its values. This means that instead of 'hard-wiring' a system to specific database and application characteristics, it is desirable that the system is adaptable to change in such a way that good performance is still attained.

A database may contain very large collections of values and these may be part of a complex overall structure. Indeed, the values which represent individual entities of the application domain may themselves be large and/or complex. The size and complexity of the database can result in very high costs for retrieval operations. To be borne in mind is the fact that a so-called retrieval operation may not be a simple look-up operation - but may involve complex processing operations on values in the database. As databases are increasingly becoming an integral part of complex application systems, such as scientific processing and design systems, such processing operations can be expensive both in space and time.

Update operations may also be expensive not only in terms of the time to locate values required and any processing costs, but also, in terms of ensuring that the overall consistency of the database is maintained. Thus, an update operation may incur high overheads such as constraint checking activities.

If the system takes several minutes (or even hours) to perform the required operations, then effectively the system may be unusable. This is particularly true in transaction processing systems, such as airline reservation systems, that typically involve a large number of small transactions with a low critical response time.

Different applications may have conflicting requirements. Thus the location and/or representation of collections of values suited to one application may differ from that appropriate to another application. It may be possible to meet the requirements of both applications through replication of data or multiple representations but this then introduces additional overheads in ensuring the consistency of the database. It is important to remember then that the objective of the person responsible for the system must be the overall efficiency of the system rather than the optimal solution with respect to any one application.

A database management system is a generalised software system that should provide efficient data management for a range of application systems. It must provide methods of representing and processing data such that operational performance will reach

reasonable standards of efficiency regardless of specific data and application characteristics. This can be done through the provision of support for a small number of general constructs and various implementations of these constructs to suit data and application characteristics. Further, operations on a database should be specified at a logical level which is independent of physical representation and implementation: the system can then determine the methods of evaluation through consideration of the data characteristics and the current forms of representation. Hence, as the database system evolves, the underlying representations and implementations can be evolved in turn without recourse to the application programmers or end users.

The requirements for efficient data management are not orthogonal to those for expedient data management. Both require the notion of a data model which provides a small number of constructs in terms of which one can model the application domain with relative ease. Also, they both require the notion of a query language which enables the user to specify operations on the database at a logical level in terms of the conceptual model and independent of physical representation.

A given collection of values may have a number of possible representations. The most appropriate representation will depend on the size of the collection in terms of the number of values it contains, the characteristics of the member values and the envisaged operation characteristics. If a collection contains only a few members, then a very simple representation such as a linked list of values may suffice. If however a collection is large then a linear search to retrieve a particular value would be unacceptable and therefore some form of index structure would be maintained.

Typical index structures employ variants of hashing techniques or B-trees (or some combination of these). An index structure is built over some property or combination of properties of the values of the collection depending on access patterns. For example, in a collection of person objects, then if there are lots of selection operations based on the surname of a person then it is reasonable to construct an index of the collection based on the surname values. The choice of index structure can take into consideration the stability of the collection and also the characteristics of the properties over which an index is constructed. For example, some forms of index structure are easily expanded as the size of a collection grows, whereas others invoke significant overheads if the variation in the size of the collection is high. Further, the distribution of the property values is significant in the selection of an index structure.

It is, of course, possible to maintain several index structures over a single collection. But it is important to remember that the incurred overheads of maintaining an index structure can be high as the the index will have to be updated as values are updated. For this reason, clearly it is preferred to maintain index structures over relatively stable properties.

A database management system should provide a number of forms of index structure and representation of collections. Then the choice of maintaining a particular

index structure over a given collection may be made either by the database system administrator, who has overall responsibility for the system, or by the system itself. It is possible for a system to gather statistics on access patterns and, on the basis of these, decide when to construct a new index structure - or delete an existing one. At present, such systems are rare; however, in a number of systems an index may be constructed purely for the evaluation of a particular query.

The method of evaluation of a particular operation on a collection, or set of collections, depends upon the representation of those collections and also the characteristics of those collections. For example, if an operation is to select specific objects from a collection based on the values of some property of those objects, then the method of evaluation depends upon whether or not that collection has an index structure on that property. The size of a collection is also significant in selecting the implementation of an operation.

A given query on a database is specified in a query language in terms of the conceptual model. This query can be translated into an algebraic expression in terms of operations on collections. Then the selection of an evaluation plan for the query expression consists of two stages. First, the query expression is transformed into an equivalent expression based on the algebraic properties of the operations. This stage is generally known as logical optimisation. Then an evaluation plan is constructed for the transformed query expression and this plan takes into account the physical representation of the collections and also their characteristics. This stage is known as physical optimisation.

The issues of query optimisation are well understood in the area of relational database systems. These were the first systems to make a clear separation between the logical and physical levels of a database system such that the end-users interact with the database only in terms of its conceptual model (or views thereof) and the system deals with the translation of operations specified at the user level into operations at the physical level.

Unfortunately, it seems that in many of the object-oriented database systems this important separation of the logical and physical levels has been abandoned and the application programmers have to express their operations on the database in terms of physical representations and implementations. This apparent regression towards the navigational style of database application programming found in the early network and hierarchical database systems has now been recognised. A number of researchers and developers are attempting to separate out the logical and physical levels by supporting high-level query languages and multiple representations of structures and implementations of logical database operations.

1.2 The Collection Model

From the foregoing discussion on database system requirements, it should be clear that while a general persistent programming language supports effectual data management, it does not support all of the features for expedient and efficient data management required by data-intensive applications. A persistent programming system provides support for the persistence of individual values, but it does not provide explicit support for the notion of a database as a representation of an application domain in terms of interrelated collections of values. In other words, the persistent system has no notion of a data model as discussed in the section on expedient data management, and there is no distinction between the logical and physical levels of representation. Clearly, as they have no notion of a database, they also have no notion of a query language that expresses operations on a database.

It is important to emphasise that although some persistent systems have been extended to support collections, this still falls short of our database system requirements as there is no explicit support for expressing the sorts of conceptual dependencies among these collections required to model classification structures and relationships.

Further, it is important to emphasise that we are not presenting these as deficiencies of persistent systems but rather are highlighting the distinction between database systems and persistent systems. A database system is based on a persistent system - but it has additional facilities appropriate for the support of data-intensive applications. Indeed, a persistent programming language extended to support the notions of a data model and query language is a database programming language. For example, the database programming language Galileo [ACO85] is a persistent programming language with semantic data model features and abstraction mechanisms designed to support database application programming.

In this thesis, we address the questions of what these additional facilities should be and how they should be provided in the context of object-oriented systems. The proposed collection model is intended as a general model on which to base the provision of data management support in an object-oriented system. Although the model was developed in the context of a particular object-oriented platform, the general model is independent of the underlying persistent object system. In this way, the reported work differs from that of database programming languages such as Galilieo in that the data model is not tightly integrated with a specific programming language.

The collection model has two components - the structural and operational models. The structural model specifies the forms of collections supported and also the sorts of conceptual dependencies that can exist between collections. In other words, it specifies both the notion of collection and the notion of database as built on top of a persistent object system. Correspondingly, the operational model specifies operations on collections and operations on a database.

We separate out these two parts of the collection model to emphasise that while the operational model is dependent on the structural model, the structural model can be supported independently of the operational model. While we advocate the extensive use of high-level queries in application programming, the application programmer can choose to adopt the structural model as a means of modelling their application domain and then use basic iterators and navigational techniques to implement their applications directly. This incurs penalties in terms of supporting system evolution in that the application code is then dependent on the physical representation and may not be able to take advantage of new index structures. However, this may be appropriate to certain applications and certainly the use of the structural model alone is still beneficial.

The basic concepts supported in the structural model are entity categories, relationships between entities and rich classification structures both of entity categories and relationships. An entity category is represented by a collection of atomic values. These atomic values may be any values supported by the underlying type system and, in the case of objects, these will be object references. Entity relationships form relations between entity categories and these are represented by collections of pairs of atomic values. Thus a relationship between two objects will be represented by a pair consisting of the references of those objects.

Both entity categories and relations between categories can be part of classification structures. These classification structures allow entities to be considered as belonging to different roles in the application. For example, a person entity might at one time be considered as a staff entity, at another time as a lecturer entity, and at yet another time as a tennis player. There are conceptual dependencies between these roles to indicate, for example, that lecturer is a specialised role of staff and therefore every lecturer entity is also a staff entity. In a similar way, relations can also be specialised. For example, given relationships between persons and their associated departments, then the relationships between staff and the departments which employ them would be a specialisation of the more general association.

The structural model was given the name BROOM (Binary Relational Object-Oriented Model) to emphasise the importance of support for the direct representation of relationships between entities. Although very simple, the entity-relationship model [Che76] has proved very popular in the modelling of the structural properties of application domains. The basic concepts in this model are entity categories and relationships.

With the recent trends towards object-oriented data models, it was claimed that these can model both the dynamic and static properties of application domains and further that they also capture the high-level semantics. However, the lack of direct representation for relationships is now recognised as a major deficiency of object-oriented models. This can be seen to have a number of disadvantages as described by Rumbaugh in [Rum87]. Since relationships between objects are represented by

methods of these objects and therefore are effectively buried within objects, the overall structure of the application domain is not readily apparent. By decomposing relationships in this way, we cannot handle a relationship as a single logical unit. As Rumbaugh states:

> "... it is not possible to separate the abstraction from the implementation with the same clarity as the relational model."

Further, in the design of large systems, relationships have been shown to be a useful abstraction mechanism for partitioning systems into subsystems. Recently, there have been a number of proposals for some form of extension to object-oriented models to support relationships as first-class objects.

The semantic data models [HK87], [PM88] might be considered as a development of the entity-relationship models that support classification structures based on *isa* relationships between entity categories. Since one of the fundamental concepts of object-oriented data models is that of subtyping and inheritance, these are often considered to support classification structures. However, they often omit support for the rich conceptual dependencies that can arise in classification structures - such as categories partitioning other categories, the fact that certain categories are mutually exclusive and also the idea of supporting alternative classification views. These have been incorporated into the BROOM model through the concept of collection families.

The operational model is based on an algebra of collections. This mirrors the relational algebra which was fundamental to the success of the relational model. Its success was due to its simplicity, uniformity and high-level query languages stemming from the introduction of the single generic collection type - the relation. The basis for its high-level query languages was an algebra of operations on these collections as opposed to the notion of operations on individual data records that had underpinned the network and hierarchical data models. Unfortunately, the drawback of the relational model is that it is just too simple and lacks semantic structure.

Although, there have been some proposals for an algebra which operates on collections of objects, a number of object-oriented database systems use object at a time processing and have thereby lost the advances of the relational model in terms of its high-level query processing. A collection algebra can form the basis of high-level query languages for object-oriented systems and, importantly, the optimisation of query expressions and query evaluation strategies.

In effect, the operational model also supports operations on a database in that an operation involving one collection can generate operations on other collections as determined by the conceptual dependencies among collections. For example, deleting an object from one collection can propagate the deletion of that object from other

collections which are dependent on that collection. Thus, if we were to delete a particular student object from the collection `Persons` then we would also have to delete it from the collection `Students` if there is a dependency that every member of `Students` is also a member of `Persons`.

In summary, the collection model presented in this thesis incorporates many of the favourable features of the relational, entity-relationship and semantic data models. It has direct support for the representation of relationships; it supports rich classification structures; and it has an operational model based on an algebra of collections.

1.3 Structure of Thesis

This thesis presents a general data model which may form a foundation for the development of data management services in object-oriented systems. It assumes as a platform any reliable, persistent object store and considers the provision of expedient data management through a structural and an operational model that together form the particular data model referred to as the collection model. This model can then be used as a basis for efficient data management by means of multiple physical representation structures and query optimisation techniques.

Hence, the focus of this work is on the data modelling aspects of object-oriented database systems. In particular, there is an attempt to redress the apparent imbalance in many proposed object-oriented database systems where the emphasis has been on effectual and efficient data management and the issue of expediency has been somewhat neglected. As a result, many of the existing systems provide little support for the concepts that have become central to the work on data modelling. Further, by omitting support for the higher-level database structures of data models, the attention to efficiency addresses access to individual objects or single collections of objects. By supporting database structures involving multiple collections of objects, and operations on these collections, optimisations can be made at a higher-level which means that they tend to be more global and less localised. At this higher-level, the optimisation techniques are better able to utilise semantic information of the application domain.

We begin in Chapter 2 with a discussion on the foundations of data models. A data model determines the basic constructs available for the construction of conceptual models of application domains. We therefore examine the general philosophical foundations of conceptual modelling as a basis for determining the basic requirements of data models. From these requirements, we present a general framework in which to consider the main characteristics of the various categories of data models. The chapter is concluded with a discussion of the data modelling support provided by existing object database management systems.

Chapter 3 deals with the structural aspects of the collection model. The specification of the BROOM model is presented in four stages. Firstly, there is an informal overview which describes the main features of the model and looks at some simple examples. Next, the fundamental concepts on which the model is built namely, collections and collection families, are presented in detail. This is followed by a meta-circular description of the BROOM model in which the model is described in terms of itself. This description is used as an intermediate stage of specification which is refined into a formal specification in the language Z [Spi89], [Dil90], [PST91]. Such a meta-circular description is also useful both as a documentation aid for the model and as a basis for supporting the uniform treatment of data and metadata.

The semantic modelling capabilities of the BROOM model are examined in Chapter 4. First the support for relationships is discussed in detail. Then each of the semantic data modelling abstractions referred to as aggregation, generalisation and association is examined with examples to demonstrate how these would be represented in the BROOM model.

The operational aspects of the collection model are presented in Chapter 5. Three levels of operation are possible and the chapter begins with an examination of these levels. The main theme of the chapter is the presentation of a collection algebra which deals with operations on collections. The properties of the algebra are presented and a discussion of how the associated algebraic transformations could be used in query optimisation.

A database is not a static entity but rather is dynamic in that it evolves over time. The entities represented will change and also the forms of their representations may change as entities adopt different roles throughout their lifetime. In addition, the structure of the database may evolve either to reflect changes in the real world systems that they model or because of changes to the requirements of the database system. In Chapter 6, we discuss the various forms of database evolution and how these can be supported. In particular, we propose an extension to the collection model to support object evolution.

The collection model was developed within the Comandos project [CBHdP93]. Comandos is an Esprit project concerned with the construction and management of distributed open systems. In Chapter 7, we describe how the collection model was realised as part of a Comandos system. The collection model was designed as a general model and is not specific to the Comandos system. To illustrate this point, we also describe a prototype object data management system, COLLEEN, which was based on the collection model and implemented in MacProlog [LPA91].

Chapter 8 concludes with a summary of the contributions of this work and some discussion of on-going and future work based on the collection model.

Chapter 2

Foundations of Data Models

The collection model proposed in this thesis is a particular data model which primarily was designed to support data management in object-oriented systems. Before going on to present this model, we first consider in some detail exactly what a data model is and what its requirements are, both in general, and also in the specific context of object-oriented systems.

A data model supports the construction of a model of a data-intensive application system with the intention of representing that application domain by means of a database system. The process of constructing an application model using a particular data model is referred to as data modelling. We term the constructed model a conceptual model of the application domain. Such a conceptual model should be *adequate* in that it should capture the relevant features of the application domain, and, furthermore, it should be *natural* in that it should correspond to the sorts of mental models that users construct for mental processing.

The general area of study concerned with the construction of models which correspond directly and naturally to our own conceptualisations of reality is known as either conceptual modelling or cognitive modelling. The process of data modelling is a special case of conceptual modelling and it follows that the foundations of conceptual modelling are an important starting point in an attempt to determine the general requirements of data models. Therefore, this chapter begins with an examination of some of the philosophical foundations of conceptual modelling that are particularly pertinent to database systems and give some insight into the underlying basis for the proposed collection model.

Arising from these philosophical considerations, we arrive at some requirements for data models that in turn form the basis of a general framework for data models. We present this framework in section 2.2 and then go on to consider the various categories of existing data models in terms of this framework in section 2.3.

The chapter concludes with a section that discusses the data modelling support provided in a number of existing object-oriented database systems.

2.1 Philosophical Foundations

A conceptual model of an application reality is a general description of the sorts of things that exist in that part of the real world with which the application is concerned. The purpose of such a model is to provide a basis for communication about that application reality between the user and the application system. It is through this model that the user ascertains the nature of the information stored in the system and indeed possibly the nature of the reality itself. Further, it is in terms of the conceptual model that the users specify information to be retrieved and processing to be performed. Additionally, these conceptual models communicate to the database management system the designer's requirements of the application system regarding information to be managed and consistency constraints.

Given that these conceptual models form the basis of user assimilation and communication, it is desirable that they conform to the cognitive models of the human mind. While the precise nature of mental models is still open to much debate and research, there are some general principles that have been proposed on the basis of various psychological and linguistic studies. To examine the philosophical foundations of conceptual modelling, it is therefore important to consider studies of the philosophy of mind and the philosophy of language. In turn, since conceptual modelling is concerned with the construction of models of reality, we must address the question: "What kinds of things exist in reality?". This is one of the fundamental questions of metaphysics and is the concern of that part of philosophy known as ontology.

A full account of these topics is beyond the scope of this section: indeed they form a significant part of an entire field of study known as Cognitive Science [SFG+91]. Rather we discuss some general principles that provide a useful insight into the nature of data models and their requirements.

Our starting point is the claim that the process of abstraction is fundamental to human information processing. In particular, there is strong evidence to support the claim that an abstraction mechanism based on *generalisation* underlies cognitive models. An excellent introduction to this area and an overview of the psychological evidence to support this view of cognitive models is given by Sowa in [Sow84].

In natural language, general concepts are referred to by general terms. Early in its development, a child learns to use general terms such as `man` in addition to particular terms referring to particular entities such as `john`. This is an important step as the child moves from the particular to the general as it recognises the similarity among

certain groups of entities. The introduction of a general concept allows the child to make statements not only about particular entities within its realm of experience but also predictive statements about entities that may exist beyond its realm of experience. Of course, a child may make mistakes in its learning of general concepts and have to backtrack and adapt its notion of a certain general concept. However, it is amazing how quickly and easily most children make this transition to the use of general concepts.

Many movements of contemporary art have attempted to capture this process of abstraction through generalisation. Abstract art is intended to provide visual representations of general concepts whether these be generalisations of particular visual objects such as faces or of something inherently non-visual such as emotions. (See [Atk90] for a discussion of abstraction and its role in contemporary art.)

The generalisation process does not stop there. It is further recognised that general concepts may themselves be generalised. Thus the concepts `man` and `woman` can be generalised to the concept `person` by recognising the similarities and abstracting away the differences.

Another form of generalisation is the recognition of associations among general concepts. For example, if a child has a general concept of `person` and a general concept of `shop`, then they will learn that persons work in shops and persons are customers of shops. Further, they will learn the general concept of a transaction of buying an article in a shop.

In this way, we make 'order out of chaos' by classifying entities by means of general concepts. Note that the structures may not arise in a purely 'bottom-up' manner. In fact, in the world of general concepts, it is common to form a classification structure through the specialisation of general concepts. For example, as a child begins to recognise that some persons are referred to as 'he' and some as 'she', then it specialises the general concept `person` into the general concepts `man` and `woman`.

The resulting classification structures may not be strictly hierarchical as in the case of biological taxonomies. For example, given the general concepts `house` and `boat`, then we might recognise that there are entities which are both a house and a boat and therefore introduce the general concept `houseboat` which is a specialisation of both. In general, a classification structure will form a directed acyclic graph.

It is also usual for us to classify entities in different ways depending upon the context. For example, the general concept of `person` might be specialised according to sex, race, nationality, profession or religion on different occasions.

The basis for a general concept may or may not be well-defined. We all have a good intuition of what we mean by the general concept `person` - but how could we define it? It would be possible to give some form of precise biological definition in terms

of cell structure. However, few of us have knowledge of this precise definition - and we tend to think in terms of some looser definition which might be considered as a working approximation to the precise definition.

Other general concepts might be regarded as purely definitional in that they do not belong to 'natural' classifications. In other words, they are not classified by the natural or mathematical sciences in terms of biological, chemical, physical or mathematical properties but rather by definitions introduced by humans. For example, `bachelor` was first introduced to denote the general concept of a man who is not married. It is sometimes the case that such definitions may be changed for the sake of convenience. For example, in some contexts, the definition of `bachelor` has now been amended to include females who are not married. This contrasts with amendments (rather than refinements) to classification structures of the natural and mathematical sciences which tend to arise for reasons of correction rather than convenience.

It might be argued that sociological definitions are no less 'natural' than, say, a biological definition. However, it is useful to distinguish between general concepts that are introduced for convenience and are purely definitional from those that arise 'naturally' and for which we might struggle to find a definition. This distinction between precision and convenience arises in other ways. Even though a general concept may have a precise definition and that definition is well-known to us, it may be the case that at certain times we substitute a looser, more convenient definition. For example, we may know when questioned that a tomato is a fruit rather than a vegetable, but when we use the general term vegetable to describe our shopping, we may well include tomatoes in this category.

Socrates considered that the acquisition of knowledge was dependent upon the attainment of definitions of general concepts. He was particularly concerned with the definitions of ethical concepts such as justice and virtue. Plato and Aristotle attempted to determine the nature of these general concepts and exactly what forms these definitions should take. Plato proposed the existence of an abstract world of Ideas or Forms which would provide perfect exemplars for general concepts and would exist independently of the particulars in the concrete world. Aristotle rejected Plato's notion of an independent abstract world. He introduced a scheme for categorising objects in terms of genus and differentia based on a set of indivisible primitives. Aristotle's proposal for the categorisation of natural phenomena formed the basis for science until the 17th century and still influences thinking in cognitive science.

Over the centuries philosophers have continued to address Socrates' quest to define general concepts. There have been numerous proposals belonging to the many schools of thought and an overview of these can be found in books such as [Bec88], [Arm89], and [Sta72]. We mention only a few below.

The distinction between token and type was first introduced by Peirce in the 19th century in his work on semantics and this terminology has been widely accepted.

A type embodies a concept whereas a token is an instance of a concept. Then the question that must be addressed is : What is a type?

The notions of type proposed by Russell [Rus56] and Ryle [Ryl49] differed in terms of whether type was an extensional or intensional notion, respectively. Thus, Russell regarded a type with respect to a given predicate as the set of entities spanned by that predicate. Ryle, on the other hand, regarded a type as a set of predicates which spanned a given set of entities. But what these (and other variants described by Sommer in [Som67]) have in common is their regard for 'type' as defining necessary and sufficient conditions for membership of a set whether that set is a set of entities or a set of predicates. Russell's type for a given predicate P was the set of all entities that were spanned by P. So membership of the type was solely determined by P. Ryle's type for a given set S was the set of all predicates that spanned the members of the set S and no other entities. So membership of the type was solely determined by the set S.

Many studies of meaning or semantics by philosophers and linguists have assumed that general concepts can be defined by types in this way. But recent developments in psychology [Ros75] and linguistics [Lak87], as well as in philosophy [Wit53], have challenged the view that most of our concepts are grounded in the kind of definitions Socrates sought.

The notion of type as a means of classifying entities by a set of necessary and sufficient conditions was questioned by Wittgenstein [Wit53]. He claimed that there are general concepts for which there might be no precise definition. Wittgenstein discussed this in terms of the general concept of 'game'. He claimed that there is no precise definition of game: this means that we cannot give a single set of distinguishing properties exhibited by all entities which we would classify as being games.

In recent decades, this argument has been taken further by the work of a number of philosophers - notably that of Putnam [Put77a], [Put77b], Kripke [Kri77] and Quine [Qui77] in their challenges to the traditional theory of meaning.

We therefore have two quite distinct views. Some philosophers consider general concepts as being defined in terms of a type which gives necessary and sufficient conditions for determining whether or not a particular general concept applies to a particular entity. Others consider that a general concept is associated with a group of entities but that it may not be possible to state necessary and sufficient conditions to determine membership of this group.

Both of these views can be encountered in the realms of Computer Science and Artificial Intelligence. In programming languages, a type may be considered as either an intensional or extensional notion but in either case is associated with a set of defining properties. These properties specify the form that a value may take. For example, a record type describes the structure to which all values of that type will

conform. In Artificial Intelligence, we often encounter a different notion of type which gives typical properties of entities associated with a general concept, but not all entities associated with that concept have to exhibit all of these properties. Such a notion of type corresponds to the notion of prototype and the classification of entities may be based on the idea of family resemblances: entities are associated with general concepts because they have a greater degree of similarity to other entities associated with that concept than to those entities associated with other concepts.

The question then is which view should prevail in database systems. The answer to this question is both views. For it seems that in database systems there are two forms of classification that are sitting side by side.

Like knowledge representation techniques in Artificial Intelligence, data modelling is concerned with the construction of conceptual models of reality. It is therefore involved with the description of general concepts of the application domain. In accordance with a number of contemporary philosophers, linguists and psychologists, we consider that it may not be possible to define these concepts in terms of a set of necessary and sufficient conditions. Further, as discussed previously, even if a defining set of properties can be found, this may not be appropriate. For example, it might be possible to determine the membership of a group `Males` by examination of the genetic code of individuals. However, this information may not be available, or indeed of interest, to the person doing the classifying in an application system.

On the other hand, a database schema is considered not only as a description of reality but also as a description of the representation of that reality in the database. When we describe representations, then we are in line with the programming language view of types in that we are defining the forms that values may take. Thus by the means of types we impose a classification on values.

We therefore distinguish these two forms of description. The former corresponds to classifying entities into categories according to general concepts of the application domain. The latter corresponds to describing the representation of entities within the database and this is done in terms of types which define the forms of these representations in terms of a set of structural and behavioural properties.

As a result, we separate out the notions of typing and classification in data models. Clearly, the two are linked in that similar entities will have similar representations. Thus an entity category will have an associated type that describes the form of representation of its members. In general, we will adopt the convention of using names starting will a lower case letter to denote types and names starting with an upper case letter to denote categories. Then we might have a category `Persons` and the representation of the entities belonging to that category is given by the type `person`.

It is possible for two or more categories to have the same associated type for their

members. For example, entities in the categories `Persons`, `Friends` and `Enemies` might all have the same representation as specified by the type `person`. Then the membership of these categories may be solely determined by the user who will assign an entity to the appropriate categories at the time that the representation of that entity is created in the database.

In summary, we feel that it is important to recognise that the data modeller is at one and the same time constructing a model both of reality and of the database that will represent that reality. It is therefore essential to distinguish these two activities by separating out the notions of typing and classification. This issue is particularly important in the context of database programming languages where the notions of type in programming languages and type in database systems must coexist.

2.2 A General Framework

We describe the world in terms of entities, entity categories and relations. Entities have properties which may be either structural or behavioural. A structural property describes a value associated with an entity at some given point in time and such a property is referred to as an attribute. A behavioural property specifies some sort of action that an entity can perform. An entity category is a 'natural' grouping of entities according to their external characteristics. These external characteristics are described in terms of the actions which can be performed on the entities and the relations in which they may participate.

Relations may be on entities or on entity categories - or indeed they may be higher-order in that they are on relations. A relation on entities corresponds to what is often referred to as user-defined relationships among entities. The fact that one category may be a refinement of another category is a relation on categories. Similarly, the fact that one relation may be considered as a refinement of another relation yields a relation on relations.

A conceptual model of an application reality should be capable of modelling these various kinds of concepts. This means that a data model should provide constructs that can be used to represent each of the above notions. There are lots of ways in which these concepts can be represented and this is reflected in the large variety of data models that have been proposed. Can any general statements be made about how these concepts can be represented and the data model constructs to support them?

In figure 2.1, we present a view of the representation of the three general concepts - entity, category and relation - and the corresponding data model constructs required to support these concepts.

concept	representation	construct
entities	*values*	value
categories	*collections of values*	collection
relations	*associations of values and collections*	database

Figure 2.1: Representation of Concepts

An entity will be represented by a denotable value of the database. Its representation may not be a single atomic value but rather a complex value such as an object or record value. If a particular data model does not provide constructs for complex values, then a complex entity may have to be represented in terms of related simpler entities.

A category is a grouping of entities and therefore its representation will involve a grouping of the representations of the member entities. So a representation of a category would be some form of collection of values. Such a collection may or may not be itself a value. If it is considered a value, then one could have collections of collections and so on. Similarly, it might be possible to have a collection value as a component of a record value in which case a complex entity could be represented say by a record which contained a collection. One of the main ways of characterising a data model is in terms of the forms of nested values that can be supported.

The representation of a relation on entities must involve some way of associating entities that are related. In other words, its representation must be some form of mapping between the values representing entities. Such associations may also be represented by collections of values. For example, binary associations could be represented by a collection of pair values.

A relation on categories must have some form of representation that associates the collections that represent those categories. This requires some higher-level data model construct that facilitates the representation of the dependencies between categories expressed by the relation.

Even a relation on entities will involve some form of dependency between collections since relations on entities do not have an independent existence. How can we relate one entity to another if these entities themselves do not exist? It is common that a

relation on entities will include in its representation some indication of the categories to which these entities must belong. Thus the representation of a relation requires not only associations of values - but also an association of the collections to which these values belong. This linking together of collections into an overall, possibly complex, structure that represents the application reality is indeed our notion of a database.

Hence, in very general terms, we can state that a data model should have at least three levels of construct. Firstly, it requires constructs that specify the forms of denotable values that may be used to represent individual entities. Secondly, it requires constructs that specify the forms of collections that may be used to represent categories. And, thirdly, it requires constructs that can be used to represent relations and thereby specify how a database may be built out of collections of values.

So far we have concentrated on the structural aspects of the representation of the application reality. But a database system is dynamic: it must support update, retrieval and processing activities. The database must reflect the changes in reality. In addition, the application system must be capable of answering users' enquiries about the reality that the system represents. Such update, retrieval and processing activities are the dynamic aspects of a database system and they model the behaviour of the application reality in terms of the various actions that can be performed.

These actions are represented by operations of the database system. Operations can be on values, collections of values or on the whole database. An operation on a value can represent an action undertaken by an entity or an action applied to an individual entity. An operation on a collection of values can represent an action applied to a category: it may construct the representation of a category from the existing representations of other categories. An operation on a database may construct one database representation from another and may reflect a change in reality. Therefore just as there are three levels of structure, there are three levels of operation as indicated in figure 2.2.

Clearly, the available levels of operation are dependent upon the available levels of structure. If a particular data model had no construct to represent a database as interdependent collections, then it would be meaningless to have operations defined on a database. A data model which has constructs for the three levels of structure may have operations at all three levels or it may only have operations at the value level or at the collection and value levels.

Our general framework for the characterisation of data models therefore describes a data model in terms of the three levels of structure and the three levels of operation.

Before going on to examine some of the more prevalent data models in terms of this general framework, it is important to say something about how the collection model fits into this framework.

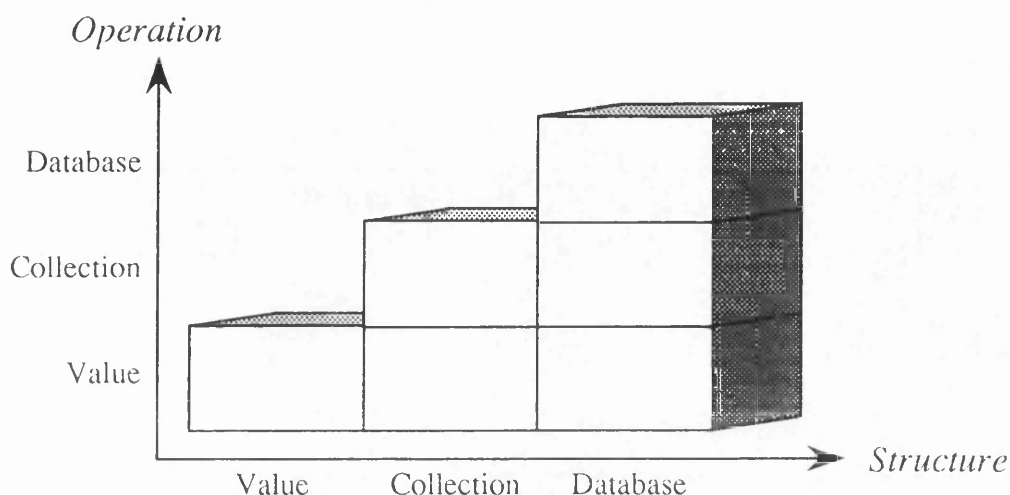

Figure 2.2: Three Levels of Structure and Operation

The collection model is intended to be a general collection model that is not dependent on a particular underlying type system. This means that it does not assume anything about the forms of denotable values of a database. This is not unlike many descriptions of the relational model which start by assuming that certain base types (domains) are supported, but they do not concern themselves with the precise nature of these values. The collection model therefore resides above a type model.

A data model has an associated constraint model which is the basis for the representation of conceptual dependencies that can exist between properties, entities, categories and relations. Now these constraints are sometimes categorised as inherent, implicit or explicit (see [TL82] for a discussion of this). Inherent constraints are those that are built into the data model and as such cannot be violated. Here, we will use the term implicit constraint to mean those that are expressed in terms of the basic structures of the data model. Finally, there may be explicit constraints which are any general form of constraint - structural or dynamic - that may be placed on the database. We will consider only inherent and implicit constraints as we feel that these are strictly part of a data model. But any data model may be extended with a general constraint model to facilitate the representation of all sorts of conceptual dependencies.

The placement of our collection model is illustrated in figure 2.3. It is situated on top of an underlying type model which specifies the forms of values and operations on these values. In turn, the collection model could be extended with a general constraint model that might for example use a rule-based language to express various forms of explicit constraints.

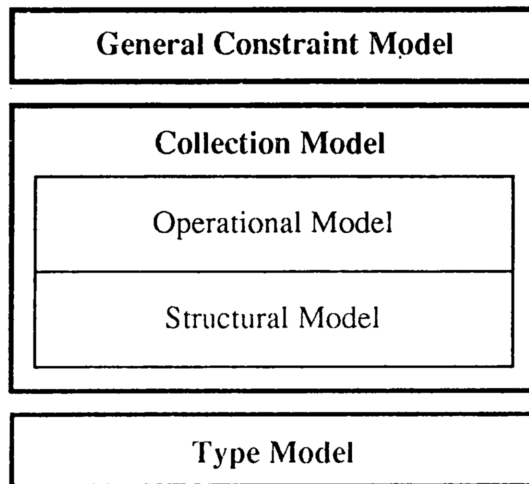

Figure 2.3: The Collection Model

2.3 Data Models

During the last twenty five years, a large variety of data models have been proposed. These vary not only in their form but also in their intended purpose. While many of them have been proposed as an underlying model for the implementation of a database management system, others have been proposed as a conceptual modelling tool with the intention that a description of the application domain first be produced in terms of that data model, and then, translated into the data model of the database management system in which the application system will be realised.

However, this distinction is not always a clear one as sometimes a data model intended for use as a conceptual modelling tool is later realised by a database management system. Then an operational part of the model has to be specified. For the data models that we shall mention, this means that some of them only have a well-defined structural part to the model as the operational part was either never defined or was not defined at the time of conception.

Here we only present the main features of a number of data models with respect to our general framework. For a more complete discussion on data models, we recommend two sources. The first is the book by Tsichritzis and Lochovsky [TL82] which presents the requirements and features of data models in detail and then examines a wide range of data models. The second is a paper by Schek and Scholl [SS91] which discusses the evolutionary paths that have led to object data models.

Network Data Models

Early database management systems were based on the network data model [COD71], [TF76], or a restricted version of it known as the hierarchical data model [McG77]. These models tend to reflect the physical structures of data and are therefore based on the idea of files of records with links between them. Record values are supported and an entity category is represented by a set of records of a given type. The only forms of relations supported are those of relationships between entities and they are represented by what is known as DBTG-sets. The representation is usually a form of linked list structure and this is reflected at the conceptual level in terms of the restrictions imposed on relationships that can be modelled by DBTG-sets. Relationships have to be functional and further they can not be recursive in that they can not link records of the same type.

Operations are specified at the level of records and a navigational style of programming is supported. Therefore, both the structural and operational parts of the network model are criticised for being at too low a level and they require the application programmer to be aware of the physical representation structures and to keep track of their navigation through these structures.

Relational Data Models

Although implementations of the relational model were already in existence, the formal introduction of the model is usually attributed to Codd in his paper [Cod70].

A relation is a collection of tuples. Effectively a tuple is a record since it consists of a set of labelled values. It is commonly considered that the values of a tuple must be atomic: they should be simple values of some domain which is defined as a restriction of base values such as integer and string. This condition is known as the first normal form of relations. It is interesting to note that in his exposition of the relational model in [Cod70], Codd does not state this as a basic restriction of the model but rather introduces it as a secondary restriction that might be appropriate in certain cases. However, the first normal form condition is now considered to be part of the standard definition of the relational model.

The structural part of the relational model represents entities as tuples of atomic values and entity categories as relations which are sets of tuples. A relational database is a group of collections of tuples but these collections do not have any explicit links between them. In other words, the relational model does not support any form of conceptual dependency among relations. The relational model does not therefore support our notion of relations as associations between entities and categories of entities. Relationships between entities are modelled in the relational model by tuples of values. The fact that the tuple of values representing a relationship can be used to

establish an association between two or more entities is not modelled by the schema but is expressed by user-defined operations.

The relational model was one of the first data models to have operations at the level of collections and these are given in terms of a relational algebra. The join operator of this algebra can be used to establish associations between different collections of values.

With respect to our general framework, the basic relational model can be considered as supporting the two lower levels of both structure and operation. The fact that the relations are not linked together into an overall structure means that there is no notion of a single complex structure which models the application domain. Then there cannot be any operations at the level of a database. It is the responsibility of the end-user or application programmer to realise this third level by means of queries and application code.

The main criticisms that have been levelled at the relational model concern its lack of semantic modelling capability. If we consider the relational model in terms of our general framework, we can immediately identify two directions of development of the model to address this issue. The first is the need to support the notion of a database in terms of the ability to express conceptual dependencies among collections, and the second is to extend the limited forms of values supported.

Much of the work on data dependencies can be viewed as an attempt to extend the relational model to support the notion of a database structure by the expression of conceptual dependencies among relations. The notion of keys was introduced to provide some form of entity reference. The cross-referencing of keys is the means of linking relations together into a database. The ability to express conceptual dependencies among collections means that the third level of structure, the database, is supported. Then the third level of operation follows with the use of concepts such as referential integrity to define the semantics of insertion and deletion operations on a database. For example, the deletion of a tuple from one relation could result in the deletion of tuples in other relations to ensure the consistency of the database as expressed by the data dependencies.

The relational model was extended to support further forms of conceptual dependencies so that not only relations between entities but also relations between categories could be modelled. RM/T [Cod79] extended the relational data model with various metadata relations that described these dependencies. This model distinguishes between relations which represent entity categories from those that represent relationships between entities; it also supports classification structures in which one entity category may be a generalisation of another entity category.

Proposals for nested relational models remove the first normal form restriction that the only form of value is the tuple of atomic values and allow relations as values of

attributes. This can be thought of as removing the restriction that the tuple and set constructors can only be applied once and in a specific order - "build tuples and then from these build relations". By allowing the tuple and set constructors to be used repeatedly, we arrive at nested relations. If we further relax the restriction that set and tuple constructors have to strictly alternate, then we end up with what is usually called complex objects [AB88] or extended relations [PT86]. As stated by Schek and Scholl in their paper which examines the evolution of object data models [SS91]:

> "Essentially nested relations or complex objects do not really extend the data model with new concepts, they just permit a more deliberate use of the type constructors already available. Hence rather than just 'flat files' we can now model quite complex hierarchical structures."

A number of proposals exist for extending the operational part of the relational model to deal with complex values: these include [JS82], [AB84],[SS86],[PT86], [RKB87] and [RKS88].

Graph Data Models

We consider as graph data models those which firmly place the emphasis on relationships rather than entities. With respect to our general framework, the values tend to be very simple and may include only atomic values: the complexity of the structure comes from the links between these values. Examples of such models include the binary relational model [Abr74], the functional data model [Shi81], and the recent graph-based model of Levene [LP91].

The Semantic Binary Relational model of Abrial was influenced by semantic nets used for knowledge representation in Artificial Intelligence [Qui68], [Rap68]. A semantic net is a labelled, directed graph and the nodes are atomic entities which may refer to either tokens or types. Arcs between nodes can therefore represent relationships between particulars, relationships between particulars and concepts, or, relationships between concepts. In this way, the semantic nets combine data and metadata.

Data models distinguish between data and metadata and only metadata appears in the schema. Then the graph data models tend to be based on graphs in which the nodes represent concepts and the links represent relationships between concepts. The labels on links are either specially designated labels or user-specified names of relationships. Specially designated labels might include *isa* or *part-of* and these correspond to special system-supported relationships between categories.

The values supported tend to be only simple values such as string or integer and some notion of entity identity. An entity category corresponds to a concept node of the

graph and is the set of all instances in the database associated with the concept. The graph represents the database with the edges representing both relations between entities and entity categories. Therefore the graph models support all three levels of our general framework. The operational components of these models vary enormously and include both functional [Shi81] and rule-based styles [PL93].

The general criticism aimed at these models arises from the simple forms of values supported. Entities can not be represented as complex values which means that all the properties and component parts of an entity have to be represented as relationships between primitive entities. The resulting graph structures can be very complex. One way of addressing this issue is to support nested graph structures such that the node of a graph can itself be a graph [LP91], [PL93].

Semantic Data Models

The semantic data models are those which were designed specifically to support the conceptual modelling process. They can be considered as having two origins: the Entity-Relationship model of Chen [Che76] and, like the graph data models, the semantic nets.

The Entity-Relationship model was proposed as a database design tool rather than a basis for the implementation of database management systems and therefore the original proposal had no associated operational model. The model brought to the fore the idea of there being two basic notions - entities and relationships between entities.

The semantic data models which developed from semantic nets took on board the abstractions which were the basis for system-supported relationships between entity categories - the *isa* and *part-of* relationships. Smith and Smith [SS77] introduced these ideas into relational models. From these origins a number of semantic data models have evolved with perhaps the best known being SDM [HM81] and TAXIS [MBW80]. In these models, entities are represented by complex values and relationships between entities are viewed as attributes of the entities. Thus, a relationship is decomposed into two directional, functional components; a dependency between the two component parts may be declared in order that consistency is ensured.

These semantic data models therefore tend to focus on entities rather than relationships. This imbalance is to some extent redressed by the enhanced entity-relationship models. e.g. [EWH85], [SSW80],[TYF86]. These combine both approaches by incorporating support for rich classification structures into the Entity-Relationship model.

The semantic models support all three levels of structure in our general framework. However, with all of these models, the emphasis tends to lie with the structural

component rather than the operational component; we therefore omit any discussion of the operational aspects.

Sometimes the graph data models are also classified as semantic data models due to their emphasis on conceptual modelling and data semantics. However, we have separated them in this section to highlight the distinction between models which adopt the entity-based view of the world from those that adopt the relationship-based view. However, when models reach a more balanced position in which entities and relationships are placed at the same level of abstraction, then clearly, the distinction between the two approaches blurs. In the remainder of this thesis, we will extend our notion of semantic data models to include the graph-based data models.

Two papers provide good surveys in the area of semantic data models. Peckham and Maryanski [PM88] give a comparative survey of a number of semantic data models while Hull and King [HK87] examine semantic data modelling concepts through the introduction of a generic semantic data model based on IFO [AH87].

Object Data Models

As with some of the other categories of data models that we have discussed, there is no agreed characterisation of object data models. For some, the roots of object data models clearly lie in object-oriented programming and an object data model is considered as supporting the fundamental notions of object-oriented programming, i.e. encapsulation, inheritance and message passing. Then an object has dynamic behaviour which is modelled by the methods of that object. In an alternative view, object data models are characterised by the fact that they are entity-based rather than value-based and support inheritance of attributes through *isa* relationships. If this view is adopted, then most of the semantic data models would be classified as object data models.

The two views expressed above are not conflicting since an object data model could exhibit both sets of properties. The view adopted tends to reflect the context of the model and, possibly, the background of the person holding the view. As stated, the former view has its roots in object-oriented programming while the latter has its roots in data modelling. The object data models underpinning most of the existing object-oriented database management systems tend to belong to the former view. On the other hand, there have been many proposals for generic object data models which are intended for conceptual design and these tend to adopt the latter view in that they have their basis in semantic data models and they may or may not deal with dynamic behaviour. These models include [SS90], [SLR+92], [BK91] and [MD91]

We prefer to take a general view of object data models and consider their characterising property to be the fact that they are entity-based rather than value-based. Typically, they will support classification structures and inheritance mechanisms.

They may or may not support the modelling of behaviour by methods: in the case that they do, then we refer to them as object-oriented data models.

In most of the the proposed object data models, the notion of entity predominates and relationships are decomposed and represented as properties of entities. As we will discuss at length later, this is now considered a weakness of these models and recently there has been a number of proposals for object data models with direct support for the representation of relationships: these include [AGO91],[Bra90],[DG90], [NQZ90] and [Rum87].

The various object data models differ greatly in terms of the supported levels of structure and operation.

2.4 Object-Oriented Database Management Systems

We now turn to consider a number of well-known object-oriented database management systems and examine the data modelling support in these systems. Our claim that many of these systems are limited in their data modelling support is reflected in the fact that it is often difficult to obtain a clear description of their data modelling capabilities in the literature where the emphasis tends to be on the effectual and efficient aspects of data management. There are three general sources of information on object-oriented database management systems [Cat91b], [ZM90], [Cat91a].

We have chosen to classify these systems according to three categories. While the boundaries between these categories is not rigid and one might argue about the classification of some of these systems, it is felt that they are useful in placing the emphasis of systems. The first is the category of systems that are solidly based in the C++ culture [Str87]. The second category is that of systems which were developed with object-oriented database management in mind and then a particular object-oriented language (or languages) was selected as a platform. The third category is that of systems based on extensions to relational technologies: such systems may not be classified as true object-oriented systems but it is useful to include them for purposes of comparison.

Systems based on C++

A number of commercially developed systems fit into this category. The impression is that they have been developed with C++ as the starting point with the intention of supporting database application programming in C++. They have extended

the language by means of library and preprocessor technology to support notions of persistence, transactions and efficient storage of and access to bulk structures.

One of the first of these was ONTOS [Ont91] of Ontologics Inc. ONTOS provides a persistent object store for C++ objects through a class library. The definition of a persistent class involves deriving new classes from a client library base class called `Object`. Collections of objects in ONTOS are provided through specially defined persistent classes called aggregates. The ONTOS aggregates include set, list, array and dictionary. Dictionary is an aggregate which provides associative lookup. The level of operation is that of individual objects with iterator classes provided to scan aggregates. Gradually more support for high-level database activity has been provided with the development of database tools for browsing and querying using object SQL.

ObjectStore [LLOW91] of Object Design Inc. is a system with similar roots to ONTOS. It supports collections of objects with set, bag and sequence behaviour. Relationships are represented by references embedded in objects, but a special relationships facility allows the programmer to declare that two such object attributes are inverses, and maintains integrity between the two component parts of a relationship. The system supports one-to-one, one-to-many and many-to-many relationships in this way. The application programmer specifies what form of action should be taken to maintain the integrity of relationships under update.

The ObjectStore query language consists of query expressions embedded in C++. These query expressions are targeted at a single collection. In other words, a query is a selection on a collection. The forms of selection can be expressed in terms of nested query expressions and can therefore be quite complex involving several other collections. Thus, the ObjectStore query language can express operations equivalent to relational semi-joins but not full joins, i.e. the result of a query is a subset of the collection being queried.

ODE [AG89] of AT & T Bell Labs is based on the programming language O++ which is an extension of C++. All persistent objects of the same type are grouped together into a cluster of the same name as the type. Thus, a cluster is a collection of objects which gives the extent of a type. Objects may also be grouped into sets and bags. The language O++ provides a special `for` statement for iteration over collections. The language is capable of expressing queries equivalent to the full relational join and also recursive queries. ODE supports both constraint and trigger mechanisms. Constraints and triggers can be specified in class definitions and constraints can be used to derive specialisations of classes by means of predicates.

Systems based on Other Object-Oriented Technologies

In this category, we include systems which either have implementations in languages

other than C++, or where the implementation is in C++ but efforts have been made to keep the system general so that the object model is not tightly integrated with the C++ type model.

The GemStone database system [BOS91] developed at Servio Logic Corporation was one of the first database systems based on object-oriented technologies. GemStone was developed on a Smalltalk platform [GR85]. The GemStone class hierarchy is similar to that found in Smalltalk: objects are grouped into classes which are organised into an *isa* hierarchy rooted at the `Object` class. A query is on a collection of objects and the query language OPAL has a syntax based on Smalltalk.

O_2 [Deu91], [LRV88] of O_2 Technology is a complete object-oriented database system and support environment. It supports C and C++ language interfaces but other language interfaces have been proposed. O_2 provides a complete environment which includes a query language, a user interface generator, a fourth generation language, a graphic programming environment with debugger and a database browser. The original goal of O_2 was to be language independent and this was achieved by providing a type model and data definition language while the methods were written in a variety of languages. The type model includes complex types for tuples, lists, bags and sets.

A number of query languages have been proposed for O_2: these include RELOOP [CDLR90], O_2Query [BCD89] and the functional language LIFOO [BM89].

ORION [BCG+87], [KBC+89] is an object-oriented database system developed in Common Lisp at MCC. The focus of the ORION project was to provide mechanisms to support complex applications such as CAD/CAM and Office Information Systems and it therefore placed a lot of emphasis on the investigation of techniques for schema evolution and version control. The ORION query language is limited to expressing queries against single collections (and their subcollections). In other words, like ObjectStore, it cannot express the equivalent of full relational joins.

Extensions of Relational Technology

A number of systems have been developed based on relational technologies. The advocates of such systems appeared reluctant to abandon the successes of relational systems and therefore chose to extend these systems with object-oriented features rather than develop systems based on new technologies. Exactly what aspects of relational technologies did the designers of these systems wish to retain? In some cases, it was the appeal and familiarity of the relational data model together with its simplicity and generality that was considered important. On the other hand, some designers were keen to reutilise the relational storage and processing technologies that existed and therefore to build their systems on top of a relational platform.

POSTGRES [SK91], [RS87] was developed at Berkeley as a successor to INGRES [SWK76]. The POSTGRES data model extends the relational model to support complex structures. An attribute may be of either a base type, a composite type or a procedure type. Composite types include arrays of base types or relation types. The user may introduce new base types into the model by means of abstract data types. Each tuple has a unique identifier.

POSTGRES has a set-oriented query language called POSTQUEL. This language has support for nested queries, transitive closure and inheritance. Procedures may be defined in terms of POSTQUEL or C functions. A relation scheme may inherit the attributes of one or more specified parent relation schemes. The POSTGRES system also has support for rules, views, version control and historical data.

Starburst [LLPS91] is an extensible relational database system developed by IBM, San Jose. Rather than define a new data model, it extends the relational model by externally defined data types known as EDTs. These EDTs are similar to the abstract data types of POSTGRES. Starburst supports SQL with extensions and employs a rule-based query optimiser.

Iris [FBC+87] of Hewlett-Packard Laboratories realises an object model using a relational platform. The object data model is based on DAPLEX [Shi81] and TAXIS [MBW80]. Classification is expressed in terms of a type graph. A set of subtypes may be specified as disjoint which means that the extents of the subtypes must be non-overlapping. Functions can be stored, derived from other functions or externally defined in a general purpose programming language. Object SQL is supported and queries are translated into relational algebra for evaluation against the relational storage system. Limited forms of type and object evolution are supported.

Summary

In many cases, the object data models of object-oriented database management systems support only the two lower structural levels of our general framework. Collections of values tend to be stand-alone and the forms of interdependencies supported are very limited. Frequently, there is no clear notion of a database structure.

Often, the data model is integrated with the type model of the associated programming language and therefore lacks generality. This also means that data model issues become confused with type issues. For example, in many of these systems, the *isa* relationships between entity categories are modelled by a combination of subtyping and predicates on collections.

In the object-oriented data models, relationships between entities tend to be represented by embedded object references. Support for modelling relationships consists

of the ability to express a dependency between the corresponding object attributes that represent such a relationship.

The levels of operation supported are at the level of values and collections. However, it is still the case that in many of these systems the main support is at the level of values. The associated query languages may be limited in terms of expressibility as is the case with ObjectStore and ORION.

On the whole, the systems based on extensions of relational technology fare better with respect to their support for database activities. Not only do these systems tend to support all three levels of structure and operation, but they also support general constraint models and facilities such as version control and views.

In summary, the object data models of existing object-oriented database management systems tend to be limited in terms of data modelling support. They are frequently specific to a particular implementation platform. Further, many of them do not support well-established database system requirements such as data independence, database evolution and views.

Chapter 3

The Structural Model: BROOM

In this chapter, we specify the structural part of the collection model which determines the forms that collections may take and the ways in which collections may be interrelated by means of static constraints. The structural part of the collection model is referred to as BROOM.

A complete specification of a data model usually will consist of an informal textual description along with a formal specification. The informal textual description will provide an overview of the model and identify its main features. The formal specification should be a complete and precise definition of the semantics and is the basis for the implementation of the model. Somewhere between these two forms of specification is an intermediate level of specification which is a description of the overall structure of the model and may be presented using a graphical notation. Before giving the overall structure of the model, it may be necessary to introduce some basic definitions and notions. We describe the intermediate level of specification as semi-formal since usually it will combine formal and informal descriptions. Further, it may be incomplete in the sense that it may describe only some aspects of the model.

As an intermediate level of specification, we employ a metacircular description of the structural model BROOM, i.e we describe BROOM in terms of itself. The production of this metacircular description represented an important stage in the design process as it provided an abstract overview of the system which was then refined into a detailed formal specification. Thus, the metacircular description provided an initial structuring of the formal specification.

For those acquainted with the stages of database design, the concept of using a high-level data model to obtain an initial structuring of a system specification which is later refined and then realised should be familiar. For example, it is common to use the entity-relationship model in this way for the process of relational database design. The entity-relationship description identifies the entities of interest in the

application domain - and the relationships between these entities. This description may then be transformed into an equivalent relational database schema which may undergo transformations and refinements into a form suitable for implementation. The entity-relationship model plays an important role in establishing the overall structure of the relational schema which is then enhanced with implementation and operational details. Further, the entity-relationship model of an application is an important part of the system documentation.

Similarly, we believe that data modelling concepts may play an important role in providing an intermediate specification of any application system and thereby in determining the overall structure of a formal specification of that system. In this way, a data model can help bridge the somewhat large gap between the informal textual description and the detailed formal specification of an application system.

In particular, we advocate the use of metacircular descriptions as an intermediate level of specification for data models. Many data models have an associated graphical notation which can be particularly useful as a design and documentation aid. Producing a metacircular description is a useful test of the modelling capabilities of the data model. Further, the metacircular description can be used as a basis for the development of a database management system which supports a uniform treatment of metadata and data. In such a system, browsers and query processors can be used to view and retrieve both schema information and data.

The formal specification of the model is given in the specification language Z [Spi89], [Dil90], [PST91].

Then the complete specification of our structural model, BROOM, consists of the following parts:

- an informal overview
- definitions of the basic notions of collection and collection family
- a metacircular description
- a formal specification in the language Z.

In accordance with this, the specification of the BROOM model is presented in the four sections of this chapter with each section corresponding to one of the above parts.

3.1 An Overview of BROOM

The collection model is a general model designed to support the management of potentially large, interrelated collections of values. In our proposed three level database structure for databases indicated in figure 3.1, the BROOM model is concerned with levels 2 and 3. It specifies the ways in which values can be grouped into collections and how these collections can then be linked together to form a database.

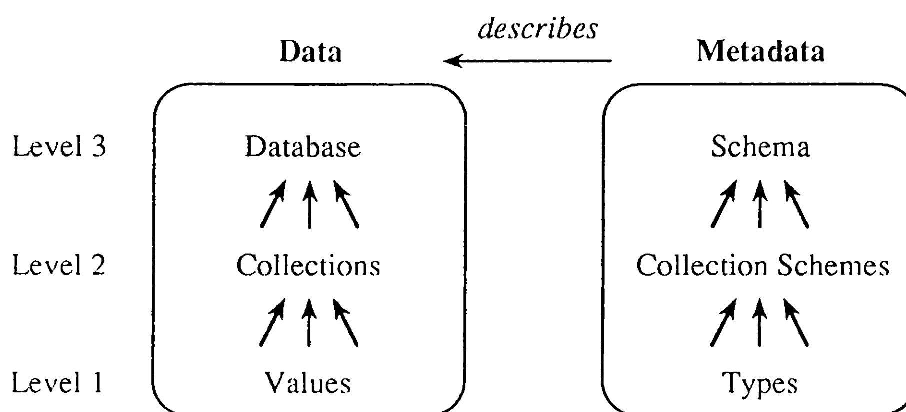

Figure 3.1: The Three Levels of Database Structure

Recall that here the term "value" is used to mean any form of data item that can be described by the underlying type system whether it be a base value such as an integer or string value, or a complex value such as a record or object value. It is important to emphasise that the collection model does not specify the forms that these values may take: this is determined by the underlying type system. In the case of object-oriented systems, the possible forms of values contained in collections will be determined by the type system of the underlying persistent object store and associated persistent programming languages. Generally, we consider an entity of the application domain as being represented by an object value.

While we do not assume any specific underlying type system, there are some general properties of object-oriented type systems that are of interest.

An object type may be considered as a metalevel description of similar objects specifying the properties of these objects in terms of their attributes and operations. Note that here we are concerned only with the abstract properties of an object and therefore in the signatures of its operations, not in their implementations. If an object is created in accordance with a given object type then we say that it is an instance of that type and it will exhibit the properties associated with that type.

One object type, t_1, is said to be a specialisation of a second object type t_2 in the event that the properties of t_1 include the properties of t_2 or refinements of those properties. In such a case, t_1 would be declared as a subtype of t_2 and the properties associated with t_2 would be inherited by t_1. Then an object which is an instance of t_1 will also be an instance of t_2. In this way, an object may be an instance of many types and exhibit the properties of all of those types.

The subtyping relationships between object types determine a type graph. In the event that the form of this graph is restricted to a tree structure, then the type graph is a strict hierarchy and only single inheritance is supported. If no such restriction applies, it is possible that an object type is a subtype of two or more object types that are not themselves related by a subtyping relationship. In such a system, we say that multiple inheritance is supported.

Values are grouped into collections that represent entity categories in the application domain. An entity category corresponds to a role of entities within an application. Significant roles are identified by consideration of the usage of the system in terms of both entity and application characteristics. Note that, while the role of an entity clearly is related to the type of value that represents that entity, this is not the sole determining factor. To illustrate this, we will consider a simple university database.

The university is interested in the various persons associated with the university and these are represented by objects of type `person`. Then objects which are instances of type `person` might be grouped into a number of collections to represent the various categories of person within the application system. The group of categories to be represented may be chosen both to capture relevant semantics of the application domain and also to support envisaged access patterns. For example, an application dealing with car parking permits might require access to all persons associated with the university and we would therefore have a collection `Persons` with all existing person objects as members.

In addition, there might be an application which requires access only to persons who are registered as students in which case we could form a collection `Students`. Further, if only staff individuals can be responsible for university projects, then we might form the collection `Staff`. It is not necessary that the member objects of `Students` and `Staff` are of different types since these objects are classified by means of the membership of collections. Only in the event that we actually require the objects of one or other of the collections to have different "internal" properties in terms of their form and behaviour need we create specialised subtypes of `person` for students and staff.

Relationships between entities of the application domain are represented by mappings between collections of values representing these entities. In our university example, the relationships between staff individuals and the projects for which they are responsible could be represented by a relation called `Manages` which maps members of

the `Staff` collection to members of the `Projects` collection as illustrated in figure 3.2. `Staff` is referred to as the source collection of `Manages`, and `Projects` as the target collection. This relation can be represented by a collection of pair values of the form (s_i, p_j) where s_i is a member of the source collection and p_j is a member of the target collection.

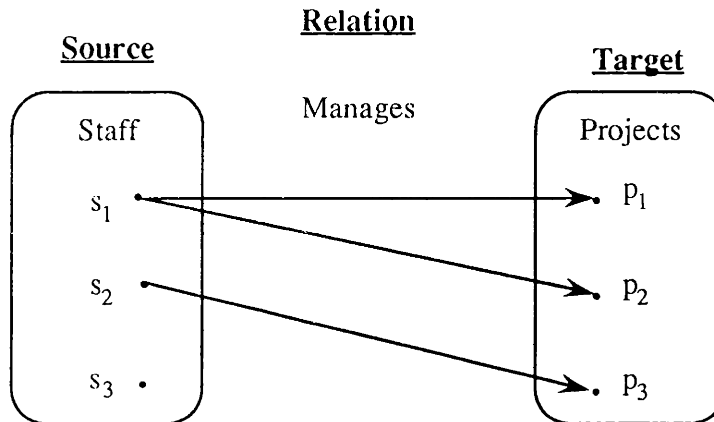

Figure 3.2: A Relation between Collections

There are therefore two forms of collections supported in the BROOM model. Unary collections are those which have atomic values as elements and, generally, represent categories of entities of the application domain. Binary collections are those which have pair values as elements and represent relationships between entities of particular categories. The source and target collections of a binary collection may be any form of collection; hence, a three way association may be represented by a binary collection which links a binary collection to a unary collection.

Collections (unary or binary) may exhibit set or bag properties depending upon whether or not the collection may contain duplicate elements. Further, a collection may have an associated ordering based on one or more attribute values. This is similar to the forms of collections supported in ObjectStore [LLOW91].

It is worth noting at this stage that we identify an object value with the unique object identifier associated with an object. Thus, it is not the objects themselves which are members of a collection of objects but rather the object identifiers. This is of particular significance when considering that objects may belong to many collections.

The graphical notation for collections is given in figure 3.3. A unary collection is represented by a shaded rectangle with different shadings used for sets and bags. The name of the collection is given in the non-shaded part of the rectangle and the type of the members of the collection may be specified in the shaded part. A binary

collection is represented by a shaded rounded-rectangle with the name of the collection inside. A collection which has an associated ordering has a tagged representation.

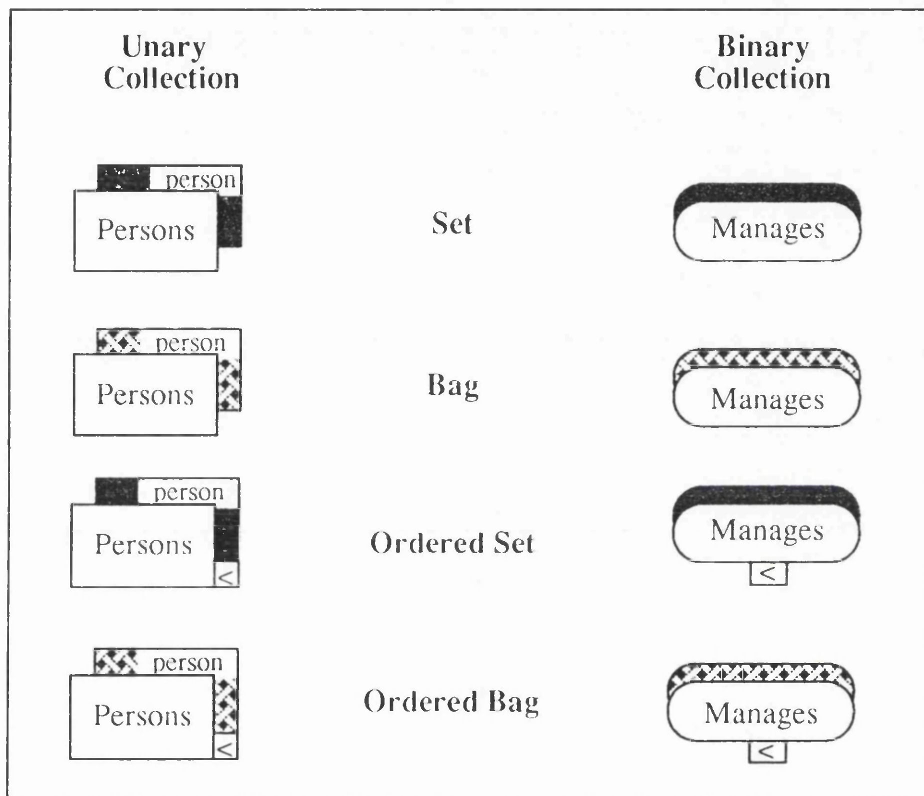

Figure 3.3: Graphical Notation for Collections

Collections may be linked together to form a schema representing the dependencies among collections. There are two basic forms of dependencies among collections. Firstly, collections may be linked together into a classification structure by means of constraints which specify collection families. Secondly, a binary collection representing relationships will be linked to a source and target collection with associated cardinality constraints.

A collection family specifies one or more parent collections and one or more child collections. A child collection is said to be a subcollection of its parent collection. If there is a single parent and a single child, then there is a simple subcollection relationship between the two collections. If a parent has a number of children, then these children may have restrictions which indicate that they are disjoint and/or that they form a cover of the parent in which case every member of the parent collection must be a member of at least one of the children; if the children are disjoint and form a cover of the parent collection, then they are said to form a partition of the parent.

If a collection is a child of two or more parent collections, then it may be specified to be the intersection of the parent collections.

Figure 3.4 gives a simple example schema for the university database expressed in the graphical notation of BROOM. In this example, all the collections have set behaviour and are unordered.

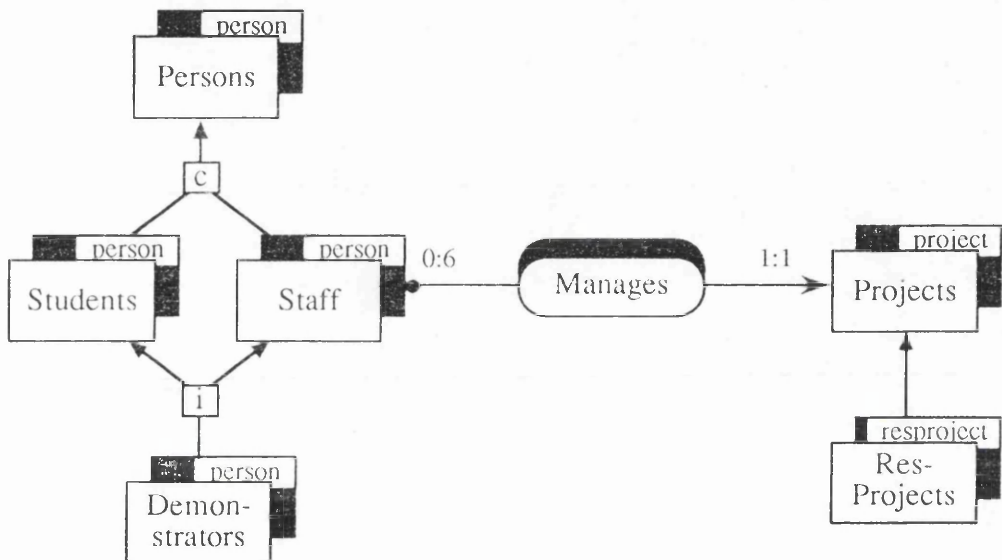

Figure 3.4: An Example Schema Diagram

A directed arc is used to indicate a parent of a family. Where a collection family consists of a single parent and a single child, then a simple subcollection relationship exists and there is a directed arc from the child to the parent with no constraint box. This can be seen in the case of the collection of research projects `ResProjects` which is a subcollection of the collection `Projects`.

In general, the collections of a collection family are linked together by means of arcs and constraint boxes. As described above, a collection family may satisfy conditions of disjointness, coverage, partitioning or intersection, and these are represented by a small constraint box containing `d`, `c`, `p` or `i`, respectively. The collections `Students` and `Staff` form a cover of the collection `Persons`. Thus the collections `Persons`, `Students` and `Staff` form a collection family in which `Persons` is the parent and `Students` and `Staff` are the children. The collection `Demonstrators` is the intersection of the collections `Students` and `Staff` and this is indicated by a constraint box containing an `i` connecting the child collection `Demonstrators` to the two parent collections `Students` and `Staff`.

The source collection of a binary collection is indicated by means of a grounded arc, i.e. an arc with a '•' at the source end, and the target collection is indicated by a directed arc. The cardinality constraints are written alongside the arcs. A cardinality constraint takes the general form `(i:j)` where `i` indicates the minimum level of participation and `j` indicates the maximum level of participation. For example, in figure 3.4, the binary collection `Manages` has source `Staff` and target `Projects`. The cardinality constraint associated with `Staff` is `(0:6)` which specifies that any member of `Staff` can appear in at most six pairs of `Manages`. The cardinality constraint of `Projects` is `(1:1)` which specifies that each member of `Projects` must appear in exactly one pair of `Manages`. Thus a staff individual can be responsible for zero up to six projects and each project must have exactly one associated member of staff.

If the maximum level of a cardinality constraint is given as `n`, then there is no set maximum limit. For example, if the cardinality constraint for `Staff` in figure 3.4 is changed to `(0:n)`, then there would be no restrictions on the number of projects that could be managed by a staff individual.

We now introduce a data modelling language for BROOM which is used to describe types, collections and constraints among collections. The simple university example of figure 3.4 is specified in terms of the BROOM data modelling language in figure 3.5. Note that there are constructs in the BROOM graphical representation language for all of the constructs in the BROOM data modelling language apart from the type descriptions.

Two object types `person` and `project` are defined. For the sake of brevity, we have included only attributes and no operations in these object types. In addition, there is a record type `address` and type `person` has an attribute `home` which is of type `address`.

Collections of the database are described by collection schemes. Each collection scheme gives the name of a collection and its type in terms of the form of the collection and the type of its members. The constraints between the collections are specified in the schema `univdb`. There are two forms of constraint statement - one for collection families and one for relationships. The constraint

```
Projects => ResProjects
```

specifies that `ResProjects` is a subcollection of `Projects`. In general, the parent collection (or list of parent collections) is given on the left of '`=>`' and the child (or list of children) on the right. If a list of collections is supplied then the list may be qualified by a constraint condition. In the case of parent collections, this condition may be `intersection`. In the case of child collections, the condition may be one of `disjoint`, `cover` or `partition`.

```
Type Declarations

record type address
            street :  string;
            city :  string
end;

object type person
            name :  string;
            home :  address
end;

object type project
            title :  string
end;

object type resproject subtype of project
            fundingbody :  string;
            budget :  integer
end;


Collection Declarations

collection Persons, Students, Staff, Demonstrators :  set of person;
collection Projects :  set of project;
collection ResProjects :  set of resproject;
collection Manages :  set of [person,project];


Schema Declaration

schema  univdb
        Persons => cover [Students, Staff];
        intersection [Students, Staff] => Demonstrators;
        Projects => ResProjects;
        Manages <-> Staff (0:6) to Projects(1:1)
end.
```

Figure 3.5: Example of Data Modelling Language

The constraint

```
Manages <-> Staff (0:6) to Projects (1:1)
```

specifies that relation `Manages` has source collection `Staff` and target collection `Projects` with the associated cardinality constraints.

This data modelling language can be considered as a neutral notation which allows us to describe schemas without having to resort to the details of the language associated with a particular implementation of the collection model. An extended version of this data modelling language, DSDL, which includes specifications of collection representations, has been used in the Comandos ODMS system [CBHdP93]. From a DSDL description of a database, a program will be generated automatically to create and initialise the database.

3.2 Collections and Collection Families

Before going on to present a more formal description of the BROOM model, it is first necessary to be more precise about the notions of 'collection' and 'collection family'.

So far we have described a collection as a group of values. Strictly, a collection is an object which comprises a group of values and it therefore has an object identity and an extension property which is the group of values of the collection.

For a given collection C, we will use $id(C)$ to denote the identity of C and $ext(C)$ to denote its extension. Two collections C_1 and C_2 are deemed identical in the case that they have the same identity, i.e. $id(C_1) = id(C_2)$ and then we write $C_1 \equiv C_2$. Two collections C_1 and C_2 are said to be equal, written $C_1 = C_2$, in the event that they have equal extensions, i.e. $C_1 = C_2 \Leftrightarrow ext(C_1) = ext(C_2)$.

In the BROOM model, the extension of a collection may be a set or a bag. If the extension of a collection is a set, then we refer to it as a set collection, and, if its extension is a bag, then we refer to it as a bag collection. Since the properties of a collection are characterised by its extension, we first introduce some general terminology and definitions for sets and bags.

Sets

The elements of a set are distinct and unordered. We denote a set using the usual set notation, i.e. a set containing the values x, y, z will be denoted $\{x, y, z\}$. The empty set is denoted by $\{\}$ or $\varnothing_{set}$.

We assume a criterion for determining whether any given value is a member of a given set and this is given by the membership relation $\in_{set}$, i.e. $x \in_{set} \{x, y, z\} =$ **true**.

Bags

As with a set, the elements of a bag are unordered. However unlike a set, a bag may contain more than one occurrence of an element. We will use angular brackets to denote bags. A bag containing one x, two y's and three z's could be denoted $\langle x, y, y, z, z, z\rangle$. The empty bag is denoted by $\langle\rangle$ or $\varnothing_{bag}$.

There are two representations of bags that are commonly used. One is to represent a bag as a function which maps elements to number of occurrences and the other is to represent a bag as a set of pairs where each pair gives an element and its number of occurrences. We describe each of these in turn.

If a bag is to be represented as a function which maps elements to the natural numbers, $\mathbb{N}$, then the domain of this function, or at least a source set if it is partial, must be specified. We therefore introduce a universal set of elements U, and then for a given bag B :

$$B : U \rightarrow \mathbb{N}$$

Then for some $x \in U$, $B\ x$ gives the number of occurrences of x in B. A membership relation $\in_{bag}$ is defined on bags such that $x \in_{bag} B \Leftrightarrow B\ x > 0$.

For example, given the bag $B = \langle x, y, y, z, z, z\rangle$ then B is given by the function defined as $\{x \mapsto 1, y \mapsto 2, z \mapsto 3\} \cup \{w \mapsto 0 \mid w \in U \wedge w \neq x \wedge w \neq y \wedge w \neq z\}$.

This leads us naturally to the second form of representation of a bag as a set of pairs based upon the usual graph notation for discrete functions. The bag B given above can be represented as the set of pairs $\{(x, 1), (y, 2), (z, 3)\}$. With this representation the membership relation $\in_{bag}$ can be defined in terms of the membership relation $\in_{set}$ as follows: $x \in_{bag} B \Leftrightarrow \exists\, n \in \mathbb{N}_1.(x, n) \in_{set} B$, where $\mathbb{N}_1$ denotes the set of positive integers.

Collections

For a given collection C it is necessary to be able to determine whether C is a set or bag collection and we therefore introduce predicates *is_set* and *is_bag* on collections such that for a collection C, $is_set(C) =$ `true` if $ext(C)$ is a set and $is_bag(C) =$ `true` if $ext(C)$ is a bag. Clearly, since $ext(C)$ must be either a set or a bag, $is_set(C) \vee is_bag(C) =$ `true`.

A collection does not represent an arbitrary grouping of values but rather a grouping of values of a common type. Hence a collection C will have an associated member

type. We introduce an additional property of collection objects that identifies their member types. For a given collection C and a given type T, if $member_type(C) = T$ then for any value x in the extension of C, x must be an instance of type T, written $x : T$.

We now define a membership relation $\in$ on collections. Let C be a collection of elements of type T, i.e. $member_type(C) = T$. Then for $x : T$,

$$x \in C \quad \Leftrightarrow \quad \text{if } is_set(C) \text{ then } x \in_{set} ext(C) \text{ else } x \in_{bag} ext(C)$$

Note that the member type of a binary collection will be a product type of the form $T_1 \times T_2$. Such product types will also be referred to as pair types and may be written in the form $[T_1, T_2]$. For example, the member type of the binary collection `Manages` in figure 3.4 is `[person,project]`.

It is useful to have a function which given a collection and an element will give the number of occurrences of that element in the collection. For a set collection the number of occurrences is either 0 or 1. Let C be a collection of elements of type T. Then for $x : T$, $C \diamond x$ gives the number of occurrences of x in C, i.e.

$$C \diamond x \quad = \quad \text{if } is_set(C) \text{ then (if } x \in_{set} ext(C) \text{ then 1 else 0) else } ext(C)\ x$$

If C is not a set collection then it must be a bag collection and $ext(C)$ is a bag. Then the number of occurrences of x in C will be given by $ext(C)\ x$.

3.2.1 Collection Families

A collection represents some general concept of the application domain. A specialisation of a general concept into a more precise concept is represented by another collection which contains only those values of the first collection that satisfy this more precise notion. The collection that represents the more specialised concept is referred to as a subcollection of that which represents the more general concept.

For two collections C_1 and C_2, if C_1 is a subcollection of C_2, then we say that C_2 is a supercollection of C_1.

Definition 3.1 **Subcollection.** Let C_1 and C_2 be collections. C_1 is a subcollection of C_2 iff $\forall x \in C_1 . C_1 \diamond x \leq C_2 \diamond x$. Then we write $C_1 \preceq C_2$. □

Note that there is no restriction that a subcollection must have the same behaviour as its supercollection in terms of whether their extensions are sets or bags. For example, assume C_1 is a set collection with $ext(C_1) = S$ and C_2 is a bag collection with $ext(C_2) = B$. Then if

$$S = \{x, y, z\}$$

and

$$B = \langle x, x, y, y, y, z \rangle$$

then $C_1 \preceq C_2$.

The subcollection relation $\preceq$ is a pre-order on collections, i.e. it is both reflexive and transitive. It is not a partial order since the anti-symmetric property does not hold. Consider two collections C_3 and C_4 with $ext(C_3) = S'$ and $ext(C_4) = B'$ where

$$S' = \{x\}$$

and

$$B' = \langle x \rangle.$$

Then $C_3 \diamond x = C_4 \diamond x$ and we have $C_3 \preceq C_4$ and $C_4 \preceq C_3$. However, it is not the case that $C_3 = C_4$ since the extension of C_3 is a set and the extension of C_4 is a bag.

Clearly, if $C_1 \preceq C_2$ the member types of C_1 and C_2 must be related in some way since all member values of C_1 are also member values of C_2. The notion of subtyping underlies (most) object-oriented type systems and we will therefore assume a subtyping relation $\leq_t$ on types and that it is a partial order, i.e. $\leq_t$ is reflexive, antisymmetric and transitive. Then for types T_1 and T_2, if $T_1 \leq_t T_2$ and $x : T_1$, then $x : T_2$. If $C_1 \preceq C_2$ then $member_type(C_1) \leq_t member_type(C_2)$.

The subcollection relation expresses a conceptual dependency between two collections. By specifying that the subcollection relation must hold between C_1 and C_2, a constraint is imposed on the membership of these collections such that any value in C_1 must also be in C_2. Such constraints may be used either to check the validity of database updates or to trigger actions such as update propagations to ensure that database consistency is maintained.

Conceptual dependencies among collections are specified by means of collection families. A collection family comprises two sets of collections referred to as the parents and the children. Each child collection is constrained to be a subcollection of each parent collection. Further forms of constraints may be specified on the members of a collection family. These are given in terms of the predicates *disjoint*, *partition* and *intersection* given below. CS and PS are sets of collections such that CS is the set of children of a family and PS is the set of parents.

$$disjoint(CS) \Leftrightarrow \forall C_i, C_j \in CS.(C_i \not\equiv C_j \Rightarrow \neg \exists x.(x \in C_i \wedge x \in C_j))$$

$$partition(PS, CS) \Leftrightarrow \forall P_i \in PS. \forall x \in P_i. P_i \diamond x = max\{C_j \diamond x \mid C_j \in CS\}$$

$$intersection(PS, CS) \Leftrightarrow \forall C_i \in CS. \forall x \in C_i. C_i \diamond x = min\{P_i \diamond x \mid P_i \in PS\}$$

A set of collections is disjoint if no pair of member collections has a common member value. A set of child collections is a cover of a set of parent collections if all of the members of the parents appear in at least one of the children. In the case of bag collections, the number of occurrences of a value in a parent must equal the maximum number of occurrences of that value in the children. In other words, at least one of the children must have at least the same number of occurrences as appears in the parents. For example, if C, C_1 and C_2 are bag collections with

$$ext(C) = B, \; ext(C_1) = B_1 \text{ and } ext(C_2) = B_2$$

where

$$B = \langle x, x, y, y, y, z \rangle$$

and

$$B_1 = \langle x, y, y, y \rangle, \; B_2 = \langle x, x, y, z \rangle$$

then the family with parent C and children C_1 and C_2 is a cover.

If each child collection of a family contains all of those member values that are common to every parent, then the set of child collections is an intersection of the set of parents. In the case of bag collections, the number of occurrences of a value in a child is equal to the minimum number of occurrences of that value in the parents. For example, if bag collections C_1 and C_2 are as above, and, C_3 is a bag collection with $ext(C_3) = B_3$ such that

$$B_3 = \langle x, y \rangle$$

then the family with parents C_1 and C_2, and child C_3 is an intersection.

These dependencies among the collections of a collection family are specified by means of a constraint term which is one of `disjoi..t`, `cover`, `partition`, `intersection` or none. A collection family which is a partition has disjoint children and the children form a cover of the parents. We now give a definition of a collection family.

Definition 3.2 Collection Family. A collection family F is a triple (PS, CS, c) where PS and CS are disjoint sets of collections and c is a constraint term such that:

(i) $\forall P_i \in PS. \forall C_j \in CS. C_j \preceq P_i$

(ii) $(c = \texttt{disjoint} \wedge disjoint(CS))$
$\vee$
$(c = \texttt{cover} \wedge cover(PS, CS))$
$\vee$
$(c = \texttt{partition} \wedge disjoint(CS) \wedge cover(PS, CS))$
$\vee$
$(c = \texttt{intersection} \wedge intersection(PS, CS))$
$\vee$
$c = \texttt{none}$

□

A collection family with constraint term none is used to represent a simple subcollection dependency between collections. For a given parent collection, all of its subcollections may be specified by means of a single collection family of this form. Alternatively, they can be specified by a number of families each with the constraint term none.

The general form of a collection family permits multiple parents and multiple children. In the case of particular constraint terms, the constraint may be such that the membership of two or more collections must be equal.

For example, assume a collection family $F = (PS, CS, \texttt{cover})$ and that we have $P_i, P_j \in PS$ such that $P_i \not\equiv P_j$. Then for any value x, $P_i \diamond x = P_j \diamond x$. It follows that $ext(P_i) = ext(P_j)$ and hence $P_i = P_j$. Thus, if a collection family has multiple parents and a constraint term cover or partition, then these parents will always be equal, i.e. their extensions will contain the same values.

Further, assume a collection family $F = (PS, CS, \texttt{intersection})$ and that we have $C_i, C_j \in CS$ such that $C_i \not\equiv C_j$. Then for any value x, $C_i \diamond x = C_j \diamond x$. It follows that $ext(C_i) = ext(C_j)$ and hence $C_i = C_j$. Thus, if a collection family has multiple children and a constraint term intersection, then the children will always be equal.

However, it is possible to have a collection family with multiple parents and multiple children and the constraint term disjoint where the parents are not constrained to be equal and neither are the children. An example of such a collection family is given in figure 3.6.

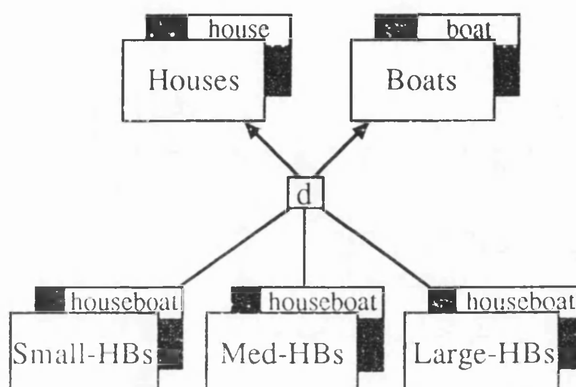

Figure 3.6: Collection Family with Multiple Parents and Children

In the collection family of figure 3.6, there are three children which represent disjoint categories of houseboats. Each child collection is a subcollection of both the parent collections `Houses` and `Boats`. It is assumed in the example that multiple subtyping is available since the member type of the children is `houseboat` and this must be a subtype of both `house` and `boat`.

From the above discussion, it is seen that there are certain forms of collection family that constrain certain collections of the family to always be equal. It is difficult to justify the use of such forms of collection family and for this reason a particular realisation of the collection model may choose to prohibit the specification of such families. However, we prefer not to restrict the definition of collection family to invalidate such collections as this would result in a loss of generality and uniformity.

3.3 A Metacircular Description of BROOM

In this section, we present a description of BROOM in terms of itself. Note that this description can be regarded as a specification of the main characteristics of the model and it is not intended to describe any particular implementation of the model. However, it could form the basis of a system in which the data and metadata are managed uniformly.

The metacircular description is presented in several stages with appropriate graphical descriptions. Each graph is a subgraph of the overall description.

As described in the previous sections, a collection can take one of two forms and one of two behaviours. It may be either a unary or a binary collection and either

exhibit set or bag behaviour depending upon whether duplicate elements may occur. In addition, a collection may have an associated ordering. These are described in figure 3.7.

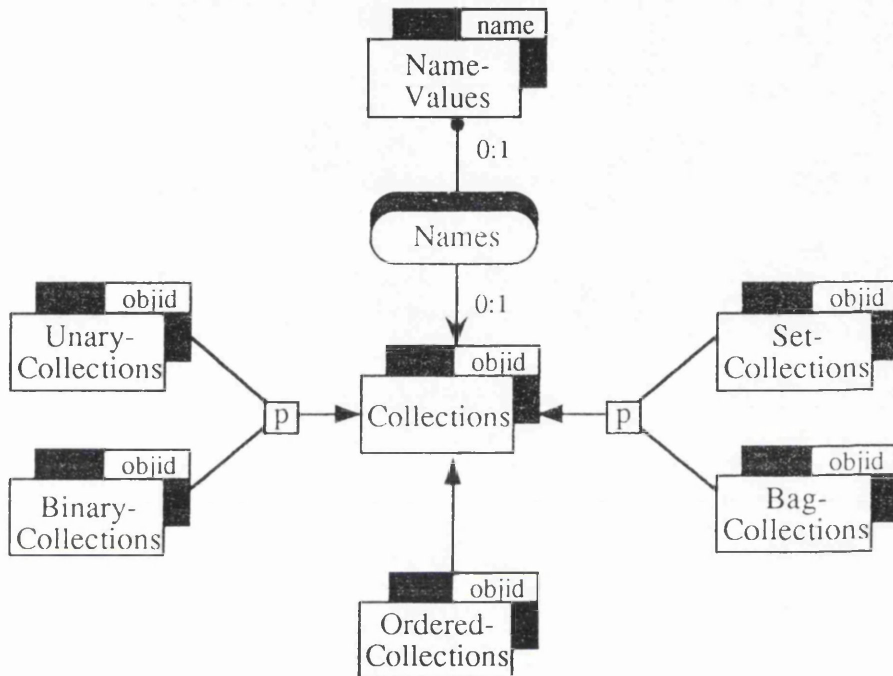

Figure 3.7: BROOM Collections

A collection may have an associated name. Generally, all collections of a database will be named but it is possible in some systems that temporary collections constructed during query evaluation may exist in the database without an associated name. For this reason, we have not modelled it as an obligatory relationship on `Collections` although some systems may wish to make it so.

Figure 3.7 provides a form of classification structure for BROOM collections. It gives no information about what a collection itself is other than saying that the member type of `Collections` is `objid` which indicates that a collection is an object. Recall that a value is any data item supported by the system and includes objects. Then a collection represents a grouping of values and is itself a value. In figure 3.8, we give a graphical BROOM description of the value structures in BROOM.

A value may be either a pair with a value as first and second element - or an atomic value. An atomic value may be either an object or a base value such as an integer or a string: other base values such as Boolean values and indeed composite record values are possible but are omitted here for the sake of brevity.

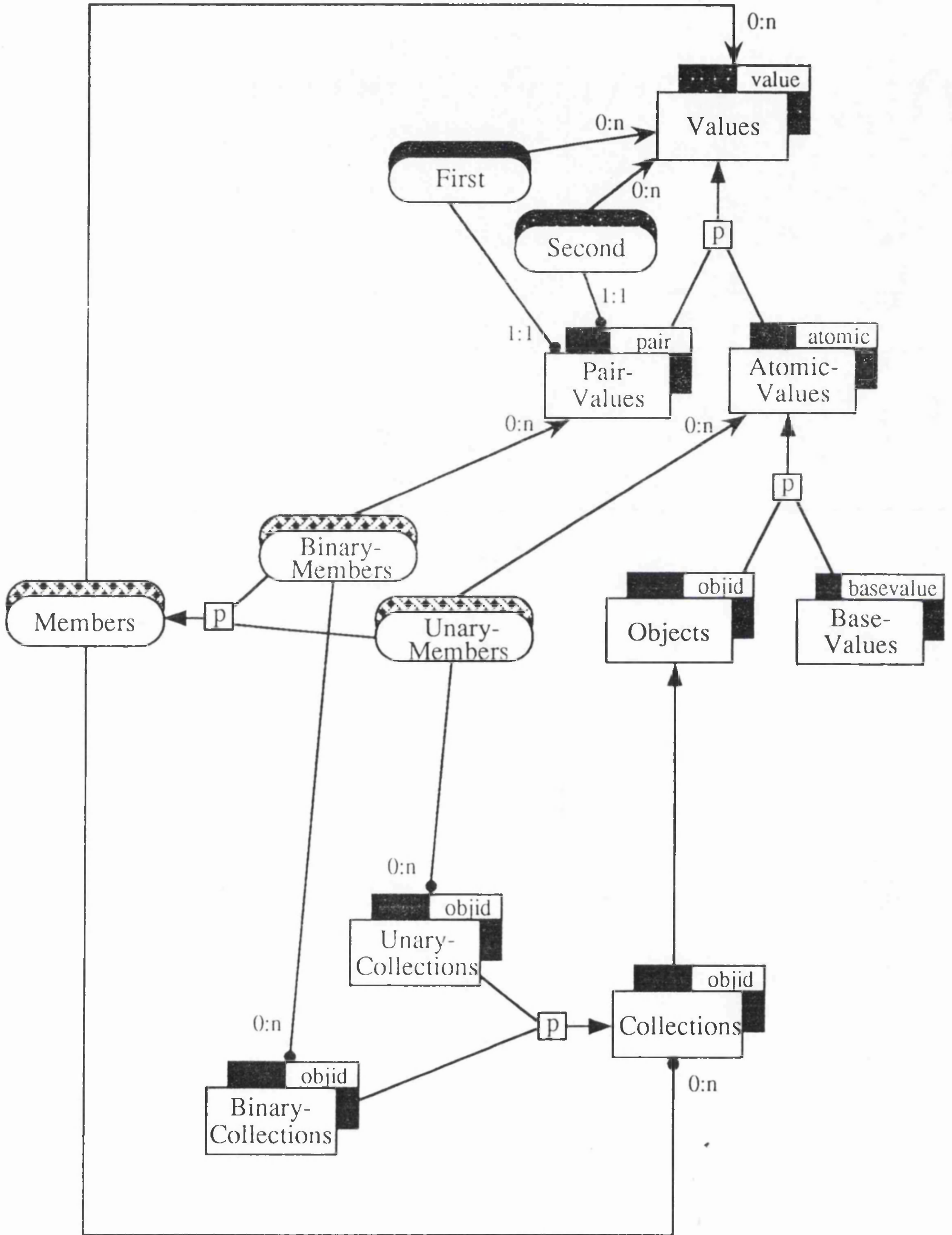

Figure 3.8: Value Structures in BROOM

Every collection is an object as indicated by the subcollection arc between `Objects` and `Collections`.

The `Members` binary collection represents the relationships between collections and their member values. The shading indicates that `Members` is a bag and therefore pairs may occur more than once in the collection. Consider a simple example of a database with two collections, S and B, where S is a set collection and B is a bag collection. Assume that the extensions of these collections are:

$$\begin{aligned} ext(B) &= \langle x, x, y, y, y, z \rangle \\ ext(S) &= \{z, w\} \end{aligned}$$

then the membership of values in collections is represented by the bag collection `Members` where

$$ext(\texttt{Members}) = \langle (B, x), (B, x), (B, y), (B, y), (B, y), (B, z), (S, z), (S, w) \rangle$$

Thus the bag collection `Members` represents the membership of both set and bag collections. If a collection is a set collection, the corresponding pairs of `Members` should occur exactly once. Note that `Members` is specialised by a partition into `BinaryMembers` which relates `BinaryCollections` to `PairValues`, and, `UnaryMembers` which relates `UnaryCollections` to `AtomicValues`.

A value in BROOM is associated with one or more types. The BROOM model is independent of the underlying type system and therefore a full description of the possible forms that types may take does not form part of the BROOM description. However, we are interested in the general categories of types in order that we can associate types with values and ensure consistency of this association at the level of collections.

Since the detailed structure of types is not of concern, we simply model types by their names and classify them according to their general form as shown in figure 3.9. This structure parallels the corresponding structure for values of figure 3.8. The main difference to note is that the `MemberType` collection is a set collection rather than a bag collection and the cardinality associated with its source `Collections` is `(1:1)`. A collection type has a single member type whatever the behaviour of that collection.

One additional piece of type information required for the collection model is the subtype ordering on types. This is modelled in figure 3.10 as a binary collection with source and target `Types`. `Subtypes` should be considered as a relation representing the declaration of subtype relationships between types introduced explicitly in type declarations. Then the reflexive transitive closure of `Subtypes` under relation composition gives the general subtyping relation, $\leq_t$ that was introduced in section 3.2.

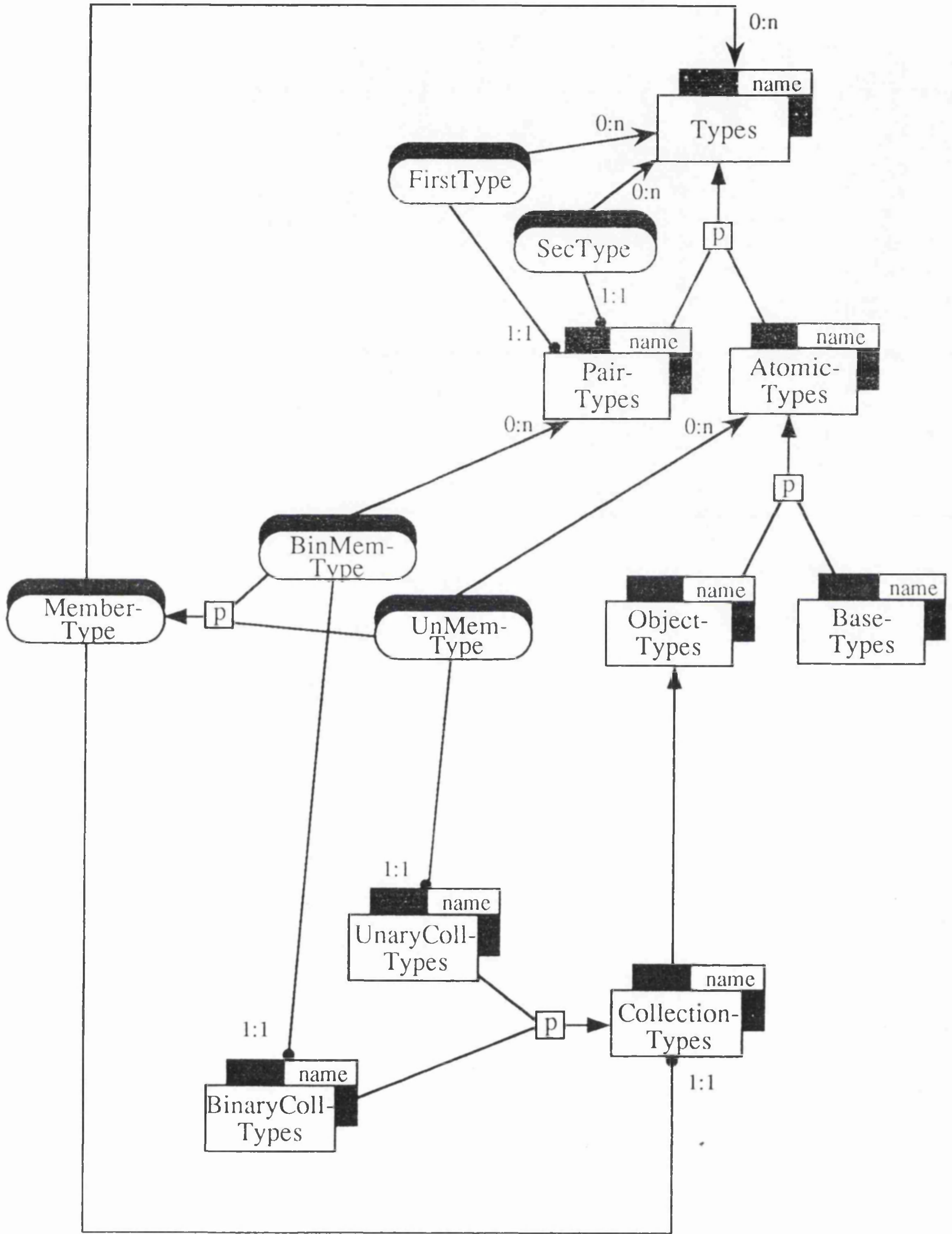

Figure 3.9: Types for BROOM

As stated previously, the relation $\leq_t$ imposes a partial order on types which means that the declaration of types should be non-circular in that no type should be defined as a subtype of itself. This condition corresponds to the anti-symmetric property of partial orders, i.e. if $x \leq_t y$ and $y \leq_t x$ then $x = y$.

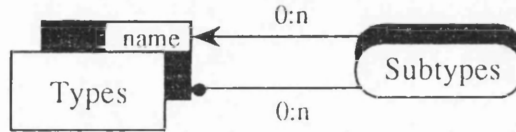

Figure 3.10: Subtype relation on Types

The value and type structures are related by the binary collection `Instances` as indicated in figure 3.11. A value in BROOM is an instance of one or more types. For example, an object value which is an instance of type `male` will also be an instance of all supertypes of type `male`. All types in BROOM - including collection types - may have any number of instances. Thus there is a many-to-many relationship between types and values.

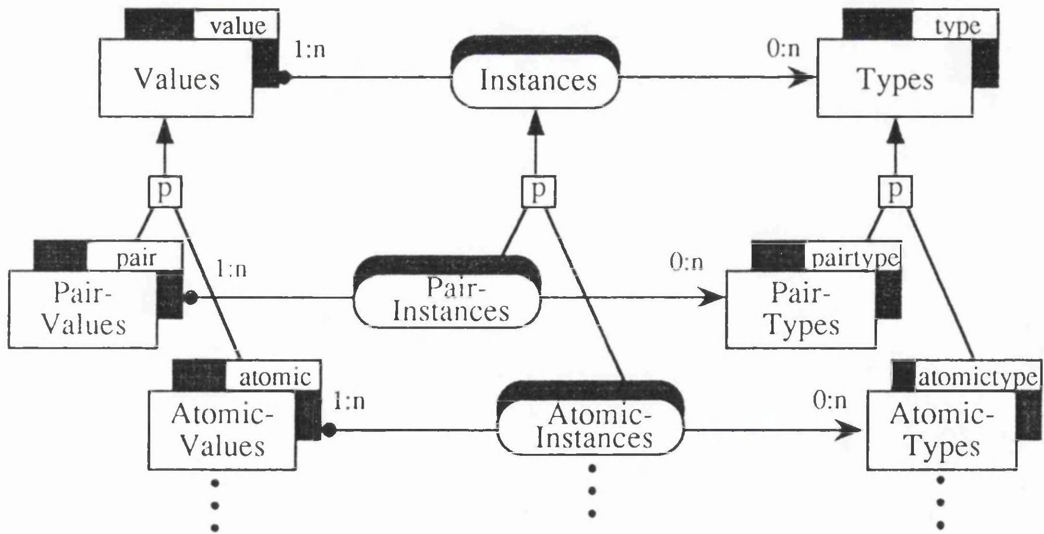

Figure 3.11: Relating Types to Values

The `Instances` binary collection is specialised over the two classification structures for values and types to ensure that there is consistency between the form of a value and its type. For example, an atomic value can only be an instance of atomic types and a unary set collection can only be an instance of unary set collection types. In figure 3.11, we show only part of the resulting structure.

The collection family structures of BROOM are specified in figure 3.12. Here, we consider a collection family as an object which comprises 2 or more collections: these collections are partitioned into a set of parent collections and a set of child collections as indicated by the partition of the relation `Contains` into `Parents` and `Children`. The cardinality constraints are such that we assume that each collection family must have at least one parent collection and at least one child collection.

Collection families are classified according to the constraints among the members of the families. If the children of a collection family are specified to be disjoint, then the collection family will belong to the subcollection `DisjointFamilies`. If the children of a collection family are specified to cover the parent collections, then the collection family will belong to the subcollection `CoverFamilies`. The intersection of `DisjointFamilies` and `CoverFamilies` gives the collection of collection families where the child collections partition the parent collections. If a collection family belongs to the collection `IntersectFamilies`, then each child will be formed from the intersection of the members of the parents. A collection family which belongs to `IntersectFamilies` can belong to neither `DisjointFamilies` nor `CoverFamilies`.

If a collection family is a member of `CollectionFamilies` but not a member of any of its subcollections, then the constraint term of the family is `none`. This means that the children of the family are subcollections of the parents - but no additional constraints apply.

Given the binary collections `Parents` and `Children`, one can form a binary collection `SubCollections` which relates collections to their immediate subcollections. This is given by the expression

$$\texttt{SubCollections} = \texttt{Parents}^{-1} \circ \texttt{Children}$$

where $\texttt{Parents}^{-1}$ is the inverse of `Parents` and o is relation composition. Clearly, like `SubTypes`, this relation must be non-circular.

If a binary collection is used to represent relationships between members of collections, then it must have associated source and target collections and each of these will have an associated cardinality constraint. This is represented in figure 3.13. `BinaryCollections` is the set of all binary collections and `Relations` is a subcollection which is the set of all binary collections that represent a relation on two collections. Each member of `Relations` is related to a source collection and a target collection through `Sources` and `Targets`, respectively. Note that the source and target collections may be any collection and hence a relation is not restricted to mappings between two unary collections: for example, it could be a mapping between a binary collection and a unary collection.

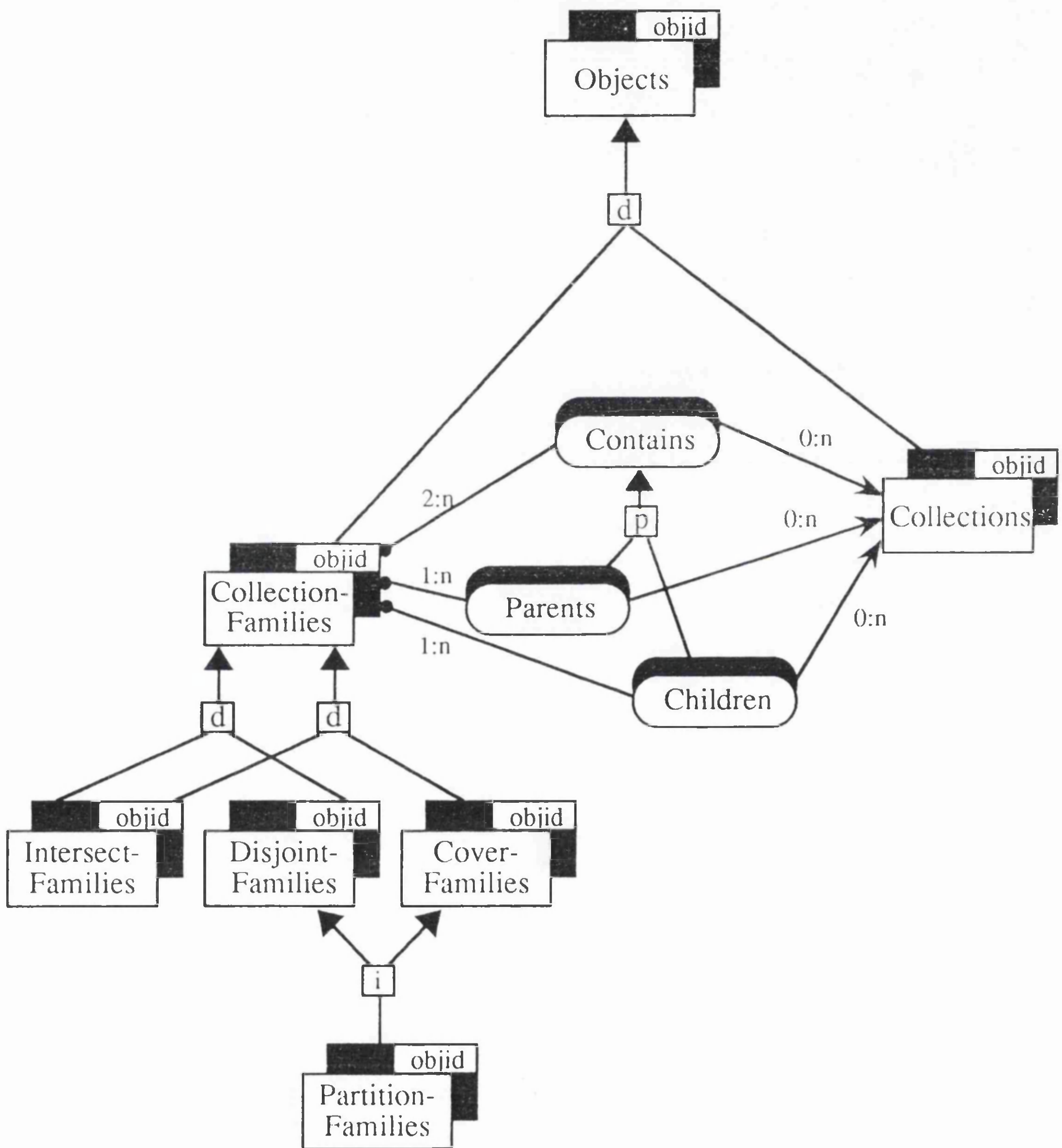

Figure 3.12: Collection Families

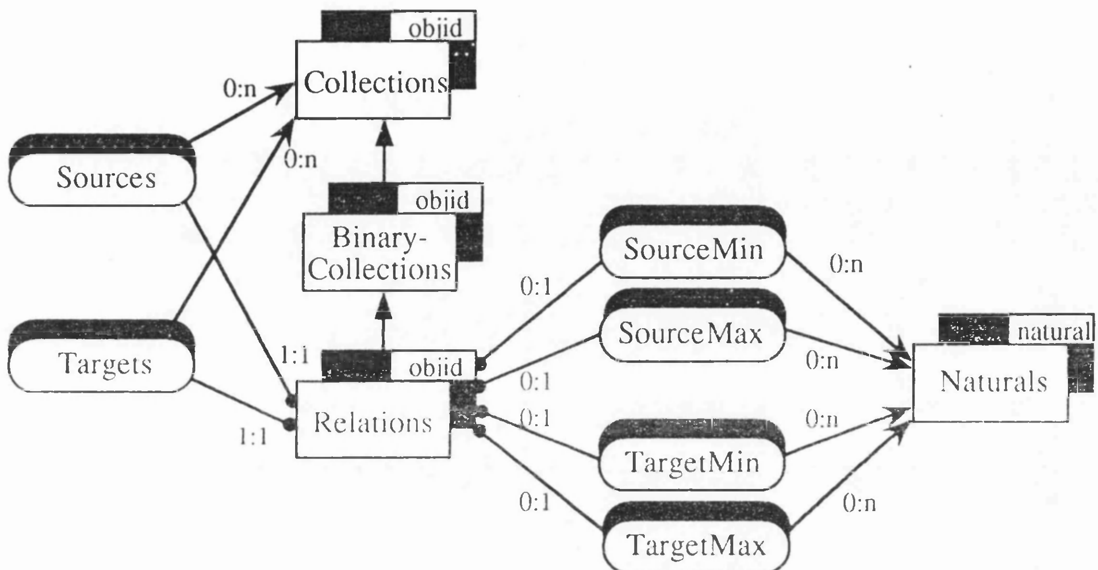

Figure 3.13: Relations

Associated with the source and target of a binary relation there is a minimum and maximum value which indicates the possible levels of participation of members of these collections in the relation, i.e. the number of pairs in which a particular value may appear. If the maximum is specified as `n`, then there is no upper limit and therefore no maximality constraint applies. Similarly, if the minimum is specified as `0`, then no minimality constraint applies. These values may therefore be considered as default values. Thus, the source of a relation optionally has a minimum value specified and this we represent by the relation `SourceMin` which has source `Relations` and target `Naturals`. In addition, the source optionally has a maximum value specified and this we represent by the relation `SourceMax`. Correspondingly, we have the relations `TargetMin` and `TargetMax` to represent the target's cardinality constraints. Note that there are the additional constraints that for any source or target relation the minimum value must be less than the maximum value: this is not captured in the metacircular description.

The process of constructing the metacircular description of BROOM forced clarification of many of the notions of the model. Certain irregularities of the BROOM graph structures that arose helped to identify areas of the model that were open to generalisation. For example, the original proposal for the BROOM model had three sorts of collections - sets, bags and relations. When the metacircular description of this first proposal was produced, the issue arose of how these should be related within a classification structure for collections. The resulting graph lacked symmetry and it was apparent that the original proposal was too restrictive and lacked orthogonality. The notion of collection and the sorts of collection available was then generalised into

the current proposal which is much more uniform. This is just one example of the many changes that were made to the model at this stage.

3.4 A Z Specification of BROOM

In this section, we present a specification of the BROOM model in the specification language Z [Spi89]. Z is based on set theory and predicate logic which are familiar to most people in the database field. It was therefore felt that it would be relatively easy for those without a detailed knowledge of the specification language to read and understand the essence of specifications in Z.

There are a number of good introductory textbooks on Z which give a complete description of the language, examples of specifications and reasoning about these specifications e.g. [Dil90], [PST91], [Spi89]. Clearly, it is beyond the scope of this section to provide a comprehensive description of Z, but it is hoped that the uninitiated reader will at least gain some insight into the style of Z specifications.

A point to emphasise yet again is that we are still discussing logical specification and not implementation. The representations we talk of in this section are mathematical representations and not physical representations.

The unit of specification in Z is called a schema. To avoid any confusion between the notion of a schema of Z and a database schema, we will refer to a schema of Z as a Z-schema. A Z-schema consists of two parts: the first part consists of variable declarations and the second part specifies properties, or constraints, on the values of these variables expressed as predicates.

Z incorporates mechanisms for introducing definitions in order that specifications may be abbreviated thus making them clearer. Indeed, a mathematical toolkit was produced by Spivey [Spi89] and this provides definitions for well-known operations on and properties of sets, relations, functions, sequences and bags. These concepts may be defined by means of generic definitions of the form:

$[X_1, X_2, ..., X_n]$
D
———
P

The $X_1, X_2, ..., X_n$ are the formal generic parameters which can occur in the types of the identifiers in declaration D. The predicate P defines the identifiers introduced in D.

We start our specification of BROOM by defining some of the basic concepts used in BROOM by means of generic definitions. We start with a definition of `disjoint`. The mathematical toolkit given by Spivey does in fact include a definition of disjointness [Spi89, p. 125] but it is defined over indexed families of sets rather than simply sets of sets. We therefore redefine it in terms of sets of sets.

$$
\begin{array}{l}
[X] \\
\hline
\mathtt{disjoint}_ : \mathbb{P}(\mathbb{P}(\mathbb{P}\,X)) \\
\hline
\forall S : \mathbb{P}(\mathbb{P}\,X) \bullet \\
(\mathtt{disjoint}\; S \Leftrightarrow \forall X_1, X_2 : S \bullet \\
\qquad (X_1 \neq X_2 \Rightarrow \neg\,(\exists x : X \bullet x \in X_1 \land x \in X_2)))
\end{array}
$$

`disjoint` can be thought of as a unary relation over sets of sets. Then it is of type $\mathbb{P}(\mathbb{P}(\mathbb{P}\,X))$ since it is the set of all sets of sets which satisfy the disjointness condition. The predicate part of the generic definition states what it means for a particular set of sets to be disjoint.

Now we turn to define concepts that correspond to the BROOM notion of collection family. While these, generally, accord with the definition of collection family given in section 3.2, it should be pointed out that it suffices here to define these over sets of sets. The main reason for this is the fact that, as seen in section 3.3, our metacircular description involves mainly set collections and correspondingly our specification deals, in the main, with sets. The one exception is that of the `Members` binary collection which has bag behaviour. This case is dealt with separately and, in fact, we shall see how the notion of partition etc. on bag collections collapses naturally into a definition in terms of sets.

Recall that a collection family comprises a set of parent collections and a set of child collections. We therefore define `subcolls` to be a relation on sets of sets as follows:

$$
\begin{array}{l}
[X] \\
\hline
\,\mathtt{subcolls}\, : \mathbb{P}(\mathbb{P}\,X) \leftrightarrow \mathbb{P}(\mathbb{P}\,X) \\
\hline
\forall S, T : \mathbb{P}(\mathbb{P}\,X) \bullet \\
(S\;\mathtt{subcolls}\;T \Leftrightarrow \forall X_1 : S \bullet \forall X_2 : T \bullet X_1 \subseteq X_2)
\end{array}
$$

Given a set of sets, S and a set of sets T, then S is related to T under `subcolls` in the event that each member of S is a subset of each member of T.

Next we introduce generic definitions for `covers`, `partitions` and `intersects` as relations on sets of sets.

$$
\begin{array}{l}
[X] \\
\hline
_\ \mathtt{covers}\ _, _\ \mathtt{partitions}\ _, _\ \mathtt{intersects}\ _ : \mathbb{P}(\mathbb{P}\,X) \leftrightarrow \mathbb{P}(\mathbb{P}\,X) \\
\hline
\forall S, T : \mathbb{P}(\mathbb{P}\,X) \bullet \\
((S\ \mathtt{covers}\ T \Leftrightarrow (S\ \mathtt{subcolls} T\ \wedge \\
\quad \forall X_1 : T \bullet X_1 = \bigcup S)) \wedge \\
(S\ \mathtt{partitions}\ T \Leftrightarrow \\
\quad (S\ \mathtt{subcolls}\ T \wedge S\ \mathtt{covers}\ T \wedge\ \mathtt{disjoint}\ S)) \wedge \\
(S\ \mathtt{intersects}\ T \Leftrightarrow (S\ \mathtt{subcolls}\ T \wedge \\
\quad \forall X_1 : S \bullet X_1 = \bigcap T))
\end{array}
$$

The definitions of the covers and intersects relations use the generalised union, $\bigcup$, and generalised intersection, $\bigcap$, definitions of the Z mathematical toolkit. If S is a set of sets, then $\bigcup S$ contains all the values which are members of some member of S. The set $\bigcap S$ contains those values which are members of all members of S.

We add one further generic definition that will be required in our specification. A relation which maps a set onto itself is termed homogeneous. There may be a requirement that such a relation represents a partial order in which case its reflexive transitive closure satisfies the conditions of a partial order, i.e. reflexivity, antisymmetry and transitivity. It follows that if a relation is to represent a partial order then it must be non-circular.

$$
\begin{array}{l}
[X] \\
\hline
\mathtt{noncircular} : \mathbb{P}(X \leftrightarrow X) \\
\hline
\forall R : X \leftrightarrow X \bullet \\
\quad R \in \mathtt{noncircular} \Leftrightarrow ((x, y) \in R^* \wedge (y, x) \in R^* \Rightarrow x = y)
\end{array}
$$

Z is a typed language where a type is a set of values. There are a number of standard types which include $\mathbb{Z}$ the set of integers, $\mathbb{N}$ the set of non-negative integers and $\mathbb{N}_1$ the set of positive integers. The specification can also include a number of user-defined types. We introduce two such types in our specification as follows:

$$[OBJID, NAME]$$

$OBJID$ is the set of all object identifiers and $NAME$ is the set of all name values. Note that we will use these name values both for the names of BROOM types and also for the names of BROOM collections. We shall not specify that the set of type names and the set of collection names have to be disjoint: a particular realisation of the model may wish to add this constraint.

Having introduced these generic definitions and primitive types, we now present the Z specification of BROOM. The general approach adopted for transforming a BROOM

description into a Z specification is to produce a Z specification for each classification structure and then combine these based on relations that map between classification structures. In this way, the Z-schemas are combined to eventually produce a Z-schema for the entire BROOM description.

A classification structure is an acyclic directed graph that describes a collection and all of its subcollections. To construct a Z-schema for a classification structure, we start at the "leaf nodes" of the structure (i.e. those collections with no subcollections) and gradually progress to the "root" collections (i.e. those collections with no supercollections). Generally, a Z-schema is produced for each collection family and these are then combined as progress is made towards the root collections, resulting in a single Z-schema for the classification structure.

We begin with the BROOM descriptions of collections given in figure 3.7. Each of the collection families with parent `Collections` is considered in turn and then these are combined into an overall Z-schema *COLLECTIONS* which corresponds to the whole figure.

Note that we adopt the convention that Z-schema names are entirely upper case whereas collection names start with an upper case letter and contain a mix of upper case and lower case letters. Further, we distinguish between BROOM collections and the mathematical constructs used to represent these collections in our Z specification. Z constructs have italicised names whereas the names of BROOM constructs are in typewriter font. For example, `Collections` is the BROOM collection used to represent the collections of a database in the metacircular description. This is represented in the Z specification by the set *Collections* and its properties are described in the Z-schema *COLLECTIONS*.

The first Z-schema is called *COLLECTIONFORMS* and it introduces the sets *Collections*, *UnaryCollections* and *BinaryCollections* as sets of object identifiers.

COLLECTIONFORMS

$Collections, UnaryCollections, BinaryCollections : \mathbb{P}\, OBJID$

$\{UnaryCollections, BinaryCollections\}$ `partitions` $\{Collections\}$

The predicate part of *COLLECTIONFORMS* specifies that the sets *UnaryCollections* and *BinaryCollections* partition *Collections*.

Next we describe the fact that BROOM collections may have either set or bag behaviour.

COLLECTIONBEHAVIOURS

$Collections, SetCollections, BagCollections : \mathbb{P}\, OBJID$

$\{SetCollections, BagCollections\}$ `partitions` $\{Collections\}$

The third collection family of figure 3.7 is a simple subcollection relationship between `Collections` and `OrderedCollections` and this is described in the Z-schema *COLLECTIONORDERS*.

$$
\begin{array}{|l}
\textit{COLLECTIONORDERS} \\
\hline
\textit{Collections}, \textit{OrderedCollections} : \mathbb{P}\ \textit{OBJID} \\
\hline
\textit{OrderedCollections} \subseteq \textit{Collections} \\
\hline
\end{array}
$$

These three Z-schemas can be combined into a single Z-schema *COLLECTIONS*. A Z-schema S_1 may be included in a Z-schema S_2 by giving S_1 in the declaration part of S_2. The effect of this is that the declarations of S_1 are considered part of the declarations of S_2 and the predicate part of S_2 is the conjunction of the predicate part of S_2 and the predicate part of S_1. Any variables common to S_1 and S_2 must be of the same type.

Using schema inclusion, we introduce a Z-schema *COLLECTIONS* which includes the previous three Z-schemas. It also describes the final part of figure 3.7 which concerns the naming of collections.

$$
\begin{array}{|l}
\textit{COLLECTIONS} \\
\hline
\textit{COLLECTIONFORMS} \\
\textit{COLLECTIONBEHAVIOURS} \\
\textit{COLLECTIONORDERS} \\
\textit{Names} : \textit{NAME} \rightarrowtail\!\!\!\!\!\!\!+\ \textit{OBJID} \\
\hline
\operatorname{ran} \textit{Names} \subseteq \textit{Collections} \\
\hline
\end{array}
$$

The cardinality constraints associated with relation `Names` are such that it can be represented by a partial function which maps names to collections. (Recall that we allow for unnamed collections which may arise as the result of query evaluation.) Further, a collection can have only one name and therefore this function is injective. $\textit{Names} : \textit{NAME} \rightarrowtail\!\!\!\!\!\!\!+\ \textit{OBJID}$ specifies that *Names* is a partial injective function from *NAME* to *OBJID*. The predicate part of the Z-schema further restricts this function such that the range is a subset of *Collections*.

The specification corresponding to figure 3.7 is complete and we now move on to the description of values given in figure 3.8. The Z-schema *COLLECTIONS* includes a description of `Collections` and `UnaryCollections` and `BinaryCollections`. Moving "up" the classification structure for `Values`, we next introduce a Z-schema which describes the subcollection relationship between `Objects` and `Collections`.

```
┌─ OBJECTS ──────────────────────────────
│ COLLECTIONS
│ Objects : ℙ OBJID
├──────────
│ Collections ⊆ Objects
└────────────────────────────────────────
```

The other type definitions that we require for the value specification are the set of all strings, STRING and the given set of all integers $\mathbb{Z}$.

$[STRING]$

The remaining unary collections of figure 3.8 have members of different types. These member values can be base values such as integers or strings, object identifiers or pairs of values. All of the underlying sets of values are disjoint and we introduce the following union types to correspond to the various required sets of values. Here we include only strings and integers as base values but clearly this could be extended (or restricted) to cater for any particular underlying type system.

$BASEVALUE == STRING \cup \mathbb{Z}$

$ATOMICVALUE == BASEVALUE \cup OBJID$

$VALUE == ATOMICVALUE \cup VALUE \times VALUE$

Then continuing to work up the `Values` classification structure we introduce a Z-schema *ATOMICVALUES*. Note that the set *BaseValues* is intended to be the set of all base values that occur in the database at a particular point in time and not the set of all possible base values.

```
┌─ ATOMICVALUES ─────────────────────────
│ OBJECTS
│ AtomicValues : ℙ ATOMICVALUE
│ BaseValues : ℙ BASEVALUE
├──────────
│ {Objects, BaseValues} partitions {AtomicValues}
└────────────────────────────────────────
```

We are now in a position to complete our Z representation of figure 3.8. The figure describes a classification structure for `Values` and relations on that structure. There are two forms of relations : some are homogeneous in that they relate a collection either to itself or to one of its subcollections while others are heterogeneous in that they relate collections that belong to different classification structures. The relations depicted in figure 3.8 are homogeneous.

In general, the specifications of homogeneous relations are included as part of the Z-schema of the associated classification structure. Relations which are heterogeneous

link different classification structures and correspondingly we introduce a Z-schema for a heterogeneous relation which includes the Z-schemas for the classification structures of its source and target collections.

The Z-schema *VALUES* incorporates the representation of the binary collection `Members` which has bag behaviour. Recall the example of `Members` given in section 3.3, where it represented the membership of a bag collection B and a set collection S with

$$ext(\texttt{Members}) = \langle (B,x),(B,x),(B,y),(B,y),(B,y),(B,z),(S,z),(S,w)\rangle$$

Then `Members` is represented in the Z specification by the bag construct of Z.

In Z, a bag of values of type X is a partial function from X to the set of positive integers $\mathbb{N}_1$. i.e.

$$\text{bag}\, X == X \rightarrow\!\!\!\!\!\!+ \; \mathbb{N}_1$$

Then the above example of collection `Members` can be represented by the Z bag

$$Members : \text{bag}(OBJID \times VALUE)$$

where

$$Members = \{(B,x) \mapsto 2, (B,y) \mapsto 3, (B,z) \mapsto 1, (S,z) \mapsto 1, (S,w) \mapsto 1\}$$

Then dom *Members* is the set of all (collection,value) pairs that occur in the bag *Members*. The collections `UnaryMembers` and `BinaryMembers` have similar representations. The binary collections `Firsts` and `Seconds` are represented by partial functions from pair values to values.

VALUES

OBJECTS
$Values, AtomicValues, PairValues : \mathbb{P}\ VALUE$
$Firsts, Seconds : VALUE \nrightarrow VALUE$
$Members, UnaryMembers, BinaryMembers : bag(OBJID \times VALUE)$

$\mathrm{dom}\ Firsts = PairValues$
$\mathrm{ran}\ Firsts \subseteq Values$
$\mathrm{dom}\ Seconds = PairValues$
$\mathrm{ran}\ Seconds \subseteq Values$
$\{PairValues, AtomicValues\}$ partitions $\{Values\}$
$\mathrm{dom}(\mathrm{dom}\ UnaryMembers) \subseteq UnaryCollections$
$\mathrm{ran}(\mathrm{dom}\ UnaryMembers) \subseteq AtomicValues$
$\mathrm{dom}(\mathrm{dom}\ BinaryMembers) \subseteq BinaryCollections$
$\mathrm{ran}(\mathrm{dom}\ BinaryMembers) \subseteq PairValues$
$\{\mathrm{dom}\ UnaryMembers, \mathrm{dom}\ BinaryMembers\}$ partitions $\{\mathrm{dom}\ Members\}$
$Firsts \in$ noncircular$[VALUE]$
$Seconds \in$ noncircular$[VALUE]$
$\mathrm{dom}\ Members \in$ noncircular$[VALUE]$
$\forall v : VALUE \bullet \neg ((v, v) \in Firsts \lor (v, v) \in Seconds \lor (v, v)\ \mathrm{in}\ Members)$
$\forall C : SetCollections \bullet (Members(\!|\{C\} \lhd \mathrm{dom}\ Members|\!) = \{1\})$

In this schema, the predicate part consists of several predicate expressions and it is assumed that these are conjoined together i.e. the predicate of the schema is the conjunction of each of the predicate statements.

The domains and ranges of *Firsts* and *Seconds* are restricted to the appropriate sets. They are then both specified to be noncircular in that no pair value can have itself as a component.

UnaryMembers and *BinaryMembers* are both bags that map collections to values. The domains of these bags give the set of (collection,value) pairs that appear in the bag. Then dom(dom *UnaryMembers*) is the set of collections that appear as first elements of the pairs of dom *UnaryMembers* and this should be a subset of *UnaryCollections*. Corresponding restrictions are given for the range of *UnaryMembers* and the domain and range of *BinaryMembers*. Given these restrictions and the fact that *UnaryMembers* and *BinaryMembers* form a partition of *Members*, it is straightforward to show that :

$\mathrm{dom}(\mathrm{dom}\ Members) \subseteq Collections$
$\mathrm{ran}(\mathrm{dom}\ Members) \subseteq Values$

Firsts, *Seconds* and *Members* are all relations on *VALUE*. In the case of *Members*, the domain is restricted to *OBJID* which is a subset of *VALUE*. In each case, the

relations should be noncircular and also irreflexive. For example, no collection can be a member of itself and no pair value can have itself has the first member of the pair. We specify the fact that a relation is irreflexive by stating that for any value v, the pair (v, v) cannot be a member of the relation. In the case of *Members*, we check that (v, v) is not a member of the bag using the Z bag membership relation denoted by 'in'.

The final predicate expresses the fact that the number of occurrences of any value in a set collection must be exactly one. Thus for a given set collection C, if we select those member pairs of dom *Members* with first element C, then the image of this set under *Members* should be the singleton set $\{1\}$. The operator $\lhd$ restricts the domain of a relation to a set of values; then

$$\{C\} \lhd \operatorname{dom} Members$$

is the set of all pairs in bag *Members* with first element C. For a relation R and a set S, the relational image $R(\!|S|\!)$ is the set of all values y such that for some $x \in S$, $(x, y) \in R$. Then since *Members* is a mapping from (collection,value) pairs to $\mathbb{N}_1$

$$Members(\!|\{C\} \lhd \operatorname{dom} Members|\!)$$

is a set of positive integers that gives the number of occurrences of pairs in bag *Members* with first element C. As stated above, if C is a set collection, the number of occurrences should always be one and the value of the above expression should equal $\{1\}$.

Having specified the BROOM description of figure 3.8 in the schema *VALUES*, we follow a similar procedure for figure 3.9 to construct a Z-schema for BROOM types.

As stated in the previous section, we are not attempting to specify the details of a particular type system and its type checking requirements. Rather we assume that any value of the database is an instance of one or more types and that these types will be related by some form of subtyping relationship. Here a type is represented simply by a name value.

We first introduce a Z-schema which describes atomic types: an atomic type is either a base type or an object type, and an object type may be a collection type.

ATOMICTYPES

$AtomicTypes, ObjectTypes, BaseTypes,$
$CollectionTypes, UnaryCollTypes, BinaryCollTypes : \mathbb{P}\, NAME$

$\{ObjectTypes, BaseTypes\}$ partitions $\{AtomicTypes\}$
$CollectionTypes \subseteq ObjectTypes$
$\{UnaryCollTypes, BinaryCollTypes\}$ partitions $\{CollectionTypes\}$

The Z-schema for types for figure 3.9 is then very similar to that for values. The main difference is that whereas each collection can have many member values and, indeed, a member may occur more than once in a bag collection, a collection type will have exactly one member type. *MemberTypes*, *UnaryMemTypes* and *BinaryMemTypes* are all partial functions mapping names to names.

TYPES1

$$
\begin{array}{l}
\mathit{ATOMICTYPES} \\
\mathit{Types}, \mathit{PairTypes} : \mathbb{P}\, \mathit{NAME} \\
\mathit{FirstTypes}, \mathit{SecTypes} : \mathit{NAME} \nrightarrow \mathit{NAME} \\
\mathit{MemberTypes}, \mathit{UnaryMemTypes}, \mathit{BinaryMemTypes} : \mathit{NAME} \nrightarrow \mathit{NAME} \\
\hline
\{\mathit{AtomicTypes}, \mathit{PairTypes}\}\ \mathtt{partitions}\ \{\mathit{Types}\} \\
\mathrm{dom}\ \mathit{FirstTypes} = \mathit{PairTypes} \\
\mathrm{ran}\ \mathit{FirstTypes} \subseteq \mathit{Types} \\
\mathrm{dom}\ \mathit{SecTypes} = \mathit{PairTypes} \\
\mathrm{ran}\ \mathit{SecTypes} \subseteq \mathit{Types} \\
\{\mathit{UnaryMemberTypes}, \mathit{BinaryMemTypes}\}\ \mathtt{partitions}\ \{\mathit{MemberTypes}\} \\
\mathrm{dom}\ \mathit{UnaryMemTypes} = \mathit{UnaryCollTypes} \\
\mathrm{ran}\ \mathit{UnaryMemTypes} \subseteq \mathit{AtomicTypes} \\
\mathrm{dom}\ \mathit{BinaryMemTypes} = \mathit{BinaryCollTypes} \\
\mathrm{ran}\ \mathit{BinaryMemTypes} \subseteq \mathit{PairTypes} \\
\mathit{FirstTypes} \in \mathtt{noncircular}[\mathit{NAME}] \\
\mathit{SecTypes} \in \mathtt{noncircular}[\mathit{NAME}] \\
\mathit{MemberTypes} \in \mathtt{noncircular}[\mathit{NAME}] \\
\forall t : \mathit{NAMES} \bullet \\
\qquad \neg\, ((t, t) \in \mathit{FirstTypes} \lor (t, t) \in \mathit{SecTypes} \lor (t, t) \in \mathit{MemberTypes})
\end{array}
$$

We have assumed a subtyping relationship on types as described in figure 3.10 and we must include this in the specification for types. We therefore named the above Z-schema *TYPES1* and include it in the Z-schema *TYPES* which completes the description of types by including the relation *Subtypes* on *Types* to represent the BROOM relation `Subtypes`.

$$
\begin{array}{l}
\textit{TYPES} \\
\hline
\textit{TYPES1} \\
\textit{Subtypes} : \textit{NAME} \leftrightarrow \textit{NAME} \\
\hline
\operatorname{dom} \textit{Subtypes} \subseteq \textit{Types} \\
\operatorname{ran} \textit{Subtypes} \subseteq \textit{Types} \\
\textit{Subtypes} \in \texttt{noncircular}[\textit{NAME}] \\
\forall (x, y) : \textit{Subtypes} \bullet \\
\quad ((x \in \textit{AtomicTypes} \Leftrightarrow y \in \textit{AtomicTypes}) \land \\
\quad (x \in \textit{PairTypes} \Leftrightarrow y \in \textit{PairTypes}) \land \\
\quad (x \in \textit{BaseTypes} \Leftrightarrow y \in \textit{BaseTypes}) \land \\
\quad (x \in \textit{ObjectTypes} \Leftrightarrow y \in \textit{ObjectTypes}) \land \\
\quad (x \in \textit{CollectionTypes} \Leftrightarrow y \in \textit{CollectionTypes}) \land \\
\quad (x \in \textit{UnaryCollTypes} \Leftrightarrow y \in \textit{UnaryCollTypes}) \land \\
\quad (x \in \textit{BinaryCollTypes} \Leftrightarrow y \in \textit{BinaryCollTypes}))
\end{array}
$$

If a type t_1 is a subtype of t_2 then, t_1 and t_2 must be the same kind of types. For example, if t_1 is an object type, then t_2 must also be an object type. *TYPES* includes a predicate to ensure that this form of subtyping consistency is satisfied.

Recall from the previous section that `Subtypes` represents explicit subtyping declarations from which the subtyping relation $\leq_t$ can be derived.

Given the Z-schemas *VALUES* and *TYPES*, the Z-schema *INSTANCES* specifies the *Instances* relation which maps values to their types.

$$
\begin{array}{l}
\textit{INSTANCES} \\
\hline
\textit{VALUES} \\
\textit{TYPES} \\
\textit{Instances} : \textit{VALUE} \leftrightarrow \textit{NAME} \\
\hline
\operatorname{dom} \textit{Instances} = \textit{Values} \\
\operatorname{ran} \textit{Instances} \subseteq \textit{Types} \\
\forall (x, y) : \textit{Instances} \bullet \\
\quad ((x \in \textit{AtomicValues} \Leftrightarrow y \in \textit{AtomicTypes}) \land \\
\quad (x \in \textit{PairValues} \Leftrightarrow y \in \textit{PairTypes}) \land \\
\quad (x \in \textit{BaseValues} \Leftrightarrow y \in \textit{BaseTypes}) \land \\
\quad (x \in \textit{Objects} \Leftrightarrow y \in \textit{ObjectTypes}) \land \\
\quad (x \in \textit{Collections} \Leftrightarrow y \in \textit{CollectionTypes}) \land \\
\quad (x \in \textit{UnaryCollections} \Leftrightarrow y \in \textit{UnaryCollTypes}) \land \\
\quad (x \in \textit{BinaryCollections} \Leftrightarrow y \in \textit{BinaryCollTypes}))
\end{array}
$$

The *Instances* relation also has to respect the classification structures that it relates. Thus, similar kinds of values must be instances of the corresponding kinds of types: an object value must be an instance of an object type, and so on.

Now the general approach for transforming the figures of the metacircular description of BROOM into Z-schemas has been shown, it should be straightforward to see how Z-schemas for the remaining figures can be constructed.

The Z-schema for collection families, as described in figure 3.12, is constructed in three stages. The first deals with the various forms of collection family.

$$
\begin{array}{|l}
\textit{FAMILYCONSTRAINTS} \\
\hline
\textit{CollectionFamilies}, \textit{DisjointFamilies}, \textit{CoverFamilies}, \\
\textit{PartitionFamilies}, \textit{IntersectFamilies} : \mathbb{P}\, \textit{OBJID} \\
\hline
\textit{DisjointFamilies} \subseteq \textit{CollectionFamilies} \\
\textit{CoverFamilies} \subseteq \textit{CollectionFamilies} \\
\textit{IntersectFamilies} \cap \textit{DisjointFamilies} = \varnothing \\
\textit{IntersectFamilies} \cap \textit{CoverFamilies} = \varnothing \\
\{\textit{PartitionFamilies}\}\ \mathtt{intersects}\ \{\textit{DisjointFamilies}, \textit{CoverFamilies}\} \\
\hline
\end{array}
$$

The sets *IntersectFamilies*, *DisjointFamilies* and *CoverFamilies* are all subsets of *CollectionFamilies*. *IntersectFamilies* and *DisjointFamilies* are non-overlapping, as are *IntersectFamilies* and *CoverFamilies*. The set *PartitionFamilies* consists of those collection families which are members of both *DisjointFamilies* and *CoverFamilies*.

Next we describe the structure of collection families in terms of parent and child collections.

$$
\begin{array}{|l}
\textit{FAMILYMEMBERS} \\
\hline
\textit{OBJECTS} \\
\textit{TYPES} \\
\textit{FAMILYCONSTRAINTS} \\
\textit{Contains}, \textit{Parents}, \textit{Children}, \textit{Subcollections} : \textit{OBJID} \leftrightarrow \textit{OBJID} \\
\hline
\textit{CollectionFamilies} \subseteq \textit{Objects} \\
\mathtt{disjoint}\ \{\textit{Collections}, \textit{CollectionFamilies}\} \\
\mathrm{dom}\ \textit{Contains} = \textit{CollectionFamilies} \\
\mathrm{ran}\ \textit{Contains} \subseteq \textit{Collections} \\
\{\textit{Parents}, \textit{Children}\}\ \mathtt{partitions}\ \{\textit{Contains}\} \\
\mathrm{dom}\ \textit{Parents} = \textit{CollectionFamilies} \\
\mathrm{dom}\ \textit{Children} = \textit{CollectionFamilies} \\
\textit{SubCollections} = \textit{Parents}^{-1} \fatsemi \textit{Children} \\
\textit{SubCollections} \in \mathtt{noncircular}[\textit{OBJID}] \\
\forall (C_1, C_2) : \textit{Subcollections} \bullet \\
\qquad (\textit{MemberTypes}\ C_1, \textit{MemberTypes}\ C_2) \in \textit{Subtypes}^{*} \\
\hline
\end{array}
$$

The *SubCollections* is defined in terms of the *Parents* and *Children* collections and must be noncircular in that a collection cannot be declared as a subcollection of itself.

The Z-schema *FAMILYMEMBERS* specifies the various forms of collection family and their associated constraints. The next stage is to specify what it means for these constraints to be satisfied. This places conditions on the values of the collections in a family and we first introduce a function `elements` which, for a given collection, will return a bag of the elements of that collection.

ELEMENTS

VALUES

$\texttt{elements} : OBJID \rightarrow\!\!\!\!\!+ \mathbb{P}\ VALUE$

$\forall C : Collections \bullet \texttt{elements}\ C = \{(x, n) \mid ((C, x), n) \in Members\}$

The Z-schema *FAMILIES* specifies that the collection family constraints are satisfied by the collections in the family.

FAMILIES

VALUES

FAMILYMEMBERS

FAMILYCONSTRAINTS

ELEMENTS

$\forall F : CollectionFamilies \bullet$
$(CS = \{\texttt{elements}\ C \mid (F, C) \in Children\} \wedge$
$PS = \{\texttt{elements}\ C \mid (F, C) \in Parents\} \wedge$
$\forall X \in CS \bullet \forall Y \in PS \bullet$
$(X \text{ subbag } Y \wedge$
$(F \in DisjointFamilies \Rightarrow$
$\quad \forall X \in CS \bullet \forall Y \in PS \bullet \neg\ \exists x \bullet (x \in X \wedge x \in Y)) \wedge$
$(F \in CoverFamilies \Rightarrow$
$\quad \forall Y \in PS \bullet \forall x \in Y.\text{count } Y\ x = max\{\text{count } X\ x \mid X \in CS\}) \wedge$
$(F \in IntersectFamilies \Rightarrow$
$\quad \forall X \in CS \bullet \forall x \in X.\text{count } X\ x = min\{\text{count } Y\ x \mid Y \in PS\})))$

For a given family, *CS* is the set comprised of the bags of elements of each child collection in the family. Likewise, *PS* is the set comprised of all the bags of elements of each parent collection. Then the schema specifies that *PS* and *CS* satisfy the constraints on the family. The conditions correspond to the definition of collection family given in section 3.2.1. It is not necessary to give a condition for F being a member of *PartitionFamilies* since this is covered by it being a member of both *DisjointFamilies* and *CoverFamilies*. Note that this specification uses the "subbag" predicate on Z bags. It also uses "count" which for a given bag and element gives the number of occurrences of the element in that bag. These are both as defined in the Z toolkit given in [Dil90].

The Z-schema for relations, as given in figure 3.13 is also constructed in three stages. The first of these represents the collection `Relations` as a set of objects which are binary collections and have associated source and target collections.

RELATIONLINKS

$$\begin{array}{l}
\mathit{Collections}, \mathit{BinaryCollections}, \mathit{Relations} : \mathbb{P}\, \mathit{OBJID} \\
\mathit{Sources}, \mathit{Targets} : \mathit{OBJID} \nrightarrow \mathit{OBJID} \\
\hline
\mathit{Relations} \subseteq \mathit{BinaryCollections} \\
\mathrm{dom}\, \mathit{Sources} = \mathit{Relations} \\
\mathrm{ran}\, \mathit{Sources} \subseteq \mathit{Collections} \\
\mathrm{dom}\, \mathit{Targets} = \mathit{Relations} \\
\mathrm{ran}\, \mathit{Targets} \subseteq \mathit{Collections}
\end{array}$$

The next stage is to represent the cardinalities associated with the sources and targets of relations.

RELATIONCARDINALITIES

$$\begin{array}{l}
\mathit{RELATIONLINKS} \\
\mathit{SourceMin}, \mathit{SourceMax}, \mathit{TargetMin}, \mathit{TargetMax} : \mathit{OBJID} \nrightarrow \mathbb{N}_1 \\
\hline
\mathrm{dom}\, \mathit{SourceMin} \subseteq \mathit{Relations} \\
\mathrm{dom}\, \mathit{SourceMax} \subseteq \mathit{Relations} \\
\mathrm{dom}\, \mathit{TargetMin} \subseteq \mathit{Relations} \\
\mathrm{dom}\, \mathit{TargetMax} \subseteq \mathit{Relations} \\
\forall R : \mathit{Relations} \bullet \\
(\exists\, n_1, n_2 : \mathbb{N}_1 \bullet \\
\quad ((R, n_1) \in \mathit{SourceMin} \land (R, n_2) \in \mathit{SourceMax}) \Rightarrow n_1 \leq n_2)) \land \\
(\exists\, n_1, n_2 : \mathbb{N}_1 \bullet \\
\quad ((R, n_1) \in \mathit{TargetMin} \land (R, n_2) \in \mathit{TargetMax}) \Rightarrow n_1 \leq n_2)))
\end{array}$$

The minimum cardinality constraint on the source of a relation is represented as a partial function from the relation to a positive integer. Similar representations are given for the other forms of cardinality constraints. If minimum and maximum cardinality constraints are given for the source or target of a relation, then the minimum value must be less than or equal to the maximum value.

As with the case of collection families, we now introduce a Z-schema for relations which specifies that the actual values in a given relation satisfy the constraints associated with that relation. From these Z-schemas, we construct a Z-schema *RELATIONS* for relations which corresponds to figure 3.13.

RELATIONS
VALUES
RELATIONCARDINALITIES
ELEMENTS

$\forall R : Relations \bullet$
$S = \texttt{elements}(Sources(R)) \wedge T = \texttt{elements}(Targets(R)) \wedge$
$(\forall x, y : VALUE \bullet$
$((x, y) \in \texttt{elements}\ R \Rightarrow (x \in S \wedge y \in T)) \wedge$
$\exists n \in \mathbb{N}_1 \bullet (SourceMin\ R = n \Rightarrow$
$\quad \text{count dombag}(\texttt{elements}\ R)\ x \geq (\text{count}\ S\ x) * n) \wedge$
$\exists n \in \mathbb{N}_1 \bullet (SourceMax\ R = n \Rightarrow$
$\quad \text{count dombag}(\texttt{elements}\ R)\ x \leq (\text{count}\ S\ x) * n) \wedge$
$\exists n \in \mathbb{N}_1 \bullet (TargetMin\ R = n \Rightarrow$
$\quad \text{count dombag}(\texttt{elements}\ R)\ x \geq (\text{count}\ T\ x) * n) \wedge$
$\exists n \in \mathbb{N}_1 \bullet (TargetMax\ R = n \Rightarrow$
$\quad \text{count dombag}(\texttt{elements}\ R)\ x \leq (\text{count}\ T\ x) * n))$

The first condition in the predicate part of *RELATIONS* specifies that for each pair in a relation, the first element must belong to the bag of elements of the source of that relation, and the second element must belong to the bag of elements of the target. The next condition specifies that if a relation has a minimum specified for the source of the relation, then the degree of participation of elements of the source must be greater than that minimum value. Note that the condition must cater for the case in which either the relation or its source is a bag collection. The interpretation of cardinality constraints in the case of bags can be explained as follows. If the minimum for the source is specified as 1, then this means that each occurrence of any element must appear at least once in the relation. Thus, if element x occurs three times in the source, then it must appear in at least three pairs of the relation. The condition uses the function on bags "dombag" which gives the domain of a bag; this is defined to be the bag of elements appearing as first elements of pairs in the bag and the number of occurrences is equal to the total number of pairs in which the element occurs. Unfortunately, this is not defined as part of the standard Z mathematical toolkit. However, it is defined in Chapter 5 as one of the operations of the collection algebra and we therefore omit its definition here. The predicate conditions for the other cardinality constraints are defined in a similar way.

Having transformed each of the figures of the metacircular description of BROOM into Z-schemas, we now combine these into a schema for a BROOM database. This Z-schema *DB* is simply an inclusion of the three Z-schemas *INSTANCES*, *FAMILIES* and *RELATIONS*.

```
┌─ DB ──────────────────────────
│ INSTANCES
│ FAMILIES
│ RELATIONS
└───────────────────────────────
```

Z-schemas can be used either to specify states or state transitions. In this section, we have been concerned only with specifying the structural part of the BROOM model and therefore only with the description of valid states. In chapters 5 and 6 we will be concerned with operations on a database and therefore will introduce Z-schemas which specify state transitions. However, we will include here the simplest and most basic example of a state transition which corresponds to initialisation of a database.

```
┌─ INITDB' ─────────────────────
│ DB'
├──────────
│ Values' = ∅
│ Types' = ∅
└───────────────────────────────
```

By convention, it is usual in the specification of state transitions to use the priming of variables to denote after states and the initialisation of a system is considered as an operation with only after states.

The initialisation schema is very simple as it requires only that the sets *Values* and *Types* be initialised to the empty sets. However, it is simple to show that from the predicates in schema *DB* it follows that all other sets must also be empty.

Chapter 4

Semantic Modelling in BROOM

In this chapter, we examine the semantic modelling capabilities of BROOM with the intention of providing a comparison with existing semantic data models and also some guidance as to how to use the various BROOM constructs. Here, we include the graph data models discussed in section 2.3 in the category of semantic data models.

The semantic data models were specifically developed to narrow the gap between an application reality and its conceptual model through the provision of powerful mechanisms for the representation of data relationships that arise frequently in database applications. It is therefore appropriate to consider the basic constructs of semantic models and how these are supported in the BROOM model.

Semantic models provide constructs for modelling the two basic notions of entities and relationships and typically some means of supporting the three abstraction mechanisms of aggregation, association and generalisation.

They are entity-based rather than value-based in that an individual entity of the application domain will have a unique direct representation in the database independent of the values of its attributes and any changes to these values that may occur during the lifetime of the entity. For example, in the IFO model [AH87], an entity is represented as an abstract type which essentially serves the same role as "entity" in the Entity-Relationship [Che76] or Functional Data Models [Shi81]. Implementations of these models will use the idea of surrogates or internal identifiers to uniquely denote entities.

Object data models are certainly entity-based in that each entity of the application domain is represented by an object with a unique identifier. A subtle difference between the semantic data models and the object data models is that while in both systems the values of these unique identifiers are hidden to the user, the user of an object data model is certainly aware of the existence of these identifiers and frequently

regards them as object references (or pointers). This is significant in terms of object identity and/or equality. The reason for this difference is likely to be a consequence of the difference in origin and usage rather than something fundamental. Many of the object data models have their origins in object-oriented programming languages and there has been a lot of emphasis on the operational aspects of the models and in providing implementations of them. On the other hand, the semantic data models tend to have their origins in conceptual modelling and in many cases they have been used as an intermediary stage in the database design process. While some database management systems based on these models have been developed, they are mainly prototypes. In general, the semantic data models place less emphasis on the operational part of the model.

The second fundamental notion in semantic models is that of the relationship. In the Entity-Relationship model, relationships are given equal standing with entities in that both are considered to be key modelling concepts and they appear in a conceptual model at the same level of abstraction. In other words, it is neither the case that relationships are considered as part of the participating entities, nor, that these entities are considered part of the relationships in which they participate. However, this balance between the notions of entity and relationship is not present in all semantic data models and indeed the way in which the balance is tipped may be used as a basis for the categorisation of semantic models. For example, a number of semantic models (e.g. SDM [HM81] and TAXIS [MBW80]) tend to place the emphasis on entities and certain system supported relationships rather than general user-defined relationships. These system supported relationships are *part-of*, *member-of* and *isa*, which correspond to the abstractions of aggregation, association and generalisation, respectively. In these models, the user-defined relationships become internalised. By this, we mean that a user-defined relationship is regarded as a property of an entity rather than being at the same level of abstraction as that entity. On the other hand, a number of models have focussed on the notion of relationships and view everything in terms of relationships. The Functional Data Model of Shipman [Shi81] and the Semantic Binary Data Model of Abrial [Abr74] are two such examples. In section 2.3, we highlighted this distinction by referring to these as graph data models.

To assess the semantic data modelling capabilities of the BROOM model, it is important to examine in detail how both relationships and the abstractions of aggregation, association and generalisation can be modelled in BROOM. We do this in the following two sections: the first section examines relationships and the second section the data modelling abstractions.

We note that there are other features of semantic data models that should be considered in a general discussion of these models. For example, they vary in terms of the extent to which they support dynamic modelling as opposed to static modelling. In object-oriented systems, an object may have associated behaviour as specified by its methods. Then an object is not static and the dynamic aspects of a system may be modelled in terms of active objects. It is for this reason that we refer to BROOM

as a structural model and not a static model. Whether it will be static or dynamic depends on the nature of the underlying type system and whether objects are active objects in the sense of object-oriented programming or static objects as is frequently the case in semantic data modelling. A comprehensive discussion of semantic data modelling and a survey of semantic data models is provided in the articles by Peckham and Maryanski [PM88] and Hull and King [HK87].

4.1 Relationships

If semantic models were categorised according to the relative standing of entities and relationships, then the BROOM model would belong to the same category as the Entity-Relationship model and its variants. The model was designed to redress the imbalance between the notions of entities and relationships that existed in most object data models. In these models, the focus was firmly fixed on entities with relationships internalised as properties of entities and represented by methods which return object references. Again, the reason for this is probably the origins of object data models in object-oriented programming languages.

As discussed in Chapter 1, this internalisation of relationships has a number of disadvantages. These arise from the fact that the relationships in which an entity participates are regarded as part of the entity rather than as separate conceptual notions. As a result, there is no clear distinction between those properties that characterise an entity from those that represent external associations of that entity. For example, a person may be characterised by properties such as name, address and birthdate. A book may be characterised by properties such as title, subject and author. Now the fact that there is an association at a particular point in time between a person and a book in that the person has the book on loan from the library is neither a characteristic property of the person nor a characteristic property of the book. In other words, the person and the book may be regarded as independent entities which happen to be related. Therefore such associations are external to the entities involved and should not be considered as properties of the entities.

Distinguishing between the characteristic properties of entities and their external associations is beneficial in a number of ways. It means that the relations between entity categories appear at the same level of abstraction as those categories. This helps the user visualise the structure of a system in terms of classification structures and links between those structures. Thus a conceptual model can be decomposed naturally into the various classification structures. This corresponds to Rumbaugh's claim that relationships provide a useful means of partitioning systems into subsystems [Rum87]. In turn, the clear distinction between entities and their external associations encourages reusability. Consider the simple example above of a person entity. By divorcing the person entity from its relationship to a book entity, it is

more likely that the characterisation of a person entity corresponds to that required in other application systems.

The clean separation of entities from relationships means that the relationships may be viewed and manipulated as separate logical units. This is important in providing the capability to attach semantic information to relationships. The facility provided in some semantic data models of being able to specify that a property of an object is in some way an inverse of the property of another object is both cumbersome and prone to error.

Recently, it has been recognised by a number of researchers that the internalisation of relationships is a major drawback of object data models and as a result there are now a number of proposals to extend object data models with direct support for relationships. These include [AGO91],[Bra90],[DG90], [NQZ90] and [Rum87].

In chapter 3, we described the use of binary collections to represent relationships. A relationship is in fact represented by a binary collection along with the associated information that specifies source and target collections and their respective cardinality constraints. One issue that has to be addressed is whether it is sufficient to support only binary relationships or if relationships of higher degree should also be supported.

Generally, we would make the case that the introduction of higher degree relationships adds to the complexity of models and actually confuses the users. It can be quite difficult to comprehend the semantics of even three or four way relationships and their associated cardinality constraints. Rumbaugh [Rum87] also excludes higher degree relationships for this reason. However, he does make the case for qualified relations: these can be considered as special forms of ternary relations and he claims that these are the most useful form. Rumbaugh states:

> "Qualified relations occur when there is a set of names, or some other set of qualifiers, that serves to distinguish the target elements in a one-to-many or many-to-many relation."

His example is that of a file system in which a directory contains a number of files: to distinguish a particular file, a directory must be "qualified" by a name. He therefore introduces a special form of ternary relation - the qualified relation - to represent the relationship between directory-name pairs and files. This very description of qualified relations highlights the fact that this situation can be modelled by a binary collection which maps pairs of directory and name values to file values. We show in figure 4.1 how this could be represented in the BROOM model.

In a similar way, we could model the common example of student course grades. We want to record which courses a student attends and this can be represented

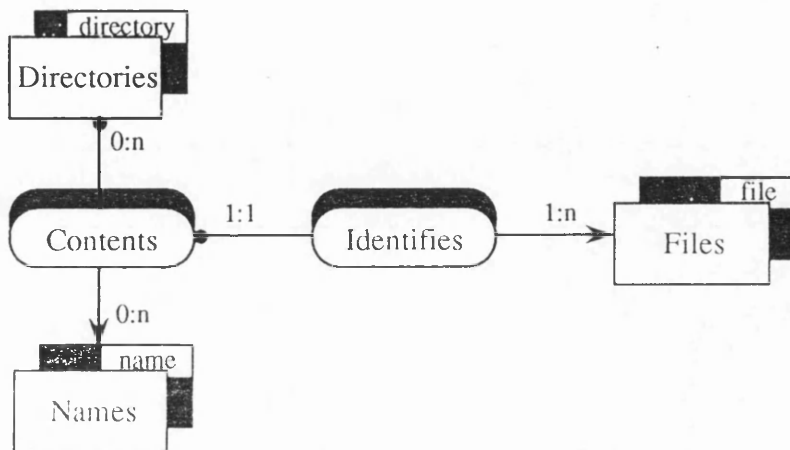

Figure 4.1: Modelling a File System

by the binary collection `Attends`. Further, we want to record the assessments of a particular student on a particular course: this can be represented by the binary collection `Attains` which maps student-course pairs to assessments as shown in figure 4.2.

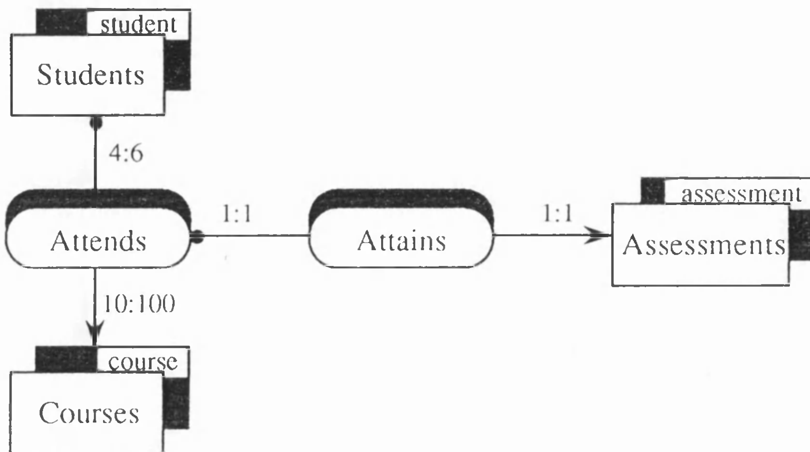

Figure 4.2: Student Course Assessments

It is interesting to note the parallel between the representation of these ternary relationships in BROOM and their representation in the Functional Data Model (FDM) [Shi81]. In FDM everything is represented in terms of functions where a function can return an atomic value or a set value. The above example of figure 4.2 could be

represented as:

`attains(student,course)-> assessment`

which is a function that takes a pair of arguments and returns an assessment. In this way, a ternary relationship is not modelled as a three way symmetrical relationship but rather as a relationship between student-course pairs and assessments.

The model in figure 4.2 assumes that each assessment is represented by a single assessment object (or record) which might record such things as the date, grade and marker. It is a general rule in the use of BROOM that if there is a lot of information to be recorded about relationships then an appropriate object type should be created with the required attributes. One way of considering this is that if a relationship has attributes then really there is a hidden entity and that entity should be made apparent. Of course, this general rule applies not only to the BROOM model but to many other models. The problem of deciding whether to consider something as an entity or as a relationship is well known in data modelling.

If however, only a simple attribute is to be recorded then this can be modelled by a binary collection mapping onto a set of base values. Consider the simple library example of figure 4.3.

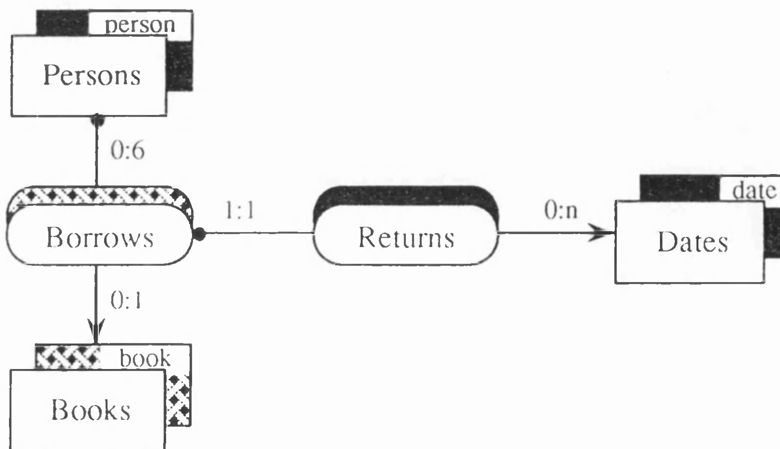

Figure 4.3: Library Example

The library users are modelled by the collection `Persons` and the books in the library by the bag collection `Books`. Note that `Books` is a bag collection which means that the library may have multiple copies of a book. The `Borrows` collection records the books on loan and it is also a bag indicating that the same person can have more than one of the same book on loan; this would be prohibited by making `Borrows` a

set collection. Then the binary collection `Returns` is used to record the date that the book is due. We assume here that `date` is a base type of the system.

In summary, we consider that the support for modelling binary relationships is adequate. Indeed we consider that it is advantageous to have only binary relationships as it forces the designer to be clear about the semantics of the relationship they are trying to model. These binary relationships should be conceived by the user as links or connections between collections. Thus it makes sense to join links to form new longer links and to invert a link to get a link in the other direction.

What should be avoided is confusion between relations as exist in BROOM and those of the relational model. The relations of BROOM are intended solely for the purpose of modelling relationships between entities. However, the relation is the only construct of the relational model and is therefore overworked in that it is used to represent entities, their attributes and relationships and there is no clear distinction among these notions. This has certain advantages for the model in terms of its simplicity but is the reason for its judged lack of semantic content. It is therefore unwise to attempt to support the notion of relationships in object data models by incorporating a notion of relation that is similar to that of the relational model as this then blurs the distinction between the fundamental semantic constructs.

We are advocating direct support for relationships at the logical level by the inclusion in the data model of a separate logical construct. This does not assume any particular physical representation of relationships. It is even possible that such a logical construct be represented at the physical level by embedded pointers. However, it is worth noting that there are also advantages to be gained at the physical level by having a direct representation of relationships thereby avoiding unnecessary pointer chasing. It is frequently the case that one wishes to use relationships to establish links from one part of the database structure to another without examining intervening objects. This has implications in terms of requirements of mapping large objects into memory purely to use them as a means of linking to other objects.

We stress that these physical considerations are incidental in that the main point is that relationships are logical units. This does not say anything about their underlying physical representation. Indeed one major factor in the design of BROOM was to move from the physical to the logical: too many object data models are at too low a level in that they require knowledge of the physical representation of data. Recall that this was considered the main drawback of the hierarchical and network data models and the introduction of support for physical data independence was welcomed: it is a pity that many object-oriented database management systems have taken a step backwards in this respect.

4.2 Aggregation, Association and Generalisation

In this section, we begin by introducing the data modelling abstractions of aggregation, association and generalisation as described in many papers and texts. We then go on to examine each of these abstractions in turn to consider how they relate to each other and how they are supported in the BROOM model.

Knowledge representation techniques have had and, continue to have, a strong influence on data modelling. In particular, the use of semantic nets [Qui68], [Rap68] in Artificial Intelligence influenced Abrial's Semantic Binary Model [Abr74] which was one of the earliest semantic data models.

Semantic nets use the *instance-of*, *part-of* and *isa* relationships on which are based the data modelling abstractions of classification, aggregation and generalisation, respectively. As discussed in section 2.3, the main difference between semantic nets and data models is the fact that semantic nets combine data and metadata whereas in most data models there is a clear separation between the two. This means that the *instance-of* relationship of semantic nets is not supported in data models. Rather the connection of particulars to concepts is under the control of the database management system.

The abstraction process of generalisation was discussed in some detail in chapter 2. It is concerned with the introduction of concepts through the classification of entities in terms of entity categories. A classification structure specifies which entity categories are generalisations of other categories by means of *isa* relationships. For two categories C_1 and C_2, C_1 *isa* C_2 specifies that C_2 is a generalisation of C_1.

Aggregation is concerned with the structure of objects in that it allows higher-level objects to be constructed out of lower-level objects by abstracting away the detail of these lower level objects at the higher level.

The data modelling abstraction of association was introduced by Brodie [Bro81] and it is also supported directly in the IFO model [AH87]. Association is concerned with forming higher-level objects out of groups of similar low-level objects. Thus in a sense it is similar to aggregation but it differs in that whereas the structure of an object formed through aggregation comprises a fixed number of variable components, the structure of an object formed through association comprises a variable number of similar components.

From these descriptions, it should be apparent that, in the BROOM model, the abstractions of aggregation and association are concerned with the introduction of types while the abstraction of generalisation is concerned with the introduction of collections. While this general statement is true, it is an over-simplification of the issues. For we consider these abstractions as corresponding to general cognitive processes and

not simply as constructs of a given semantic data model. We therefore think them worthy of careful consideration in terms of what they are trying to model and the implications of selecting particular constructs in the modelling process. This contrasts with the approach of Hull and King [HK87] who, for the purposes of their survey, adopt very restrictive definitions of aggregation, association and generalisation in terms of particular language constructs.

We now examine each of these abstractions in turn and show how similar situations could be modelled in different ways and the choice of representation depends both on the requirements of the system and the modeller's view of reality.

Aggregation

Aggregation is a process of abstraction which allows an entity to be viewed at a particular level of detail. For example, the average person may wish to regard a car as a single entity and not concern themselves with the details of all the parts that go to make up that car. This abstraction process corresponds to the introduction of higher-level entities which comprise lower-level entities. Then an entity may be represented by a complex object which is an aggregation of its component parts and these in turn may themselves be complex objects.

In an object-oriented model, aggregation is supported by complex types such as record and object types. For example, in our university database introduced in Chapter 3, we had a record type for address as given in figure 4.4.

```
record type address
            street : string ;
            city : string
end ;
```

Figure 4.4: A Record Aggregation

In many semantic data models, the same schema diagram includes descriptions of both the component entities and the encompassing entity. This means that different levels of abstraction are not distinguished graphically. Both levels appear as nodes of the same graph. We prefer to have a clear separation between different levels of abstraction and to encourage designers to avoid relationships that cross these levels of abstraction.

To illustrate this consider the representation of a course in our university example. Assume that a course has a single lecturer and has a number of recommended text-

books. Then a course could be considered as an aggregation of component parts as indicated by the object definition given in figure 4.5. Thus, for a given course, the set of recommended textbooks and the lecturer are considered to be "part-of" that course.

```
object type course
        title :  string ;
        texts :  set of book ;
        lecturer :  person
end ;
```

Figure 4.5: An Object Aggregation

If we represent courses in this way, then we should view a course entity at a higher level of abstraction than either a textbook or lecturer entity and, preferably, should not mix the two by having user-defined relationships which span the two levels. In other words, textbooks and lecturers should exist only in the context of courses. This means that the existence of textbooks and lecturers is dependent upon the existence of courses and further that applications should require access to textbooks and lecturers only through the courses of which they form a part.

Now consider the situation where we have applications that require access to lecturers either directly or through entities other than courses. Then lecturers should not be regarded as components of courses but rather as separate entities of equal standing which are related to courses. In addition, if we are interested in books in contexts other than courses, such as considering books written by staff, then book entities should also be at the same level of abstraction as courses. An appropriate BROOM model is given in figure 4.6.

Then aggregation corresponds to the *part-of* relationship and should only be used in cases where an entity is considered to be a part of another entity and should not be used to model relationships between free standing entities.

Association

Association forms higher-level objects from groups of similar lower-level objects. In some sense, it is similar to aggregation in that it hides the internal structure of the higher-level object. In another sense, it is also similar to generalisation as the resulting higher-level object is a group of objects of the same type. Here lies the crux of the issue for it is important to be clear about exactly what is trying to be achieved

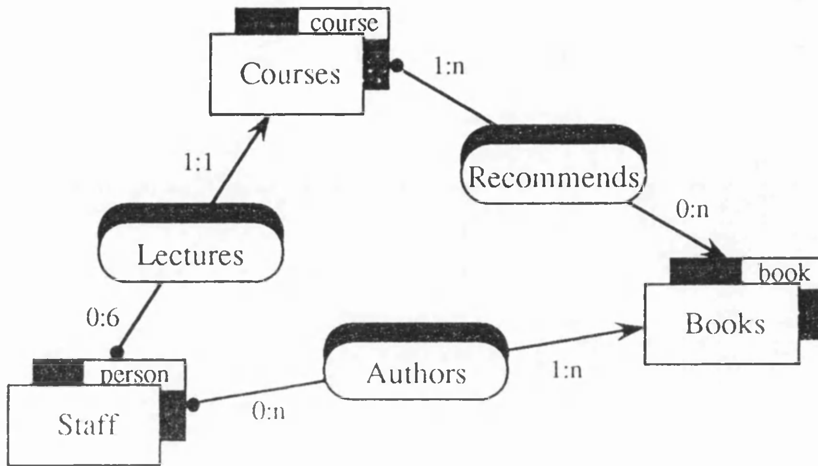

Figure 4.6: Using Relationships instead of Aggregates

by the formation of this grouping. This is best examined through consideration of an example.

An example of association that is frequently given is that of forming a `class` object out of a group of `student` objects. As with aggregation, it is important to consider whether one is really trying to move to a higher level of abstraction in which `class` objects are visible and in which `student` objects are not. Alternatively, it may be the case that one simply wishes to be able to access with ease the group of students associated with a particular course. If the latter is the case, then the situation should be modelled in terms of collections `Students` and `Courses` with a relation `Attends` that links them as shown in figure 4.7.

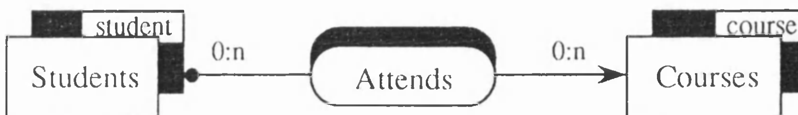

Figure 4.7: Association through Relationships

Then for a given course x, the set of students attending x is easily determined by selecting all those pairs from `Attends` such that the second member is x.

If on the other hand, one really wishes to move to a higher level of abstraction in which `class` objects appear and `student` objects are hidden, then this can be achieved through aggregation and the fact that collections can appear as components of objects. This is illustrated in figure 4.8 and it can be seen that this is similar

to the example given in figure 4.5 as an example of aggregation. Other component properties of `class` have been specified although, of course, these could have been omitted.

```
object type class
          course :  string;
          attendees :  set of student;
          representative :  student
end;
```

Figure 4.8: Classes as Entities

In our opinion, while aggregation and generalisation are fundamental modelling abstractions, association is not. Certainly, we do not dispute the need to form groupings of entities, but rather question the fact that this process is truly different from the others. It seems that it is necessary to introduce it in some data models, not because it corresponds to any particular cognitive process, but to overcome restrictions of these models. Thus in some models which couple the notions of typing and classification, association may be introduced to compensate for the fact that there is no way to form collections of similar objects other than through the introduction of subtypes. For example, SDM [HM81] introduces a grouping construct to form groups of objects of the same type. Other models which use functions to model relationships may use association in order that the function relates an object of the source to a group of objects of the target. An example of such a model is IFO [HK87]. Thus association is introduced to compensate for the fact that relationships are modelled by functions rather then relations.

In the BROOM model, classification and typing are divorced and objects of the same type may be freely grouped into collections. In addition, relationships are represented by relations rather than functions and hence one-to-many and many-to-many relationships can be modelled directly. The remaining form of grouping that occurs in BROOM is in the set valued attributes of objects such as `attendees` of figure 4.8. This raises the question of whether a set valued attribute corresponds to a collection value. This is an open question which is left to the designer of any particular realisation of the collection model. If the underlying type system supports bulk types such as set and bag, then clearly, object attributes may be of these types. If however, the underlying type system does not support such bulk types, then set and bag collections may be used in their place. In this case, such an attribute value would be an object and have the additional semantics associated with collections: it may be that this is more heavyweight than required. However, in either case, an attribute value can be a collection if desired and hence collections can be components of objects in terms of aggregation.

Generalisation

We take the view that the abstraction of generalisation is concerned with the cognitive process of introducing general concepts. These general concepts may either be generalisations of particulars or of existing concepts. Then the abstraction process of classification as discussed in the semantic data modelling literature (e.g. [PM88], [HK87]) is the special case of generalising from particulars to concepts. Since the whole abstraction process is about the formation of classification structures, it might be preferable to consider both abstractions under the heading of classification; however, in accordance with common usage in the data modelling literature, we will refer to both of them under the heading of generalisation.

The process of generalisation has been discussed at length in previous chapters and a number of examples of its use have been given. Therefore here we shall focus on three points. The first is to stress the fact that generalisation can be applied to relationships as well as entity categories; the second, is to examine the different forms of generalisation supported in some semantic data models; and the third is to consider support for multiple classification views.

In figure 4.9, we present an example of a BROOM model in which the relation `Attends` is partitioned into the relations `Minors` and `Majors`. A student may attend between four and six courses and exactly one of those courses will be registered as that student's major course and the remainder as their minor courses.

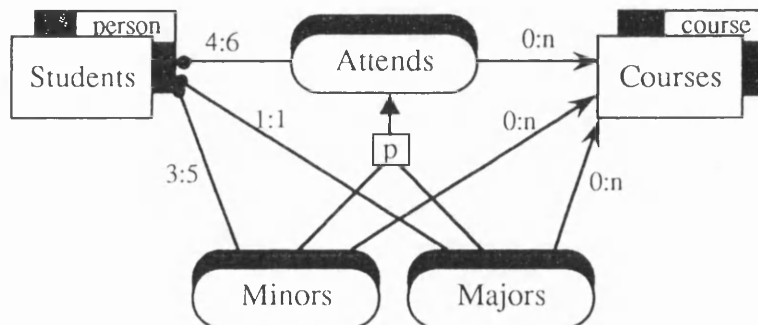

Figure 4.9: Relationship Partition

The ability to model generalisation over relationships in this way is absent in many semantic data models and this is probably due to the frequent internalisation of relationships. When a relationship is modelled as a separate logic structure, it is obvious that generalisation can be applied to relationships.

Some models, such as TAXIS [MBW80] support generalisation on dynamic as well as

static aspects of the model and have generalisation of both transactions and exception handlers.

The second aspect of generalisation that we consider here is the fact that there are proposals to support different forms of generalisation in terms of associated semantics. In particular, in the IFO model, [AH87] two forms of generalisation exist and these are referred to as generalisation and specialisation. The former corresponds to forming new concepts by generalising existing concepts and the latter to forming new concepts by specialising existing concepts.

Now it seems that in doing this they have confused the notion of the process by which one forms a classification structure with the notion of the semantics of the resulting classification structure. We have said previously that although we talk about the abstraction process of generalisation, the resulting classification structure can be arrived at either in a top-down manner or a bottom-up manner (or indeed a combination of these). Thus, one could start with the category person and specialise it to the categories staff and student, or, alternatively, start with the categories staff and student and generalise these into the category person. It is claimed in Hull and King [HK87], that the bottom-up process implies that the category person will be some form of union of the categories staff and student. In other words, in terms of BROOM constructs, the collections `Staff` and `Students` would form a partition of `Persons`. Now while this may very well be true, it is not a necessary truth and indeed it may also be true in the case of a top-down development process.

We therefore think that it is confusing to introduce two separate forms of generalisation in this way. It is also too restrictive in terms of the forms of constraints that can be imposed on families of collections related through *isa* relationships. Our preference in BROOM is to support the abstraction of generalisation in terms of collection families and to further support various forms of constraints that may exist within these families. The process by which the modeller arrives at these families is immaterial to the constraints that may be specified on them.

An important feature of the BROOM model is its capability to represent multiple classification views. Consider the example of the staff in the university database. It is possible that one application processes staff who are EEC nationals in a different way from those who are non-EEC nationals and it is therefore convenient to group staff into the two sets `EECs` and `NonEECs`. However, another application may wish to group staff individuals according to whether they are academic or non-academic staff. In other words, the grouping of entities into significant roles is dependent upon the particular application. In BROOM, a collection may belong to any number of collection families and hence it is possible for different applications to have different classification views of application entities. A BROOM model for the above example is given in figure 4.10. Here `Staff` is both partitioned into `EECs` and `NonEECs` while it is also covered by the subcollections `Academics` and `NonAcademics`.

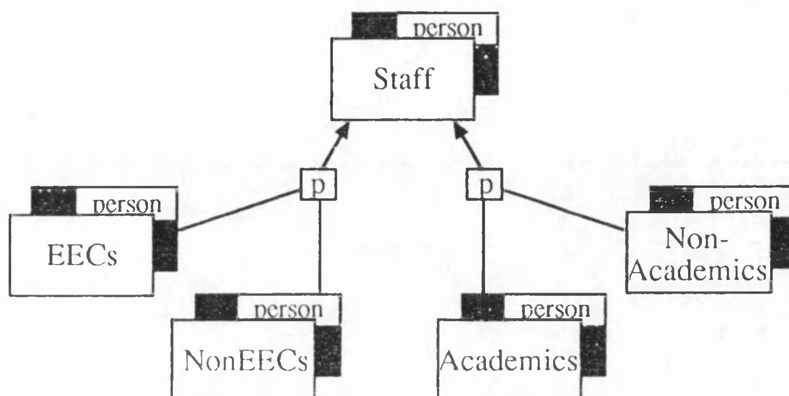

Figure 4.10: Multiple Classification Views

Consider how to represent these categories of staff if a collection can be the parent of only one partition (or other form of collection family). Then a choice must be made as to which partition takes precedence. For example, in figure 4.11 we show one way in which the classification structure of figure 4.10 could be modelled if only single classification views were allowed.

In figure 4.11, collection `Staff` is first partitioned into `EECs` and `NonEECs` and then each of these is in turn partitioned according to whether the staff individual is academic or non-academic. There are four resulting collections `EECAcademics`, `EECNonAcademics`, `NonEECAcademics` and `NonEECNonAcademics`. If the special categories which these represent are important to the applications, then the classification structure is fine. However, if the categories of interest are in fact those represented by `Academics` and `NonAcademics` in figure 4.10, then these collections have to be constructed from those in the database by means of query expressions.

The multiple classification views provide a basis for supporting view mechanisms in which a particular user or application programmer can access the database through one particular view and other views are be hidden. Such mechanisms have two general purposes. Firstly, they simplify things for the user and provide them with a view of the database that matches their requirements. Secondly, view mechanisms can provide a basis for access control in that a particular user may have authorisation to access only certain views. View mechanisms have to cater for problems of update when a user's update operations affect data not in the user's view. This is the well-known view update problem and investigations of this problem in the context of object-oriented databases include the work of Scholl et al [SST92].

The generality of the BROOM notion of collection family, together with the clear separation of classification from typing, results in a model which supports very flexible

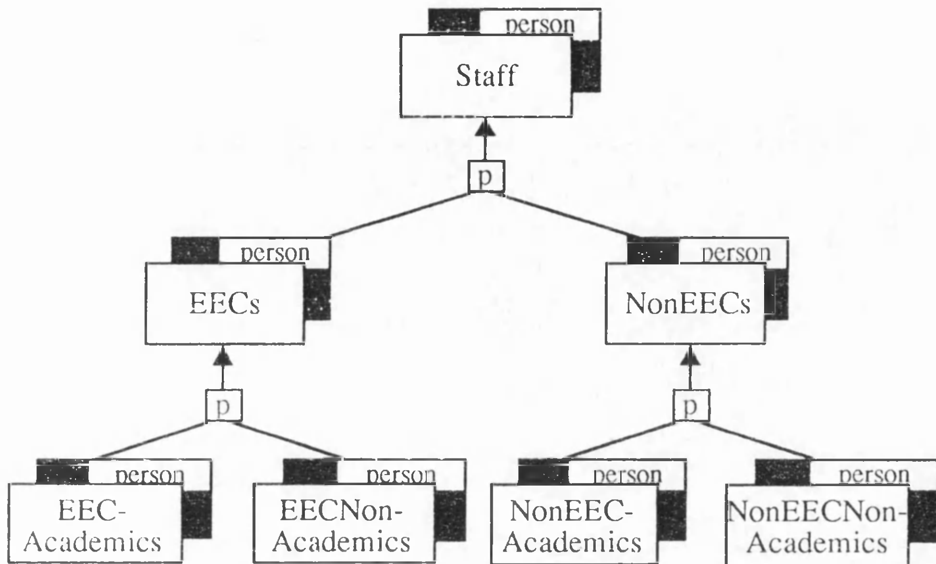

Figure 4.11: Single Classification View

and semantically rich classification structures. Object data models in which classification is done solely through types with the automatic maintenance of type extents require the introduction of subtypes to model specialisations of concepts. This means that the objects in these specialised concepts are forced to have different representations. A number of such models have therefore introduced the notion of object collections in order that objects belonging to the extent of a type may be grouped either on the basis of predicate conditions or by user specification. This prevents the unnecessary proliferation of subtypes but the user is forced to think in terms of the type and classification structures simultaneously in the presentation of the conceptual model. In the BROOM model, it is possible to present the overall model only in terms of the collections and their classification structure and relations. The user can examine the types of objects independently.

The BROOM collection family provides a single, uniform construct for the specification of classification structures. The construct is general in terms of both the fact that a family can have multiple parents and/or multiple children, and, also that there is no restriction on the number of families to which a collection can belong. As a consequence, support for multiple classification views arises naturally. The current model supports four special forms of constraints on families - denoted by the constraint terms `cover`, `disjoint`, `partition` and `intersection`. These are the forms commonly found in the literature on conceptual modelling; however, the generality of the collection family construct is such that additional forms of constraints could easily be incorporated. This contrasts with the situation in many other models where, either very restricted forms of classification structure are supported, or, additional

constructs are introduced to cater for different forms. Such object data models tend to lack uniformity and flexibility in their approach and as result they do not have the semantic modelling capability of BROOM.

Chapter 5

The Operational Model

The structural part of the collection model specifies the forms of collections and conceptual dependencies among collections that are supported in the collection model. In other words, it specifies the forms of valid database structures. A database is a representation of an application reality, but it is not of much use unless we can use it to ask questions about that reality. Then we must be able to perform operations to retrieve and process information contained in the database. The operational part of the collection model specifies the forms of operations supported.

Just as there are three levels for the structural model, so there are three corresponding levels for the operational model. Thus, there are operations on individual values, operations on collections of values and operations on a database. In section 1 of this chapter, we examine these three levels of operation and discuss how each level of operation may be considered as supporting a particular style of database programming.

The examination in section 1 of the levels of operation reveals that it is the level of operations on collections that is the prime concern of the operational part of the collection model. Operations on collections are specified by the proposed collection algebra. In section 2, we present the operations of the collection algebra and describe how the Z specification presented in Chapter 3 could be extended to include specifications of these operations. In section 3, we demonstrate the use of the operations of the collection algebra with some simple examples. The chapter concludes with a discussion of the properties of the collection algebra.

5.1 Operational Levels

As discussed in chapter 2, the three levels of operation are dependent upon the three levels of structure. In the proposed collection model, there are indeed three levels of structure - value, collection and database - and correspondingly, there are three possible levels of operation.

In this section we discuss briefly each of these three levels of operation and how the provision of operations at a particular level can be considered as supporting the particular styles of database programming as summarised in figure 5.1

Structure	Style of Programming
Database	Programming by Constraints
Collections	Programming by Querying
Values	Programming by Navigation

Figure 5.1: Levels of Operation

Operations on Values

Each value has an associated type which specifies the operations that can be performed on that value. In the case of object values, the associated object types may have a number of methods and these determine the behaviour of objects of that type in terms of the operations that can be performed on such objects, or indeed, by these objects. Operations on values are therefore determined by the underlying type system and the types of the application system.

Typically, object-oriented programming tends to be navigational in style. The programmer navigates from one object to another by means of "pointer chasing", i.e. following references to other objects. If the same operation is to be performed on a collection of similar objects, then the tendency is for a depth-first style of programming. The programmer tends to view the collection as a collection of complex objects rather than as a collection of objects with relationships to other collections of objects. Each object of the collection is processed in turn with the necessary pointer chasing that the execution of that operation may involve.

The recognition that many applications have collections of similar objects and require access to all of the members of a given collection, or at least a significant number of them, suggests that there should be some form of support for scanning a collection in order that each member can be processed in turn. In object-oriented languages with

support for collections, this is commonly done through the provision of iterators. An iterator is an object which keeps track of the current position in a collection and has a method to produce the next element of a collection.

In figure 5.2, we give an example of using an iterator to scan a set collection **Persons** and construct a collection **Results** with all those persons with the name "MacDonald". The code is based on a prototype of the BROOM model implemented using the object database management system Ontos [Ont91].

```
ODMSSetIterator* personIterator = Persons -> getIterator;
person* thisperson;
while  ( personIterator -> moreData())
       { thisperson = personIterator -> Next();
         if (strcmp(thisperson -> name(),"MacDonald"))
            Results -> Insert(thisperson);
       }
```

Figure 5.2: Using an Iterator to Scan a Collection

We refer to this style of database programming as *programming by navigation*. It is relatively low-level as it requires knowledge of the physical representation of data. The application programmer has to think in terms of a currency pointer which keeps track of the current position in collections over which iterations are being performed. Further, since it is based on value-at-a-time processing, any answers to queries are produced one value at a time and have to be explicitly inserted in a result collection whereas database applications frequently require collections of values as a result.

Database management systems based on the network and hierarchical models employ this style of navigational programming. One of the main advantages of the relational model was that it abandoned this in favour of an operational model at the level of collections of values.

Operations on Collections

Database programming tends to process collections of values rather than single values. It is therefore preferred if operations can be expressed in terms of these collections rather than in terms of the member values. The relational data model was one of the first to support operations on collections of values as opposed to individual values and this may be deemed one of the main reasons for its success. The relational algebra provided operations over sets of tuples and the results of these operations

were also sets of tuples. Then the relational systems were among the first to support breadth-first rather than depth-first programming.

As a consequence of providing operations over collections, the application programmer is presented with a higher-level, more declarative style of programming. For example, the collection expression

```
Results = (Persons % name = "MacDonald")
```

corresponds to the code of figure 5.2. Here, the '%' is used to indicate a selection operation on a collection. It specifies that the collection `Results` is equal to the collection comprising the members of `Persons` that satisfy the condition `name = "MacDonald"`.

We refer to this style of programming in terms of collections rather than individual values, *programming by querying*. It is at a higher-level than the navigational style described above in that the operations are expressed in terms of logical collections and it is therefore independent of physical representation details. Such query expressions could be translated into lower-level code in terms of iterators.

Programming at the higher-level can have further advantages in terms of performance as it is amenable to query optimisation techniques. These optimisation techniques will be based both on the algebraic properties of the operations involved and also on the physical characteristics of data. Thus, if the physical representation of data is altered or the properties of that data in terms of, for example, size of collections changes, then the system can take this into account in the selection of an appropriate evaluation strategy.

Operations on Databases

Given the two levels of structure - values and collections - and their corresponding levels of operation, it is natural to consider operations at the database level. Such operations are ones that take one database state and return another database state and therefore correspond to some form of update operation on a database. For when an update occurs, it is not sufficient to check the validity of the update on the individual values or collections involved but, given the conceptual dependencies that interrelate the various collections, it is necessary to check the validity with respect to the consistency of the resulting database state. Thus update operations must be considered as operations at the database level and the processing of these operations involves some form of constraint maintenance.

One approach to the problem of constraint maintenance is to use the constraints to determine changes that must be made to the database in order that consistency be

maintained. For example, the deletion of one object may result in certain inconsistencies in the database. However, it may be possible to restore consistency through further deletions. Clearly there should always be some criteria of minimum change as while deletion of the entire database would restore consistency, it is unlikely to satisfy user requirements.

The extent to which the idea of automatic update propagations is used within a system and the mechanisms to support this can vary. For example, a database system might have a rule base that specifies rules for the propagation of updates: effectively, this is akin to active databases which allow a database to be augmented with rules that specify actions to be taken under specified conditions. Alternatively, the system could be designed so that it propagates updates according to some general rules based on the form of structural constraints supported in the data model.

Then update operations can be specified at a very high-level in that a single update operation might result in many changes to the database. We call this style of programming *constraint programming* because it is a similar approach to that taken in constraint programming languages [Lel88]. A database must satisfy the set of constraints of its schema. An update operation can be considered as an additional constraint and it is the task of the constraint satisfaction system to determine a set of changes to be made to the database to satisfy the constraint set comprising the constraints of the schema and the new constraint.

While this provides a very high-level, powerful style of programming it might be of concern that the results of an erroneous update could have drastic consequences. For this reason, together with the fact that a user may wish to select between alternative strategies for achieving consistency, it would seem that this style of programming might be more suitable for interactive systems where the user can be informed of the consequences of their actions and in some cases enter a dialogue to select alternative actions. Another approach is to interact with the application programmer at the transaction design stage. A transaction design tool can be used to advise an application programmer of situations where constraints could be violated and recommend actions to be taken to prevent inconsistencies arising. The basis for such an approach has been investigated by Sheard and Stemple [SS88].

Levels of Operation

It should be stressed that while we would generally advocate higher-level, more declarative styles of programming, we are not suggesting that the other levels of operation can be dispensed with. In other words, operations at one level depend on operations at the lower levels. If a system supports programming by querying, then it is still the case that there will be operations at the level of individual values as specified by

the types. Indeed a system which supports the three levels of structure will require some form of operations at each of the three levels.

It has already been stated that the operations at the level of values are as specified by the underlying type system. The operations at the level of the database are concerned with update operations and are therefore associated with the evolutionary aspects of database systems as discussed in the next chapter. The remainder of this chapter focuses on the level of operations on collections.

5.2 A Collection Algebra

There are now a number of proposals for algebras for object data models e.g. [And91], [CDLR90], [SST92], [SZ89], [SZ90a], [Str90], [VD90b], [VD90a] and [VD91]. Sometimes these are referred to as object algebras but we feel this is something of a misnomer in that really they are algebras over collections of objects. We therefore prefer to call them collection algebras.

Andersen [And91], Scholl et al [SST92], Shaw and Zdonik [SZ89], [SZ90a] and Straube [Str90] all restrict their attention to set collections. Both Cluet et al [CDLR90] and Vandenberg and DeWitt [VD91] cater for multiple kinds of collections.

In this section, we present a collection algebra for the BROOM model. It is similar in approach to many of the other proposed algebras with the main difference stemming from the difference in the underlying data model. The BROOM model has two main forms of collections, unary and binary, which are used to represent entity categories and relationships, respectively. Correspondingly, the collection algebra has operations over these two forms of collections. Any operation that can be applied to unary collections can also be applied to binary collections by regarding the pairs belonging to that collection as simple member values. But there are operations which are specific to binary collections in that they assume that the form of member values is a pair. In addition, there are operations specific to ordered collections which assume an ordering of the member values.

The semantics of an operation will depend on the behaviour of the operand collections. For example, a union operation on two set collections will create a collection in which the extension is formed from the set union of the extensions of the operands. If a union operation is performed on two bag collections, then the extension of the resulting collection will be formed by taking the bag union of the extensions of the two operand collections.

Our presentation of the collection algebra will proceed as follows. First we provide a general description of the operations on collections including those that are specific

to binary and ordered collections. This is followed by definitions of the corresponding set and bag operations on collection extensions. From this we are then able to provide a general definition of collection operations.

Most of the operations presented in this section should be familiar as they correspond to operations of sets, bags and relations. Note that the group presented is far from minimal in that many of these operations can be defined in terms of other operations. Rather we intend the operations presented in this section to be considered as a fairly comprehensive group of convenience operations. This neither implies that each of these operations must be made directly available to the user in a query language based on this algebra nor does it imply that each of these operations should have a corresponding direct implementation. For example, we include a description of operations which return Boolean values: these include ones to check for the equality and the subcollection relations on collections. It may well be the case that these would not appear in a particular query language.

This situation can be likened to many presentations of relational algebra which start by introducing a large number of relational operators. Then a minimal set of operators is presented in terms of which the other operations can be defined. When a query language for a particular relational system is presented, it may well be the case that some of the operations introduced in the general overview are not supported in that language.

In the last section of this chapter, we examine the properties of the collection algebra and this includes a discussion of the notion of a minimal group of operations for the collection algebra.

For a given collection operation, we must be able to determine first of all whether the operation is valid and, secondly, the type of the resulting collection. A given collection type is specified by three pieces of information :- the form of the collection - unary or binary; the behaviour of the collection - set or bag; and the type of the members.

We start by considering the validity of a binary operation on collections in terms of the compatibility of the member types.

As introduced in section 3.2.1, we assume a subtyping relation on types that defines a partial ordering $\leq_t$ such that for any two types t_i and t_j, $t_i \leq_t t_j$ iff t_i is a subtype of t_j.

For two types t_i and t_j, we define a common supertype (upper bound) of t_i and t_j to be any type t_k such that $t_i \leq_t t_k$ and $t_j \leq_t t_k$.

If t_k is a common supertype of t_i and t_j, such that for any other common supertype t_l of t_i and t_j, $t_k \leq_t t_l$, then t_k is the least common supertype (least upper bound) of

t_i and t_j and we write $t_k = t_i \sqcup t_j$.

Assume $t_k = t_i \sqcup t_j$ and $t_l = t_i \sqcup t_j$. Then from the definition of least common supertype, it follows that $t_k \leq_t t_l$ and $t_l \leq_t t_k$. Since the relation $\leq_t$ is a partial odering, it is antisymmetric and it follows that $t_k = t_l$. Hence, if $t_i \sqcup t_j$ exists, then it is unique.

If $t_i \sqcup t_j$ exists, we say that types t_i and t_j are *compatible.* In the case that $t_i \sqcup t_j$ does not exist, then we say that types t_i and t_j are *incompatible.*

There are a number of operations which are available on all forms of collections. We start by giving a general informal description of these operations along with a signature. The type of a collection of member type t will be denoted $coll[t]$.

5.2.1 Operations on Collections

Union

$\cup : (coll[t_1], coll[t_2]) \rightarrow coll[t_1 \sqcup t_2]$

A union operation forms a collection containing the elements of the two operand collections. Types t_1 and t_2 must be compatible and the member type of the result collection is the least common supertype of t_1 and t_2. Note that this will be true even in the case where all of the elements of the result collection happen to be of a type which is more specialised than $t_1 \sqcup t_2$.

Intersection

$\cap : (coll[t_1], coll[t_2]) \rightarrow coll[t_1 \sqcup t_2]$

An intersection operation forms a collection containing those elements common to the two operand collections. In other words, the intersection of two collections is the "overlap" of those collections. As with the union operation, types t_1 and t_2 must be compatible.

The member type of the resulting collection may seem overly general since values that belong to both of the operand collections will be instances of both t_1 and t_2. It would be preferred to have the member type of the result collection inheriting the properties of t_1 and t_2, i.e. a subtype of both t_1 and t_2. However, for the general case, this would require a type system which supported multiple subtyping. We therefore

give the signature of the intersection operation in its weakest form and would expect it to be strengthened in the case of systems which would support the stronger form.

Difference

$- : (coll[t_1], coll[t_2]) \rightarrow coll[t_1]$

A difference operation forms a collection containing those elements of the first operand collection that are not members of the second operand collection. Types t_1 and t_2 must be compatible.

Selection

$\% : (coll[t], t \rightarrow \texttt{bool}) \rightarrow coll[t]$

A selection operation on a collection C forms a collection containing those elements of C that satisfy a given predicate. The predicate is represented by a function that maps each element of C to one of the Boolean values `true` or `false`.

Map

$\propto : (coll[t_1], t_1 \rightarrow t_2) \rightarrow coll[t_2]$

A map operation on a collection C applies a function to each element of C and forms a collection of the results.

Cartesian Product

$\times : (coll[t_1], coll[t_2]) \rightarrow coll[(t_1, t_2)]$

The Cartesian product of collections C_1 and C_2 is a collection of all pairs such that the first element of a pair is an element of C_1 and the second element of a pair is an element of C_2.

Flatten

$\pm : coll[coll[t]] \rightarrow coll[t]$

The flatten operator takes a collection of collections of the same member type and flattens them to a collection of values of that member type.

Reduce

$\oplus : (coll[t_1], (t_1, t) \rightarrow t, t) \rightarrow t$

The reduce operator is used to give aggregate operations such as `sum` and `product` over collections of values. Assume an initial value of type t and some function which takes a value of type t_1 and a value of type t to give a value of type t. Then this function can be applied in turn to each member of a collection with elements of type t_1 to generate a final value of type t by using the value of t produced at each stage as input to the next stage.

Cardinality

$\# : coll[t] \rightarrow$ `int`

The cardinality of a collection is the number of elements of that collection.

Subcollection

$\subseteq : (coll[t_1], coll[t_2]) \rightarrow$ `bool`

A subcollection operation on two collections tests whether the elements of the first collection are contained in the second collection. The operand collections must have compatible member types.

Member

$\in : (coll[t], t) \rightarrow$ **bool**

The member operator is used to test whether a value of type t is an element of a collection with member type t.

Equals

$= : (coll[t_1], coll[t_2]) \rightarrow$ `bool`

The equals operator tests whether two collections have the same member elements. The member types of the two operand collections should be compatible.

5.2.2 Operations on Binary Collections

Domain

dom : $coll[(t_1, t_2)] \rightarrow coll[t_1]$

The domain operator takes a binary collection C and forms a collection of all the values that appear as the first element of a pair belonging to C.

Range

rng : $coll[(t_1, t_2)] \rightarrow coll[t_2]$

The range operator takes a binary collection C and forms a collection of all the values that appear as the second element of a pair belonging to C.

Domain Restriction and Subtraction

dr : $(coll[(t_1, t_2)], coll[t_3]) \rightarrow coll[(t_1, t_2)]$

ds : $(coll[(t_1, t_2)], coll[t_3]) \rightarrow coll[(t_1, t_2)]$

Domain restriction takes a binary collection C and a collection C' and forms a binary collection comprising all those pairs of C with first value in C'. Domain subtraction with the same operands C and C' forms a binary collection comprising all those pairs of C with first value not in C'. Types t_1 and t_3 must be compatible.

Range Restriction and Subtraction

rr : $(coll[(t_1, t_2)], coll[t_3]) \rightarrow coll[(t_1, t_2)]$

rs : $(coll[(t_1, t_2)], coll[t_3]) \rightarrow coll[(t_1, t_2)]$

Range restriction takes a binary collection C and a collection C' and forms a binary collection comprising all those pairs of C with second value in C'. Range subtraction with the same operands C and C' forms a binary collection comprising all those pairs of C with second value not in C'. Types t_2 and t_3 must be compatible.

Inverse

inv : $coll[(t_1, t_2)] \rightarrow coll[(t_2, t_1)]$

The inverse of a binary collection C is the binary collection of pairs formed by swapping the first and second elements of pairs of C.

Composition

$\circ : (coll[(t_1, t_2)], coll[(t_3, t_4)]) \rightarrow coll[(t_1, t_4)]$

The composition of binary collections C_1 and C_2 is a binary collection of all pairs formed from matching pairs of C_1 and C_2. A pair (x, y) of C_1 matches a pair (w, z) of C_2 if $y = w$. Then the composition of C_1 and C_2 will contain the pair (x, z) formed by taking the first element of the pair from C_1 with the second element of the pair from C_2. The types t_2 and t_3 must be compatible.

Nest

nest : $coll[(t_1, t_2)] \rightarrow coll[(t_1, coll[t_2])]$

The nest operator is a form of grouping operator in that for each value that occurs in the domain of the binary collection, it forms a pair consisting of that value and the collection of values of the range of the binary collection to which it is related.

Unnest

unnest : $coll[(t_1, coll[t_2])] \rightarrow coll[(t_1, t_2)]$

The unnest operator expands a binary collection which contains pairs comprising a collection of values as the second element.

Division

div : $(coll[(t_1, t_2)], coll[t_3]) \rightarrow coll[t_1]$

The division operator takes a binary collection and a unary collection and returns the collection of all domain values of the binary collection such that the value is related to every member of the given unary collection. The types t_2 and t_3 must be compatible.

Closure

$* : coll[(t_1, t_2)] \rightarrow coll[(t_1, t_2)]$

The closure of a binary collection C is the reflexive transitive closure of the relation represented by C, i.e. $*C = \bigcup_{i=0}^{\infty} C^i$ where $C^0 = id_C$, $C^1 = C$ and $C^i = C^{i-1} \circ C$ for $i \geq 1$. id_C is the identity relation on elements of C. Types t_1 and t_2 must be compatible.

In the following subsections, we shall define the corresponding operations for both set and bags. We shall omit definitions of some of the operations that have already been defined or for which the definition follows directly from other definitions. These are the member operator $\in$, the equals operator $=$ and the closure operator $*$.

5.2.3 Operations on Ordered Collections

The set of operations available on ordered collections will depend upon the forms of ordering supported. If total orderings are supported, then the elements of an ordered collection may be viewed as a sequence. This means that the collection is effectively indexed numerically and that can be used to select elements from the collection according to numerical position. For example, given a collection C and $i, j \in \mathbb{N}$ such that $i \leq j$ and $j < \#C$, the operation $C[i : j]$ would construct an ordered collection with members of C indexed by n where $i \leq n \leq j$.

In general, elements of an ordered collection can be selected according to whether they precede or succeed given elements with respect to the ordering. For example, given a `Persons` collection ordered on the attribute `name` of the member type, `Persons["MacDonald":"Scott"]` would construct an ordered collection comprising all those members of `Persons` with name values that lie in the specified range with respect to the associated ordering.

A further issue concerning ordered collections is whether collections which are results of operations on collections will have an associated ordering. We assume that operations such as selection which have as a result a collection of the same type will preserve ordering, i.e. if the operand is an ordered collection then the result collection will be ordered and the ordering will be the same as that of the operand. In the general case of a binary operation, the result will be an ordered collection only in the case that the operands are both ordered and have the same ordering and this will be the ordering associated with the result. In addition, if the two operands of a Cartesian product are ordered, then the result will be ordered and the ordering will be based on the orderings of the operands in the usual way.

5.2.4 Operations on Sets

To define the operations over collections, it is first necessary to define corresponding operations over the extensions of collections. We start by defining these operations over sets.

Let S, S_1 and S_2 be sets.

$$\begin{aligned}
S_1 \cup_{set} S_2 &= \{x \mid x \in_{set} S_1 \vee x \in_{set} S_2\} \\
S_1 \cap_{set} S_2 &= \{x \mid x \in_{set} S_1 \wedge x \in_{set} S_2\} \\
S_1 -_{set} S_2 &= \{x \mid x \in_{set} S_1 \wedge \neg\,(x \in_{set} S_2)\} \\
S \,\%_{set}\, p &= \{x \mid x \in_{set} S \wedge p(x) = \texttt{true}\} \\
S \propto_{set} f &= \{f(x) \mid x \in_{set} S\} \\
S_1 \times_{set} S_2 &= \{(x, y) \mid x \in_{set} S_1 \wedge y \in_{set} S_2\} \\
\pm_{set} S &= \{x \mid \exists\, C \in_{set} S.x \in C\} \\
\oplus_{set} S\ f\ v &= \text{if } S = \varnothing_{set} \text{ then } v \\
&\quad\ \text{else } f(x, \oplus_{set} S'\ f\ v) \text{ where } S = S' \cup_{set} \{x\} \text{ and } x \notin_{set} S' \\
\#_{set} S &= \oplus_{set} S\ \lambda(x, v).(v+1)\ 0 \\
S_1 \subseteq_{set} S_2 &= \forall x.(x \in_{set} S_1 \Rightarrow x \in_{set} S_2)
\end{aligned}$$

We shall not specify the precise forms that can be taken by the predicate expression p in the selection operation %, and, the function f in the map operation $\propto$ and the reduce operation $\oplus$. To some extent, the available forms may depend upon the level and context in which the collection algebra resides. For example, if the collection algebra were to be made available to an application programmer by an embedding of a query language in C++, then it might be the case that the selection predicate p was specified by means of a Boolean function and a map or reduce function f could be any C++ function. Then both p and f could be considered as unknown quantities in that they might be side-effecting and their very generality would make it difficult to reason about their properties. On the other hand, an end-user could be presented with a query language in which, say, the forms of the selection predicates might be restricted to simple expressions in terms of object attributes. It could even be the case, that the end-user query language could bear little resemblance to the collection algebra with operations of the query language being mapped into operations of the collection algebra. Then the available query language operations would effectively restrict the valid forms of the operations of the collection algebra.

In particular, the definition of the reduce operator $\oplus_{set}$, is such that it is assumed that the function f is based on an operation which is both commutative and associative.

For example, the cardinality operator $\#_{set}$ is defined in terms of a reduce operation in which the reduction function is based on the operator + of integer addition and this is both commutative and associative. Then the result of the operation is order independent. This cannot be checked in general, and so two options arise. Either it is the responsibility of the user to ensure that f satisfies these properties, or, a given query language may provide a restricted set of specific reduce operations which correspond to frequent aggregate operations such as `sum`, `count`, `max`, `min` and `average`. The cardinality operation $\#_{set}$ defined above in terms of the reduce operator corresponds to `count`.

Operations on Sets of Pairs

Let S, S_1 and S_2 be binary sets, i.e. sets of pairs.

$$
\begin{array}{lcl}
\texttt{dom}_{set}\ S & = & \{x \mid \exists\, y.(x, y) \in_{set} S\} \\
\texttt{rng}_{set}\ S & = & \{y \mid \exists\, x.(x, y) \in_{set} S\} \\
S\ \texttt{dr}_{set}\ C & = & \{(x, y) \mid (x, y) \in_{set} S \wedge x \in C\} \\
S\ \texttt{ds}_{set}\ C & = & \{(x, y) \mid (x, y) \in_{set} S \wedge x \notin C\} \\
S\ \texttt{rr}_{set}\ C & = & \{(x, y) \mid (x, y) \in_{set} S \wedge y \in C\} \\
S\ \texttt{rs}_{set}\ C & = & \{(x, y) \mid (x, y) \in_{set} S \wedge y \notin C\} \\
\texttt{inv}_{set}\ S & = & \{(y, x) \mid (x, y) \in_{set} S\} \\
S_1\ \texttt{o}_{set}\ S_2 & = & \{(x, z) \mid \exists\, y.((x, y) \in_{set} S_1 \wedge (y, z) \in_{set} S_2)\} \\
\texttt{nest}_{set}\ S & = & \{(x, S_1) \mid x \in_{set} \texttt{dom}_{set}\ S \wedge S_1 = \{y \mid (x, y) \in_{set} S\}\} \\
\texttt{unnest}_{set}\ S & = & \{(x, y) \mid (x, S_1) \in_{set} S \wedge y \in_{set} S_1\} \\
S_1\ \texttt{div}_{set}\ S_2 & = & \{x \mid \forall\, y \in_{set} S_2.(x, y) \in_{set} S_1\}
\end{array}
$$

The $\texttt{nest}_{set}$ operation is similar to the *group_by* operation of relational algebra [Gra81]. The `nest` and `unnest` operations are fundamental in non-first-normal-form relational systems as they form the basis for conversion between normalised and unnormalised relations [JS82].

5.2.5 Operations on Bags

We now define the corresponding bag operations. First recall that in Chapter 3, we introduced two representations of bags. Assume that a bag B contains two occurrences of x and one occurrence of y. Then we may either use a bag notation and

denote B by $\langle x, x, y\rangle$ or we may use a set representation where $B = \{(x,2),(y,1)\}$. Then we may write $x \in_{bag} B$ and $(x,2) \in_{set} B$.

Let B, B_1 and B_2 be bags.

$$\begin{aligned}
B_1 \cup_{bag} B_2 &= \{(x,n) \mid \exists\, n_1, n_2.((x,n_1) \in_{set} B_1 \wedge (x,n_2) \in_{set} B_2 \wedge \\
&\quad n = max(n_1,n_2))\} \\
B_1 \uplus B_2 &= \{(x,n) \mid \exists\, n_1, n_2.((x,n_1) \in_{set} B_1 \wedge (x,n_2) \in_{set} B_2 \wedge \\
&\quad n = n_1 + n_2)\} \\
B_1 \cap_{bag} B_2 &= \{(x,n) \mid \exists\, n_1, n_2.((x,n_1) \in_{set} B_1 \wedge (x,n_2) \in_{set} B_2 \wedge \\
&\quad n = min(n_1,n_2))\} \\
B_1 -_{bag} B_2 &= \{(x,n) \mid \exists\, n_1.((x,n_1) \in_{set} B_1 \wedge ((x \notin_{bag} B_2 \wedge n = n_1) \vee \\
&\quad \exists\, n_2.((x,n_2) \in_{set} B_2 \wedge n_1 > n_2 \wedge n = n_1 - n_2)))\} \\
B \,\%_{bag}\, p &= \{(x,n) \mid (x,n) \in_{set} B \wedge p(x) = \mathtt{true}\} \\
B \propto_{bag} f &= \oplus_{set}\ B\ \lambda(x,n), B_1.(\{(f(x),n)\} \uplus B_1)\ \varnothing_{bag} \\
B_1 \times_{bag} B_2 &= \{((x,y),n) \mid \exists\, n_1, n_2.((x,n_1) \in_{set} B_1 \wedge (y,n_2) \in_{set} B_2 \wedge \\
&\quad n = n_1 * n_2)\} \\
\pm_{bag} B &= \{(x,n) \mid \exists (B_1,n_1) \in_{set} B.\exists\, n_2.((x,n_2) \in_{set} B_1 \wedge \\
&\quad n = n_1 * n_2)\} \\
\oplus_{bag} B\ f\ v &= \text{if } B = \varnothing_{bag} \text{ then } v \\
&\quad \text{else } f(x, \oplus_{bag}\ B'\ f\ v) \text{ where } B = B' \uplus \{(x,1)\} \\
\#_{bag} B &= \oplus_{bag}\ B\ \lambda\, x, v.(v+1)\ 0 \\
B_1 \subseteq_{bag} B_2 &= \forall (x,n_1) \in_{set} B_1.(\exists\, n_2.((x,n_2) \in_{set} B_2 \wedge n_1 \le n_2))
\end{aligned}$$

There are two forms of union operation on bags. $\cup_{bag}$ forms the bag containing all those elements of the operand bags and the number of occurrences of an element is the maximum of its number of occurrences in the operands. In contrast, the number of occurrences of an element in the result of operation $\uplus$, is the sum of the occurrences in the operand bags. We take $\cup_{bag}$ to be our definition of bag union and we refer to $\uplus$ as bag addition. A query language based on the algebra may or may not choose to make both operations available to users. We use the bag addition operation $\uplus$ in the definition of other operations on bags: it ensures no loss of information when combining bags.

The map operation on bags is defined in terms of the reduce operation on sets. The set over which the reduce takes place is the set representation of the bag. For each pair

$(x, n) \in_{set} B$, the function f is applied to the bag element x and a bag constructed containing n occurrences of $f(x)$, i.e. $\{(f(x), n)\}$. The map of f over B is then the bag addition of all such constructed bags.

Operations on Bags of Pairs

Let B, B_1 and B_2 be binary bags, i.e. bags of ordered pairs.

$$\begin{array}{lcl}
\mathtt{dom}_{bag}\ B & = & \bigoplus_{bag}\ B\ \lambda(x, y), B_1.(\langle x \rangle \uplus B_1)\ \varnothing_{bag} \\
\mathtt{rng}_{bag}\ B & = & \bigoplus_{bag}\ B\ \lambda(x, y), B_1.(\langle y \rangle \uplus B_1)\ \varnothing_{bag} \\
B\ \mathtt{dr}_{bag}\ C & = & \{((x, y), n) \mid ((x, y), n) \in_{set} B \wedge x \in C\} \\
B\ \mathtt{ds}_{bag}\ C & = & \{((x, y), n) \mid ((x, y, n)) \in_{set} B \wedge x \notin C\} \\
B\ \mathtt{rr}_{bag}\ C & = & \{((x, y), n) \mid ((x, y), n) \in_{set} B \wedge y \in C\} \\
B\ \mathtt{rs}_{bag}\ C & = & \{((x, y), n) \mid ((x, y), n) \in_{set} B \wedge y \notin C\} \\
\mathtt{inv}_{bag}\ B & = & \{((y, x), n) \mid ((x, y), n) \in_{set} B\} \\
B_1\ \circ_{bag}\ B_2 & = & \{((x, z), n) \mid \exists\, y, n_1, n_2.(((x, y), n_1) \in_{set} B_1 \wedge \\
 & & ((y, z), n_2) \in_{set} B_2 \wedge n = n_1 * n_2)\} \\
\mathtt{nest}_{bag}\ B & = & \{(x, B_1) \mid x \in_{bag} \mathtt{dom}_{bag}\ B \\
 & & \wedge\ B_1 = \{(y, n) \mid ((x, y), n) \in_{set} B\}\} \\
\mathtt{unnest}_{bag}\ B & = & \{((x, y), n) \mid (x, B_1) \in_{bag} B \wedge (y, n) \in_{set} B_1\} \\
B_1\ \mathtt{div}_{bag}\ B_2 & = & \{x \mid \forall\, y \in_{bag} B_2.(x, y) \in_{bag} B_1\}
\end{array}$$

The $\mathtt{dom}_{bag}$ operation is defined in terms of bag reduce. The reduce function takes two arguments, a pair (x, y) and a bag B_1, and yields the bag addition of the singleton bag $\langle x \rangle$ and B_1. Thus, the number of occurrences of a given element x in the domain of a bag is equal to the total number of pair occurrences in the bag with x as the first element of the pair. The definition for $\mathtt{rng}_{bag}$ is similar in form.

5.2.6 Specifying Collection Operations in Z

We are now in a position to give complete definitions of the collection operations. Basically, it is a matter of stating when it is appropriate to apply the particular set and bag operations to the extensions of the operand collections to construct the extension of the result collection. In the case of unary operations, set operations will apply to the extensions of set collections and bag operations to the extensions of bag

collections. With binary operations, we assume that both operands must have the same behaviour. Thus, if both operands are set collections then a set operation will apply and if both operands are bag collections then a bag operation will apply.

Additional operations for converting bags to sets and vice versa could be included in the available operations on collections. Then in the event of a binary operation on a set collection and a bag collection, one of the operands could be converted. The issue of automatic conversions between the forms of collections will not be discussed further here although it was investigated in a prototype implementation of the collection algebra [Ong91].

The Z specification presented in chapter 4 could be extended to give complete specifications of the collection algebra. We shall not give all of such a specification here but rather shall indicate how it would be done by giving the specification of the union operation on collections.

First we give a schema *LUB* which defines the least upper bound of two types.

LUB

$TYPES$
$_ \sqcup _ : Names \times Names \nrightarrow Names$

$\forall t_1, t_2 : Types \bullet$
$\exists t : Types \bullet (t = t_1 \sqcup t_2 \Rightarrow$
$((t, t_1) \in SubTypes^+ \wedge (t, t_2) \in SubTypes^+ \wedge$
$\neg (\exists t' : Types \mid t \neq t' \bullet$
$(t', t_1) \in SubTypes^+ \wedge (t', t_2) \in SubTypes^+ \wedge (t, t') \in SubTypes^+)))$

Note that the notion of type compatibility is built into the schema *LUB* since the function $\sqcup$ will be defined only in the event that the operand types have a common supertype.

Since the operations on collections will be defined in terms of operations on the extensions of collections, it is convenient to introduce a function which maps collections to their extensions. This is defined as a function `ext` in the following schema.

EXTENSION

$VALUES$
$\mathtt{ext} : OBJID \nrightarrow \mathbb{P}\, VALUE$

$\forall C : Collections \bullet$
$(C \in SetCollections \Rightarrow$
$\mathtt{ext}\ C = \{x \mid ((C, x), 1)\ \text{in}\ Members\} \wedge$
$C \in BagCollections \Rightarrow$
$\mathtt{ext}\ C = \{(x, n) \mid ((C, x), n)\ \text{in}\ Members\})$

The extension function `ext` is similar to the `elements` function defined in the Z specification of Chapter 3. The difference is that the `ext` function gives either a set or bag depending on whether a collection is a set collection or a bag collection. The `elements` function gives a bag of the member elements of a collection and does not distinguish between set and bag collections.

The union operation on collections is defined in the schema *UNION*. Note that this Z-schema specifies what it means for a collection to be the union of two other collections, but it does not deal with the dynamic part of creating that collection.

$$
\begin{array}{l}
\textit{UNION} \\
\hline
\textit{LUB} \\
\textit{EXTENSION} \\
_ \cup _ : \textit{OBJID} \times \textit{OBJID} \rightarrow\!\!\!\!\!\!+\; \textit{OBJID} \\
\hline
\forall\, C_1, C_2 : \textit{Collections} \bullet \\
\exists\, C : \textit{Collections} \bullet \\
(\textit{MemberTypes}\ C = \textit{MemberTypes}\ C_1 \sqcup \textit{MemberTypes}\ C_2 \wedge \\
(C_1 \in \textit{SetCollections} \wedge C_2 \in \textit{SetCollections} \wedge \\
\quad C \in \textit{SetCollections} \wedge \textit{ext} C = \textit{ext} C_1 \cup_{set} \textit{ext} C_2) \vee \\
(C_1 \in \textit{BagCollections} \wedge C_2 \in \textit{BagCollections} \wedge \\
\quad C \in \textit{BagCollections} \wedge \textit{ext} C = \textit{ext} C_1 \cup_{bag} \textit{ext} C_2)) \\
\Rightarrow C = C_1 \cup C_2
\end{array}
$$

In this Z-schema, we refer to the operations $\cup_{set}$ and $\cup_{bag}$. It is assumed that these have been defined previously according to the definitions given earlier. The set union operation $\cup_{set}$ is equivalent to the standard operation $\cup$ as defined in the mathematical toolkit for Z. However, we have used $\cup_{set}$ in the above Z-schema to emphasise the general approach of specifying the operations on collections using operations defined on sets and bags. While the Z toolkit defines a number of the set operations and a few of the bag operations, it does not provide all of them and so Z definitions for at least some of them would have to be introduced.

The above Z-schema could in fact be replaced by a Z-schema that defines a number of operations of a similar form. In this way, a complete Z specification for the collection algebra could be constructed.

5.3 Example Queries

The collection algebra presented in section 5.2 should not be taken as a definition of a query language. Rather, it is the basis for a target language into which query languages would be mapped. Further, it is not necessary that all of the operations

presented in section 5.2 would have to be supported directly. As we have seen, some of these operations can be defined in terms of other operations. It is also true that the set of operations could be extended.

A particular query language for the collection model could be as near or far from the collection algebra as the designer of that language wished it to be. For example, it could be very close to the collection algebra in that each operation of the algebra has a corresponding syntactic construct in the language. Alternatively, it could be the case that the query language bears very little resemblance to the collection algebra and possibly provides much higher-level operations which map into one or more operations of the collection algebra.

In this section, we simply wish to give a flavour of the collection algebra through consideration of a simple BROOM schema and a few queries based on that schema. We therefore adopt a query language notation which is very close to the collection algebra.

We shall use a simple version of the infamous parts and suppliers database for the examples. The database contains information on suppliers, the parts they supply and the jobs that require those parts. The BROOM schema is shown in figure 5.3.

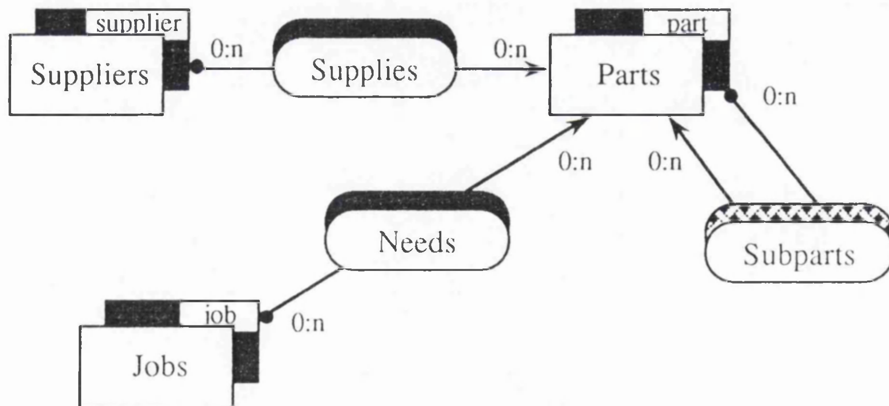

Figure 5.3: Parts and Suppliers Example

The interesting part of the BROOM schema is the representation of the relation between parts and their component subparts. A part is assembled from a number of component parts and it may require several items of a particular part. For example, a part p_1 may be assembled from two items of part p_2 and three items of part p_2. This assembly can be represented by a bag which maps parts to their immediate subparts as follows:

$$\langle (p_1, p_2), (p_1, p_2), (p_1, p_3), (p_1, p_3), (p_1, p_3) \rangle.$$

Therefore, the relation between parts and their component parts is given by the binary bag collection `Subparts`.

In addition to the BROOM schema of figure 5.3 which gives the collections and their relations, we also require type declarations for the types of the members of these collections. These type declarations are given in figure 5.4.

```
record type address
            street : string;
            city : string
end;

object type supplier
            name : string;
            location : address
end;

object type part
            partname : string
            colour : string;
            cost : integer
end;

object type job
            jobnum : string;
            site : address
end;
```

Figure 5.4: Type Declarations

Having introduced our example schema, we now examine some simple queries.

1. *Which suppliers supply red parts?*

   `RedSuppliers = dom(Supplies rr Parts%colour="red")`

   A selection is performed on collection `Parts` to find all those elements which have the value red for the colour attribute. The range restriction operation on `Supplies` selects all those pairs in the collection where the second value is in the collection of red parts. By taking the domain of the result of the range restriction, we get the collection of all suppliers who supply red parts.

2. *Find all local suppliers for each job.*

   ```
   LocalSuppliers =
   (Needs o inv(Supplies))% fst.site.city = snd.location.city
   ```

   Here we assume that local suppliers are ones which are located in the same city as the job site. First of all we construct a binary collection which relates jobs to suppliers. The inverse of Supplies, `inv(Supplies)`, gives a binary collection which relates parts to suppliers. Then the composition of `Needs` and `inv(Supplies)` relates jobs to suppliers who supply the job parts. Then we select from the collection all those pairs such that the city of the job is the same as the city of the supplier. The selection condition

   ```
   fst.site.city = snd.location.city
   ```

   uses two specially defined functions `fst` and `snd` which return the first and second elements of a pair value, respectively.

3. *Which suppliers supply every part?*

   ```
   UnivSuppliers = Supplies div Parts
   ```

   This query is an example of the division operation. It selects all the values in the domain of `Supplies` that are related to every element of Parts.

   Without the division operation the query could be expressed as follows:

   ```
   UnivSuppliers = Suppliers - dom((Suppliers × Parts) - Supplies)
   ```

   The cartesian product forms the collection of all possible (supplier,part) pairs. By removing all the pairs that appear in `Supplies`, and then taking domain of that collection, we get the collection of all those suppliers who do not supply all parts. This is then subtracted from `Suppliers` to give the collection of suppliers who do supply all parts.

4. *Find the bill of materials for part p.*

   It is assumed that the cost of a part is the cost of all the component parts plus a unit cost. In the case of parts with no subparts, the unit cost gives the actual cost of the part. For other parts, the unit cost is the assembly cost of the part.

   ```
   Bill = ⊕ ((*Subparts)% fst.partname="p") λ(x,y),v.(y.cost+v) 0
   ```

   `Subparts` is a bag collection which relates parts to their immediate subparts. The transitive closure `*Subparts` is a binary collection which relates parts to all of their subparts. For a given pair (p_1, p_2) in `*Subparts`, the number of occurrences of the pair in the collection is the total number of part p_2 required to assemble part p_1. Since we are interested only in the bill of materials for part `p`, we then select all those pairs of `*Subparts` such that the part name of the first element is `p`. Then to get the total cost of part `p`, we sum the costs of all the subparts of `p` represented in the bag collection `(*Subparts) % fst. partname = "p"`. This is done by a reduce operation which takes as arguments the function `λ(x,y),v.(y.cost+v)` and the initial value `0`. The function `λ(x,y),v.(y.cost+v)` takes a pair of arguments, the first of which

is a pair of values (x, y) and the second a value v, and returns the sum of v and the cost of y. The reduce operation maps this function over the collection, accumulating the costs and returns the total cost of part p.

5.4 Properties of the Collection Algebra

In this section, we discuss various properties of the operations introduced in the section 5.2.

We first consider the algebraic properties of the operations and how these could be used during the logical optimisation of queries to rewrite query expressions. This is followed by a brief discussion of the idea of a minimal set of operators in terms of which the other operations can be defined.

Many of the well-known properties of set operations also apply to bag operations. For example, a summary of the associativity and commutativity properties of binary operations on sets and bags is given in figure 5.5.

	Commutative		Associative	
Operation	Set	Bag	Set	Bag
$\cup$	$\surd$	$\surd$	$\surd$	$\surd$
$\cap$	$\surd$	$\surd$	$\surd$	$\surd$
$-$	-	-	-	-
$\times$	-	-	-	-
$\circ$	-	-	$\surd$	$\surd$

Figure 5.5: Associativity and Commutativity of Binary Operations

If a property applies to corresponding set and bag operations, then it will in turn apply to the corresponding operation on collections. For example, both bag union and set union are commutative, and so the commutativity property holds for union over collections, i.e. for any collections C_1 and C_2, $C_1 \cup C_2 = C_2 \cup C_1$.

The algebraic properties of the collection algebra can be used to generate rewrite rules for the transformation of query expressions into equivalent expressions which are less expensive to evaluate. Given a host of algebraic properties, the difficulty is to know when it is sensible to use a property in the sense of applying the corresponding rewrite rule.

The operations that are most difficult to reason about with respect to such properties are the unary operations of selection, map and reduce. The property of selection being distributive over binary operations is the basis for a great many logical query optimisations in relational systems. The distribution of a select operation over a binary operation promotes the selection: this results in the reduction of operand relations to the expensive binary relational operations such as join. Even in the case of relational systems, it is not universally the case that selection distributes over join; the property is dependent upon the form of the selection condition. (Full details of the properties of the relational operations is given in [CP84].)

In the collection algebra, if the predicate expressions of the selection operation are restricted to simple predicate conditions over attributes of objects, then the operation is similar to that of relational select and corresponding properties apply.

It is interesting to note that, even if the same algebraic property holds in the case of both the relational algebra and the collection algebra, the policy for the use of that property may be quite different. For example, as mentioned above, in relational systems, a standard optimisation strategy is to promote selections such that the tuples relevant to a query are identified as soon as possible in the execution of that query.

In an object-oriented system, selections may be expensive and it may be desirable to defer them. Generally, a collection is a group of object references rather than the objects themselves. For some queries, most of the processing can be performed on these collections of object references without access to the objects. This avoids the mapping of these objects into memory and this is particularly advantageous if objects are large. If a selection involves attributes of an object that are not indexed, then it will be necessary to access the actual objects and this can be expensive. It is therefore preferable to perform as much processing as is possible on the collections alone, and only then to map the individual objects into memory for further processing. Then a general rule is to promote only those selections where the predicate expression is in terms of indexed attributes and the selection can be performed without access to the objects of the operand collection.

The above discussion assumes a restricted form of predicate expression. If general forms of predicate and function expressions can occur in the unary operations of selection, map and reduce, it is difficult to reason about them. In particular, a function may be side-effecting in which case the result of one of these operations may be non-deterministic as it depends on the order of access to elements of the collection.

As a consequence, it is difficult to present general properties of these operations. In a particular realisation of the collection model, the forms that these predicate and function expressions may take can be such that it is possible to establish properties of commutativity and distributivity as in relational systems.

We now turn to consider the issue of providing a minimal set of operators for the collection algebra. In section 5.2, we presented a large number of operations on set and bag collections. Some of these operations were in fact defined in terms of other operations. The question then arises of providing a minimal set of operations sufficient to define all others. This has been addressed by Watt and Trinder in their theory of collections [WT90]. They present four basic operations for the construction and manipulation of collections as given in figure 5.6.

Operation	Behaviour	Type
empty	Builds an empty collection	$coll[t]$
single x	Builds a collection containing a single element, x	$t \rightarrow coll[t]$
$c \odot c'$	Builds a collection containing the elements of c and c'	$(coll[t], coll[t]) \rightarrow coll[t]$
iter f c	Iterates over c applying the multi-valued function f to every element, and combines the resulting collections.	$(t_1 \rightarrow coll[t_2], coll[t_1]) \rightarrow coll[t_2]$

Figure 5.6: Basic Collection Operations

They define a data type to be a collection type if it is equipped with the operations given in figure 5.6, such that they obey certain laws. Sets, bags, lists and binary trees equipped with the appropriate $\odot$ operation all satisfy the required properties of collections. Based on this collection theory, we can provide a minimal set of operations for the collection algebra.

Details of the collection theory and examples to show how the other operations can be defined in terms of the basic operations are given in [WT90].

A point to consider is the role that such a minimal set of collection operations would play. Clearly, it is useful in proving properties about any algebra to be able to work with a minimal set of operations. However, this does not mean that this minimal set of operations should correspond directly either to the operations made available in a query language or to the operation set in a particular implementation.

An important factor in the design of a query language is ease of expression and it is therefore desirable to support high-level operations that correspond to the sorts of actions frequently required by the users. At the implementation level, the minimal set of operations may be too general to provide a suitable target for the query language. Further, the generality of the basic operations makes it difficult to reason about algebraic properties that lead to useful optimisations. As we have already discussed,

the provision of a restricted form of selection allows us to derive rewrite rules for query transformation. Expressing a selection operation in terms of a general iterate operation and function makes it difficult to detect equivalent optimisations.

In this section, we have only touched upon some of the issues raised by the collection algebra in connection with query optimisation. Many of these issues can not be addressed at the level of a general collection model, but rather must be tackled in the context of a particular realisation of the model. In particular, while there are general algebraic properties of the collection algebra useful in query optimisation, a number of useful equivalences arise from restricted forms of these operations that might apply in a given system. Further, the choice of transformations for a given query will be determined by system characteristics.

The issues of query equivalence and optimisation have been examined in the context of a number of the proposed algebras, e.g. [And91], [SZ90b], [Str90] and [VD90a]. In particular, Straube's thesis [Str90] is a detailed study of query processing strategies in object-oriented database systems. He proposes an object calculus with a target algebra. Queries expressed in the object calculus are translated into equivalent expressions of the algebra. These algebra expressions are optimised by applying equivalence preserving rewrite rules. From the optimised algebra expression, an access plan is generated for query evaluation. The limitations of Straube's work stem from the restricted form of the underlying data model. However, it does provide a foundation for studies of query processing in object-oriented systems based on other object data models.

Chapter 6

Evolution

A database is evolutionary in that during its lifetime both the real-world entities being represented and the requirements of the database system may evolve. This leads to three forms of evolution within a database system. First, there is object creation and deletion which correspond to the creation and destruction of entities of the application domain. Second, in object evolution, the classification of an individual entity changes: this may also involve a change in the form of representation of that entity. Within our university database system, we would wish to reflect the changes that individual entities undergo as they change their roles in the real world. For example, an undergraduate student might graduate and enrol on a postgraduate course. Then we would want to support object evolution in which an object classified as an undergraduate student could metamorphose into an object classified as a postgraduate student. Thirdly, there is schema evolution in which the underlying structure of the database changes. For example, a change in the requirements of a university database system might cause us to add methods or attributes to the type definition of `student` or, indeed, to add a new type definition `postgraduate` and collection `Postgraduates`.

In this chapter, we examine the various requirements and forms of evolution and propose extensions to the collection model to support evolution. The main contribution is on object evolution and the reasons for this are twofold. Firstly, although for reasons we shall discuss in detail later, object evolution is considered simpler than schema evolution, still it is not supported in most of the existing object-oriented database systems. Object evolution is a basic form of evolution and yet it appears to be a problem overlooked by many researchers who rather concentrate on the perhaps more challenging problems of engineering systems to support schema evolution. Secondly, most of the support for schema evolution is under control of the underlying type system. The difficulty of schema evolution is the representation of type instances under type evolution in that the system may have to support multiple forms of representation of objects of the same type and to deal with these in a consistent and

reasonable manner during processing. The collection model is built on top of some assumed type system and has no control over type evolution and the representation of type instances.

The first section examines object creation and deletion. We then go on to propose an extension to the collection model to cater for object evolution and discuss the requirements that this would place on the underlying persistent object system. Finally, we consider schema evolution and provide an overview of the proposals for type evolution and also the forms of schema evolution that would be supported at the collection model level.

6.1 Object Creation and Deletion

There are serious philosophical questions concerning the existence of entities. At what point can we say that an entity exists? When does an entity cease to exist? In particular we can ask: Is a person who has died still a person? And a question that has been very controversial with respect to the abortion debate: At what point can we say that a person exists?

In the context of database systems, the above issues are simplified in that it is not a question of the existence of entities but rather that of the existence of their representations within a database that matters. The period of existence of a representation of an entity corresponds to that period during which the entity is of interest to the application system. Thus, if we are interested in maintaining records on dead people then the representation of a person will exist beyond the death of that person. And the representation of a person begins its existence when the database is first informed of the existence of the person entity. Then a person's existence in a university database is established at the time that they are deemed to be of interest to the various applications: this may be when they first register as a student or are employed as a member of staff.

This distinction between an entity and its representation is important. Its significance leaps to the fore in two special categories of database systems - temporal database systems and databases for computer assisted software engineering (CASE). In temporal databases you want to be able to make enquiries both about the period of existence of an entity and also about the period of existence of its representation. In a CASE system, the entity and its representation may be one and the same. Thus the object code of a program is its own representation.

An object exists in a database when it is a member of one or more collections of that database or it is referenced by an object in a collection. Whether or not an object may persist in a system outwith a database depends on the specific nature

of the system. For example, in the Comandos system [CBHdP93] an object may be created and persist without being a member of any database collection or, indeed, being referenced by a member of a collection. Then we do not consider such an object to exist in the database until it is explicitly inserted in one or more collections of the database or is reachable from these collections. However, in the simple object database management system, COLLEEN, which was also based on the collection model, an object must be inserted in one or more collections at the time of creation.

Thus, we are not so much concerned with the construction of an object but rather with its creation in the database in terms of its addition to one or more collections of the database. An object can be added to a collection by adding its object identifier to the extension of that collection. We have a general *ADD* operation which adds a value to a collection. The operation requires that the value is an instance of the member type of the collection.

Before specifying the *ADD* operation, we first introduce a Z-schema $\Delta MEMBERS$ which specifies that only *Members*, *UnaryMembers* and *BinaryMembers* can change. We adopt the usual Z convention that for a Z-schema S, the notation ΞS denotes a Z-schema which specifies that under an operation the before and after values of variables will be the same. Further, for a given Z-schema S, then S' is the same as S except that all variable names are primed. Then it is usual to use the primed variable names to refer to the state after an operation and the unprimed names to refer to the state before an operation.

$\Delta MEMBERS$

DB
DB'
$\Xi ATOMICVALUES$
$\Xi TYPES$
$\Xi FAMILYMEMBERS$
$\Xi RELATIONCARDINALITIES$

$Values' = Values$
$Firsts' = Firsts$
$Seconds' = Seconds$
$Instances' = Instances$

The Z-schemas *ATOMICVALUES*, *TYPES*, *FAMILYMEMBERS* and *RELATIONCARDINALITIES* incorporate all the specifications of objects, collections, constraints and types. All of this information is unchanged under the operations of adding and removing given objects in collections. The *Values* relation should be unchanged by the addition or removal operations and this is specified by the equality $Values' = Values$. Likewise, we specify *Firsts*, *Seconds* and *Instances* as being unchanged. This leaves the *Members* relation, with its partitioning relations

UnaryMembers and *BinaryMembers*, to be open to change by operations for the addition and removal of objects from collections. The relationship between the old state and the new state will be given by the relationship between these variables and the corresponding primed variables *Members'*, *UnaryMembers'* and *BinaryMembers'*.

We specify the addition operation *ADD* in the Z-schema below. The operation does in fact specify the addition of any form of value, and not only object values, to collections. The inputs to the operation are the value to be added and the name of a collection.

$$
\begin{array}{l}
\textit{ADD} \\
\hline
\Delta \textit{MEMBERS} \\
v? : \textit{VALUE} \\
c? : \textit{OBJID} \\
\hline
c? \in \textit{Collections} \\
t = \textit{MemberType}(c?) \Rightarrow (v?, t) \in \textit{Instances} \\
v? \in \textit{AtomicValues} \Rightarrow \textit{UnaryMembers}' = \textit{UnaryMembers} \uplus \{(c?, v?)\} \\
v? \in \textit{PairValues} \Rightarrow \textit{BinaryMembers}' = \textit{BinaryMembers} \uplus \{(c?, v?)\} \\
\hline
\end{array}
$$

If the collection specified is an existing collection and the value is an instance of the member type of that collection, then the value is added to the collection. If the value is an atomic value, then the pair $(c?, v?)$ will be added to the bag *UnaryMembers*. Note that $\uplus$ denotes the operation of bag addition. Correspondingly, if the value is a pair value, then the pair $(c?, v?)$ will be added to *BagMembers*. In either case, the pair $(c?, v?)$ must also be added to *Members*. This is not specified explicitly, since the value of *Members'* is determined by *UnaryMembers'*, *BinaryMembers'* and the associated partitioning constraint.

There are a number of points to note about this apparently simple specification. Δ*MEMBERS* represents a complex structure with lots of specified constraints and these are such that the above operation will be performed only in the case that the constraints are not violated. This means that if the collection is a set collection and $v?$ already belongs to that collection then although *Members* is a bag, the constraints are such that the operation would not take place. Similarly, if $v?$ is a pair value then the collection $c?$ must be a binary collection. This raises the question as to what happens when the constraints are violated and the operation cannot be performed.

The issue of constraint violation is determined by the constraint model and its associated constraint management system. As discussed in the previous chapter, there are a number of different approaches to constraint handling in terms of when constraints should be checked and the actions to be taken in the event of violations. At one extreme, every violation can be considered as an error condition and, at the other extreme, constraints can form the basis for a style of constraint driven system operation.

Even in the case of the fixed forms of constraints in the BROOM model, these different approaches to constraint management may be adopted. It was important that the collection model should be a general model which does not assume any particular approach to constraint management and therefore the specification does not specify actions to be taken in the event of constraint violations. Clearly, any particular realisation of the collection model should extend the specification to deal with the case of constraint violation.

We will however give an example to show how the conceptual dependencies among collections can be used to propagate updates. If a value is added to a collection then the view could be taken that it should also be added to all collections of which that collection is a subcollection. Thus, if a student object is added to collection `Students` then it should also be added to the parent collection `Persons` (if it is not already a member).

We therefore introduce a general insert operation which will add a value to a specified collection and all of its supercollections of which that value is not an existing member.

INSERT

$\Delta MEMBERS$
$v? : VALUE$
$c? : OBJID$

ADD
$\forall\, c : Collections \bullet (((c, c?) \in Subcollections \wedge \neg\, ((c, v?) \text{ in } Members))$
$\quad \Rightarrow ADD[c/c?])$

The expression $ADD[c/c?]$ is a schema renaming to give a Z-schema that is the same as ADD except that the variable $c?$ is replaced by c. Thus, the insert operation first adds the value $v?$ to the specified collection $c?$ and then for each collection c such that c is a subcollection of $c?$ and $v?$ is not a member of c, an operation is performed to add $v?$ to c.

The deletion of an object from a database corresponds to removing all traces of that object from the database in such a way that consistency is maintained. For example, if a member of staff leaves a university, then we may wish to remove the representation of that staff individual and all references to that representation from the database. Further, we may wish to remove all objects that are dependent on that object from the database. The ability to perform such an operation is important in database applications and it should not be the task of the user or application programmer to keep track of all references to and dependents of objects in the database. To quote Tsichritzis and Lochovsky [TL82] page 271:

> "The machine should be the servant of the user, not vice versa!"

The main problem associated with object deletion is that of maintaining referential integrity. Referential integrity is a form of consistency that ensures there is never a reference to an object that does not exist. This can take two forms in our notion of a database:

1. *Referential Integrity Among Objects*
   If an object has an attribute which references another object, then that object should exist. For example, the object type `module` given in figure 6.1 has an attribute `text` which references an object of type `book`. Then for the database to be consistent, all the text attribute values for modules must be valid references to book objects.

```
object type module
            .......
            text :  book ;
            .......
end ;
```

Figure 6.1: Texts as Part of Modules

2. *Referential Integrity Among Collections*
   A member of one collection may be dependent upon the existence of a member of another collection according to the conceptual dependencies specified on these collections. For example,

   (a) If collection `Students` is a subcollection of collection `Persons`, then $x \in$ `Students` $\Rightarrow x \in$ `Persons`.

   (b) If `Teaches` is a relation with source collection `Staff` and target collection `Courses`, then $(x, y) \in$ `Teaches` $\Rightarrow (x \in$ `Staff` $\wedge\ y \in$ `Courses`$)$.

If an object is deleted, then the maintenance of referential integrity among objects means that all references to that object should be removed. The problem is one of locating those references. The overall object structure may be very complex in terms of objects referencing other objects and this can make the task of locating references to objects very expensive. This is one of the reasons that a number of object systems do not support the explicit deletion of objects and rather use garbage collection as the sole means of deleting an object.

The basis of garbage collection is that an object is deleted only when there are no references to that object; this may be determined by means of a reference count associated with each object. Every time an object creates a reference to another object

then the reference count associated with the referenced object is incremented. When an object deletes a reference to another object, then the reference count associated with the referenced object is decremented. Then when the reference count is zero, the object is no longer reachable and may be deleted. The outline above does not cater for the removal of unreachable cycles of objects: a specific garbage collection algorithm should be extended to deal with the detection and removal of such cycles.

If garbage collection is the only way of deleting objects, this would mean that a database application programmer would have to track down and delete all the references to an object himself. But what we want as a *database operation* is a high-level operation that effectively does this for the programmer (or end-user). Garbage collection would still be a part of the persistent object system on top of which the database system is built - but we also want to have support for the explicit deletion of objects such that database consistency is maintained.

This means that there must be mechanisms in the underlying persistent object systems to support the explicit deletion of objects. One way of doing this is to employ a notion of "tombstone objects". When an object is deleted, it is replaced by a tombstone object. Then when there is a dereference to that object, the tombstone object is encountered, an exception is raised and as part of the exception handling mechanism the reference to the object may be deleted. Again, a reference count can be used such that when it is zero, it is known that all references to the object have been removed in which case the tombstone object can be removed. This can be performed as before by the garbage collection system.

This process is illustrated in figure 6.2.

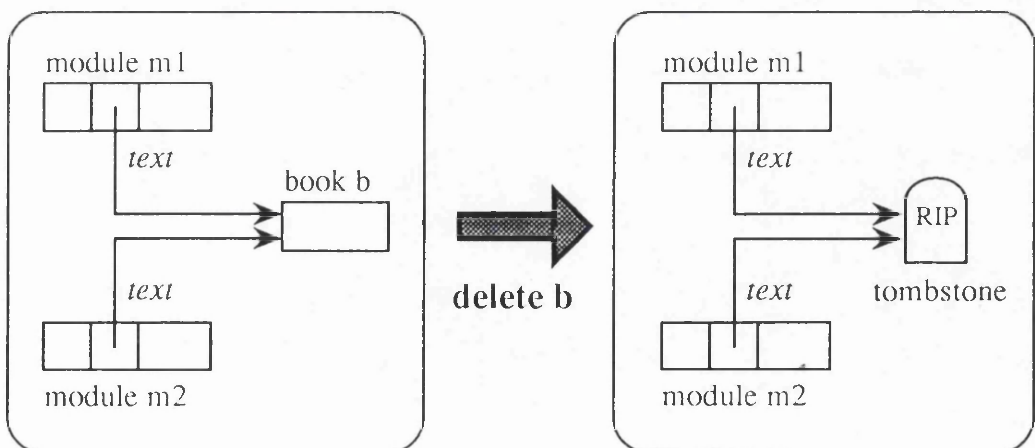

Figure 6.2: Deleting a Book Object

Objects $m1$ and $m2$ are objects of type `module` as given in figure 6.1. It is assumed that both $m1$ and $m2$ have value b for the attribute `text`. Then if object b is deleted, it is replaced by a tombstone object. Reference to the text attribute of $m1$ would result in an exception being raised as a result of which the attribute value for text in $m1$ could be replaced by some form of null value. Note that this then requires some mechanism for supporting the notion of special null pointer values.

The collection model with its direct support for the representation of relationships actually helps alleviate the problem of referential integrity among objects as it tends to eliminate "spaghetti objects" and replaces them by discrete clustered objects connected by two-way links. Relationships between objects can be represented by binary collections rather than references embedded within objects as attribute values. These binary collections could be considered as two-way links between objects and in the event that an object is deleted it is easy to locate and remove the links to that object.

Thus instead of representing the recommended texts for modules by embedding references to book objects in module objects, these relationships could be represented by a relation `Texts` with source `Modules` and target `Books` as illustrated in figure 6.3.

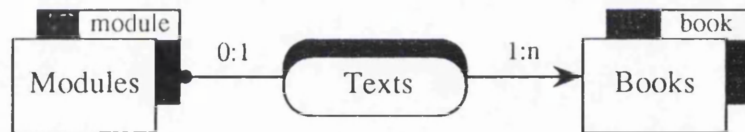

Figure 6.3: Texts as a Relation

However, it must be stressed that if the object type system is such that attributes can reference other objects, then representing relationships between objects by means of embedded references cannot be eliminated. All that can be done is to encourage users to represent relationships directly as binary collections and to only use embedded references to represent a "parts-of" relationship between objects as discussed in Chapter 4. Then it might be the case that the semantics of the deletion operation would be such that if an object is referenced by other objects, then it cannot be deleted explicitly. However, an object which is not "part-of" any other object can be deleted explicitly and all objects which are dependent on that object, through the conceptual dependencies among collections, will also be deleted: this operation can only proceed if all of these dependent objects are not "part-of" other objects.

The maintenance of referential integrity among collections under object deletion is again under the control of the constraint management system. To illustrate what might be involved in the process of object deletion, we shall give Z-schemas for three different levels of object deletion. The first is a Z-schema for the removal of a value from a specified collection.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{REMOVE}} \\
\hline
\Delta MEMBERS \\
v? : VALUE \\
c? : OBJID \\
\hline
c? \in Collections \bullet \\
((v? \in AtomicValues \Rightarrow \\
\qquad UnaryMembers' = \{(c?, v?)\} \lhd UnaryMembers) \wedge \\
(v? \in PairValues \Rightarrow \\
\qquad BinaryMembers' = \{(c?, v?)\} \lhd BinaryMembers)) \\
\hline
\end{array}
$$

Recall that *Members*, *UnaryMembers* and *BinaryMembers* are Z bags which are a special form of relations that map elements of a set into the set of positive integers, $\mathbb{N}_1$. In the case of these bags, they map collection-value pairs into an integer which denotes the number of occurrences of that value in that collection. Then the removal of a value $v?$ from a collection $c?$ is given by eliminating the pair $(c?, v?)$ from the domain of the appropriate "members" relations. Thus, if $v?$ is an atomic value, then $(c?, v?)$ is subtracted from the domain of *UnaryMembers* and this is expressed using the domain subtraction operation $\lhd$. Otherwise, $v?$ is a pair value and the pair $(c?, v?)$ is subtracted from the domain of *BinaryMembers*. As before, the value of *Members'* is determined by *UnaryMembers'*, *BinaryMembers'* and the associated constraints.

Note that in the case of a bag collection all of the occurrences of the value will be removed. Another form of the remove operation could be specified to remove only a single occurrence of a value from a specified bag collection. This further illustrates that the choice of semantics for object removal and deletion requires careful consideration.

The next level is to retain database consistency by removing the value, not only from the specified collection, but also from all of the collections that are dependent on the specified collection. This operation is referred to as deletion of a value from a collection and it is specified in the following Z-schema *DELETE*.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{DELETE}} \\
\hline
\Delta MEMBERS \\
v? : VALUE \\
c? : OBJID \\
\hline
REMOVE \\
\forall c : Collections \bullet ((c?, c) \in SubCollections \Rightarrow DELETE[c/c?]) \\
\forall r : Relations \bullet \\
((Sources(r) = c? \Rightarrow \forall v : Values \bullet DELETE[r/c?, (c?, v)/v?])) \wedge \\
(Targets(r) = c? \Rightarrow \forall v : Values \bullet DELETE[r/c?, (v, v?)/v?])) \\
\hline
\end{array}
$$

The predicate part specifies that if a value $v?$ is deleted from a collection $c?$ then it will also be deleted from all subcollections of $c?$. Further, all relations with $c?$ as either the source or target collection will have all member pairs in which $v?$ occurs deleted. The deletion of pairs from relations could violate the associated cardinality constraints. If this is the case, then the delete operation will not be performed.

The following Z-schema, *DESTROY*, deals with the destruction of a value in that it is deleted from all collections of which it is a member.

$$
\begin{array}{l}
\textit{DESTROY} \\
\hline
\Delta \textit{MEMBERS} \\
v? : \textit{VALUE} \\
\hline
\forall c : \textit{Collections} \bullet \textit{DELETE}[c/c?] \\
\forall c : \textit{BinaryCollections} \bullet \\
(\forall v : \textit{Values} \bullet \textit{DELETE}[c/c?, (v, v?)/v?]) \wedge \\
(\forall v : \textit{Values} \bullet \textit{DELETE}[c/c?, (v?, v)/v?])
\end{array}
$$

Our interpretation of the destroy operation is that not only will the value $v?$ be deleted from all collections (unary and binary) of which it is an element, but also all pairs in which $v?$ appears either as the first or second element will be deleted from the binary collections to which they belong. Thus all trace of $v?$ is removed from all collections of the database.

There are a number of points that are raised by the above specifications. The *DELETE* schema is used to demonstrate how deletions can be propagated in order that consistency is maintained. It only deals with the constraints that correspond to the subcollection and source/target dependencies among collections. It does not deal with either the constraints that may occur in collection families such as disjointness, or cardinality constraints. The specification could be extended to propagate deletions in order to preserve these constraints. In the example of figure 6.3, the deletion of a book object from `Books` could result in the deletion from `Modules` of the module objects related to that book object in order that the cardinality constraints associated with `Texts` should be maintained. This may not be quite what the user expects. In fact, one could envisage the situation where a large part of a database was deleted automatically by the system in order to preserve consistency. For this reason, the widespread use of update propagations requires caution and it is wise to have some sort of control and confirmation procedures in the automatic propagation of updates.

All of the three schemas above deal only with the removal of objects from collections: they do not deal with the destruction of the actual object itself. In other words, an object could be removed from all the collections to which it belongs but it would still be in the persistent object store and might be referenced by other objects of the database. As discussed above, the policy and mechanisms for the deletion of an

object from the persistent object store are determined not by the collection model but by the underlying persistent object system.

6.2 Object Evolution

Support for object evolution has two main requirements. Firstly, the system must be able to move objects from one collection to another collection. This may involve the deletion of the object from the first collection (and all references to the object as a member of that collection) and then the insertion of that object in the second collection. The deletion of the object from the specified collection is similar to object deletion as specified in the previous section in that an object is removed from the specified collection and all dependent collections and the object itself is not deleted.

Secondly, there must be some form of control on how objects can evolve which reflects how entities may evolve in the real world. For example, we might be happy to have a student entity evolve into a staff entity, but we would be unhappy to have a student entity evolve into a department entity. Why is this the case? We regard student entities and staff entities as being similar "kinds" of entities, namely person entities, and that being a student or a member of staff corresponds to roles that person entities can adopt at a particular point in time. However, student entities and department entities are fundamentally different kinds of entities. Thus the classification of entities involves two forms of classification: a fundamental classification of entities into kinds and a further classification of entities into roles. An entity can change roles, but it cannot change kinds.

In [Zdo87], Zdonik introduces a similar distinction between these two forms of classification. He does not separate out the notions of typing and classification as has been done in the collection model and therefore uses the typing system as his means of classifying entities. Then he distinguishes between the two forms of classification by the introduction of *essential types* which are similar in their purpose to our notion of kinds in that they represent the fundamental classification of entities.

We propose an extension to the collection model which partitions unary collections into classification categories `Kinds` and `Roles` to represent the two forms of classification and these are then used to control object evolution. Those unary collections which are regarded as representing a fundamental classification of objects corresponding to the basic kinds of real-world entities will be members of the set `Kinds`: those that correspond to roles that can be adopted by entities will be members of the set `Roles`.

Roles represent a flexible and dynamic form of classification in the sense that an object may change its roles to reflect the change of properties and behaviour that

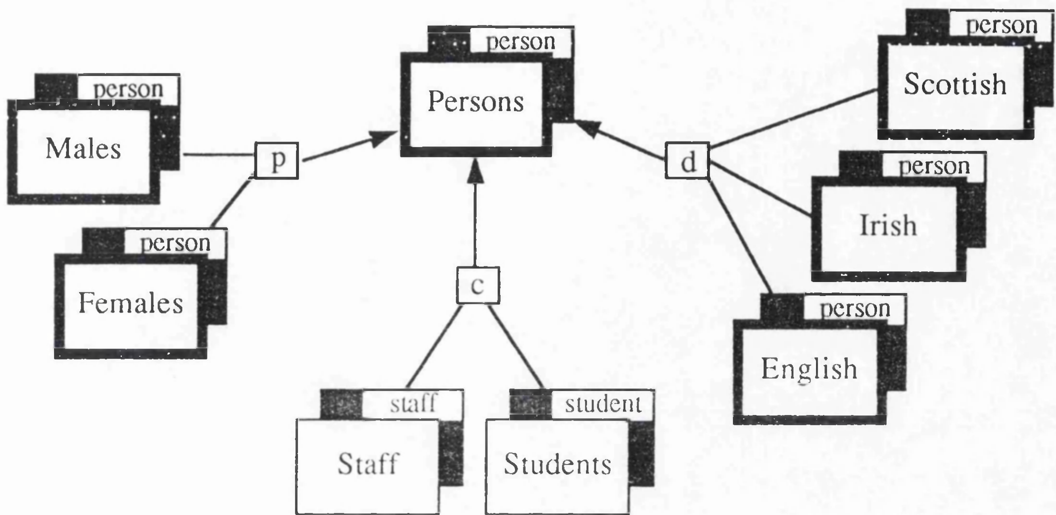

Figure 6.4: A Classification Structure for Person Entities

a real-world entity may exhibit in the course of time. However, kinds represent a fixed form of classification and an object may only acquire additional kinds through the introduction of new kinds into the classification structure - or by establishing more characteristic properties of an object in a way that allows its set of kinds to be refined.

For example, consider the classification structure of figure 6.4 which models person entities. The graphical notation of BROOM is adapted to distinguish the classification categories of collections. A unary collection which represents a kind has a bold outline.

Once an object has been established to be of a particular kind, then it will remain of this kind - although it may possibly assume additional kinds. In our example, by choosing `Males` and `Females` to be kinds, we regard these as fixed classifications: this means that an object would not be permitted to migrate from one to the other. Further, these kinds form a partition of `Persons` which means that every person object must be assigned to one or other kind at the time of creation. Another application might allow for sex changes - in which case collections `Males` and `Females` would be roles rather than kinds.

Since in our example, collections `Staff` and `Students` are both roles associated with kind `Persons`, then these are not considered as fixed classifications and it would be permissible for an object to migrate from `Students` to `Staff`. Our model also has three disjoint kinds representing nationalities. These do not form a cover of `Persons` which means that not every person must be classified as belonging to one of these

nationalities. However, once classified as belonging to a nationality then it would not be possible for a person object to evolve from one nationality to another. It is possible for a person object who is not classified as belonging to one of the three nationalities to later be added to one of these three kinds when information about that person's nationality becomes available.

This notion of immutable classification according to kinds may seem rather strict. What happens if someone is classified as Scottish rather than Irish by mistake? Of course, one should be able to recover from and correct such mistakes - but this should be separate from the "normal" operation of the application system and require special action. It is not unlike entering a tuple in a relational database system with the wrong key attribute. The only way to recover from such a situation may be to delete the tuple and redo the insert operation with the correct values. Providing system support to copy values and maintain referential integrity would be beneficial - but still such an operation should be exceptional rather than normal and perhaps require special access privileges. The main point is that in many object-oriented database systems the effective deletion and re-creation of an object would be the only way to change the classification of an object even under normal operation. We propose that there should be mechanisms to support such object evolution - but also that we might want to place controls on the form that this might take under normal operation.

As can be seen from the above example, object evolution in the context of the BROOM model involves the migration of an object within a classification structure. Certain migrations may also require a metamorphosis of the object from one type to another. Consider the case of an entity which evolves from being a student to being a member of staff. Then the object representing that person must migrate from the collection `Students` to the collection `Staff`. But the member types of these collections are not the same. This means that the representation of that individual will retain all the properties of type `person` but they will lose all the properties specific to type `student` and they should gain properties that are specific to type `staff`. We call this process of an object undergoing a change in its representation object metamorphosis.

If a persistent object system is to support object evolution, then it must be able to support object metamorphosis. An object should be able to change the type of its representation while retaining the same object identity. Unfortunately, this is not supported in most existing persistent object systems. Note that object metamorphosis is not the same as the casting of objects in, for example, C++ [Str87]. Casting merely provides an alternative view on an object and the object itself does not change. This can result in violations of type safety in that it allows an object of type T_1 to be considered as an object of type T_2 where T_2 is a subtype of T_1 - but if an attempt is made to access a T_2 property of the object, then a hidden type violation may occur. Further we do not want to have to write specific methods to allow an object to metamorphose between a specific pair of types. Rather what should be supported is a general mechanism to allow any object to change its type.

If, however, such support is not provided in the underlying object system, then one possibility would be to severely restrict object migration in that objects could only migrate between collections which had the same member type.

Two models which do have support for object metamorphosis are Galileo [Ghe90] and COCOON [LS92]. In Galileo, they propose a *specialize* operation which allows an object to retain its existing types and gain additional subtypes. COCOON supports a less restricted form of metamorphosis with two operations: the *gain* operation allows an object to gain types and the *lose* operation allows an object to drop types. The COCOON model also separates typing from classification and they propose the use of the *gain* and *lose* operations together with strict class definitions to support object migration. However, the COCOON model does not support any form of migration control. Since COCOON has a most general object type `ObjectT`, then for a given object x, the operation

```
lose[ObjectT](x)
```

would result in the destruction of object x.

We now present details of how the categorisation of unary collections into kinds and roles can be used to control object evolution in the context of the collection model.

As discussed in Chapter 3, the subcollection relation $\preceq$ on collections is a pre-order. In particular, the subcollection relation is a pre-order on the set of unary collections which we will denote by $\mathbf{U}$. Thus:

1. for all $C \in \mathbf{U}$, $C \preceq C$ (*reflexive*)
2. for all $C_1, C_2, C_3 \in \mathbf{U}$, $C_1 \preceq C_2$ and $C_2 \preceq C_3$ implies $C_1 \preceq C_3$ (*transitive*).

We impose the restriction that every classification structure on unary collections must be "rooted". In other words, for every unary collection C there must be a unique maximal collection C_1 such that $C \preceq C_1$ and for all $C_2 \in \mathbf{U}$ such that $C \preceq C_2$, we have $C_2 \preceq C_1$. Then the set of all maximal collections is denoted by $\mathbf{M}$.

We can define a relation $\cong$ on $\mathbf{U}$ such that for $C_1, C_2 \in \mathbf{U}$, $C_1 \cong C_2$ iff the maximal collection of C_1 equals the maximal collection of C_2.

Then, trivially, $\cong$ is an equivalence relation on $\mathbf{U}$. The equivalence classes induced by $\cong$ correspond to the classification structures of unary collections. Thus for any $C \in \mathbf{U}$, the equivalence class under $\cong$ for C is the set of all unary collections that are in the same classification structure as C, i.e. all those collections with the same maximal collection.

Each unary collection is specified as either a kind or a role. Then the set of unary collections $\mathbf{U}$ is partitioned into the sets $\mathbf{K}$ and $\mathbf{R}$, i.e. $\mathbf{U} = \mathbf{K} \cup \mathbf{R}$ and $\mathbf{K} \cap \mathbf{R} = \varnothing$.

In addition, we impose the restriction that the collection at the root of a classification structure must be a kind, i.e. $\mathbf{M} \subseteq \mathbf{K}$. This is reasonable since two different classification structures should correspond to fundamentally different kinds of entities of the application domain.

Each unary collection has an associated set of kinds and an associated set of roles and these are given by the functions **kinds** and **roles**, respectively. Let $C \in \mathbf{U}$. Then

$$\textbf{kinds } C = \{K \mid K \in \mathbf{K} \text{ and } C \preceq K\}$$

$$\textbf{roles } C = \{R \mid R \in \mathbf{R} \text{ and } C \preceq R\}.$$

Consider the classification structure for university persons given in figure 6.5.

In this figure, there are four kinds and five roles such that

$\mathbf{K} = \{$ `Persons`, `Professors`, `Seniors`, `Postgrads` $\}$
$\mathbf{R} = \{$ `Staff`, `NonProfs`, `Lecturers`, `Students`, `Undergrads` $\}$.

First let us consider the interpretation of the designation of the various collections as kinds and roles. The classification structure of figure 6.5 classifies persons within a university system and accordingly the root of the structure is the maximal collection `Persons`. Our restriction that each classification structure must have a unique maximal collection ensures that there is no overlap with this structure and any other classification structure. Further, since, as required, `Persons` $\in \mathbf{K}$, every object that will belong to this classification structure must belong to `Persons` and can never be removed from that collection unless deleted from the database. In other words, an object can never migrate to a different classification structure.

The two child collections of `Persons` are designated as roles. This means that these classifications are not fixed. An object which belongs to `Students` can migrate to `Staff` and vice versa. In either case, the object will have to undergo a metamorphosis from one type to another. For example, an object migrating from `Students` to `Staff` will have to metamorphose from being a `student` object to a `staff` object.

`Staff` has two child collections - one designated a kind and the other a role. How should we interpret the situation of a role collection having a subcollection which is a kind? The interpretation used is that the kind represents a fixed classification in the context of the parent role. For example, if a person object belongs to kind `Professors` then it must remain a member of that collection as long as it belongs

to the role Staff. This then models a situation in which once a member of staff is assigned a professorship, they can lose that status only when they are no longer a member of staff of the university: they cannot be demoted. However, a staff object which belongs to NonProfs can be migrated to Professors. Therefore our classification structure represents a university system in which only promotion is allowed: a lecturer can be promoted to senior lecturer, and a senior lecturer can be promoted to professor - but a professor cannot be demoted to a lecturer. Similarly for students we have the case that an undergraduate can become a postgraduate but not the other way round.

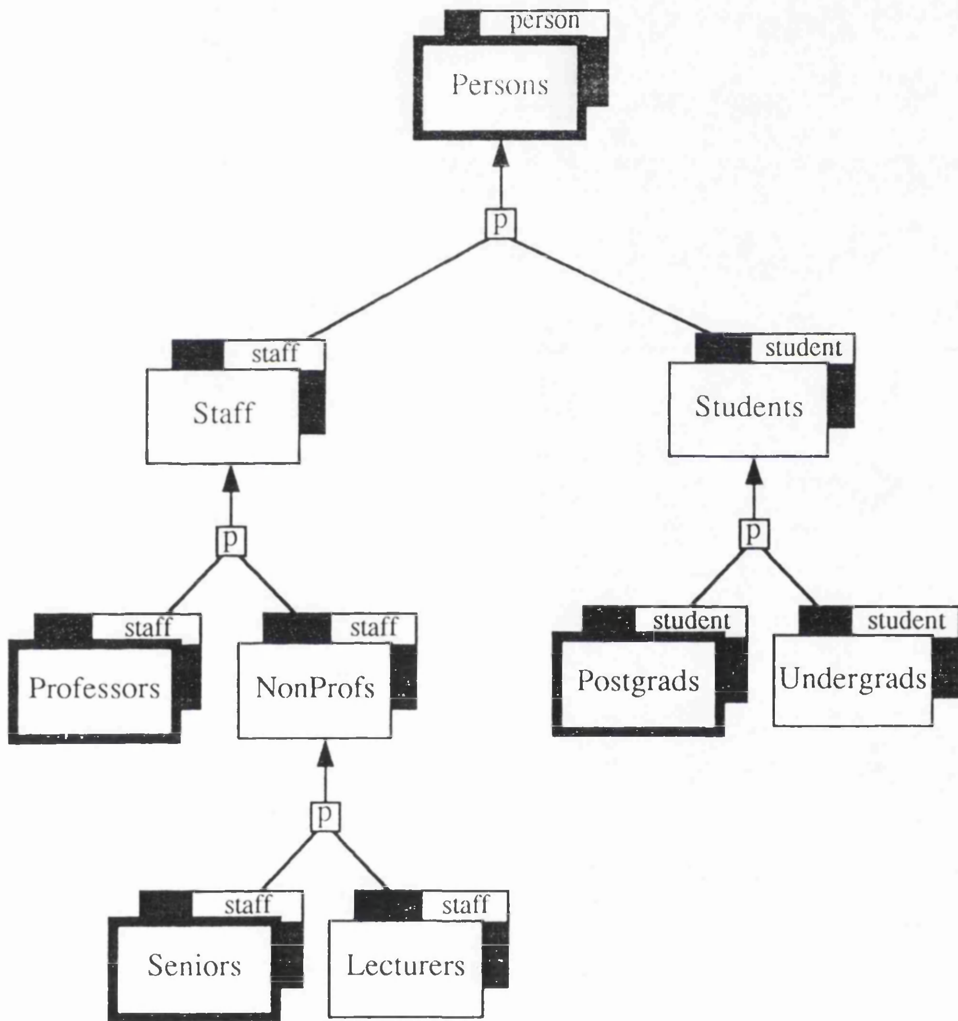

Figure 6.5: Classification Structure for University Persons

Having outlined the required operation of object evolution by consideration of the university example, we now specify it formally.

Let C_1 and C_2 be unary collections and let $o \in C_1$. Then we specify the migration of o from C_1 to C_2 by the operation

$$o :: C_1 \rightarrow C_2$$

This operation is valid in the case that:

1. there is no $C \in \mathbf{U}$ such that $o \in C$, $C \preceq C_1$ and $C \not\equiv C_1$
2. $\forall K \in (\ \textbf{kinds}\ C_1 - \textbf{kinds}\ C_2).\ \exists R \in \textbf{roles}\ K$ s.t. $R \notin \textbf{roles}\ C_2$

The first of these conditions checks that there are no subcollections of C_1 of which object o is a member. In fact the conceptual dependencies between collections and their subcollections would ensure that an object could not be removed from C_1 unless it was also removed from all subcollections of C_1 as was discussed in section 5.1 with respect to object deletion.

The second condition specifies that an object migration in which an object loses a kind is valid only in the event that the object also loses a parent role of that kind. Thus if an object is deleted from a collection $C_1 \in \mathbf{K}$, then it must also be deleted from some collection C_2 where $C_1 \preceq C_2$ and $C_2 \in \mathbf{R}$.

For example, given the classification structure of figure 6.5, consider the migration

o :: Postgrads $\rightarrow$ Lecturers

We have

kinds Postgrads = {Persons,Postgrads}
roles Postgrads = {Students}
kinds Lecturers = {Persons}
roles Lecturers = {Staff,NonProfs,Lecturers }

Then

kinds Postgrads − **kinds** Lecturers = {Postgrads}

but

roles Postgrads = {Students}

and

Students $\notin$ **roles** Lecturers

therefore the migration operation is valid.

However the migration

$$o :: \texttt{Postgrads} \rightarrow \texttt{Undergrads}$$

would not be allowed since

$$\textbf{kinds}\ \texttt{Postgrads} - \textbf{kinds}\ \texttt{Undergrads} = \{\texttt{Postgrads}\}$$

and

$$\textbf{roles}\ \texttt{Postgrads} = \textbf{roles}\ \texttt{Undergrads}$$

The object o would lose its classification as kind `Postgrads` but would retain its contextual role `Students` and the migration would therefore be disallowed.

The migration operation involves the removal of an object from some collections and its addition to other collections and we specify it in the Z-schema *MIGRATE*. There are three input variables: $o?$ is the object to be migrated, $c_1?$ is the collection from which it will migrate and $c_2?$ is the collection to which it will migrate. It is assumed that *DB* has been extended to include *Kinds* and *Roles* which are sets of *OBJID* which partition *UnaryCollections*, and that predicates are included to specify the conditions described earlier in this section.

```
MIGRATE
ΔMEMBERS
o? : OBJID
c1?, c2? : NAME
---
c1? ∈ UnaryCollections
c2? ∈ UnaryCollections
(c1?, o?) in Members
∀ k : Kinds • (k ∈ (kinds c1? − kinds c2?)
      ⇒ ∃ r : Roles • (r ∈ roles k ∧ r ∉ roles c2?))
∀ c : Collections •
((((c, c1?) ∈ SubCollections ∧ ¬ (c, c2?) ∈ SubCollections)
      ⇒ DELETE[o?/v?, c/c?]) ∧
(((c, c2?) ∈ SubCollections ∧ ¬ (c, c1?) ∈ SubCollections)
      ⇒ INSERT[o?/v?, c/c?]))
```

The conditions check that the migration operation is valid as described above. Then the object will be removed from all collections of which $c_1?$ is a subcollection and $c_2?$ is not, and, added to all collections of which $c_2?$ is a subcollection and $c_1?$ is not. The Z-schema also assumes that the specification of DP has been extended to include definitions of the functions **kinds** and **roles** as defined previously.

The above Z-schema makes no allowances for object metamorphosis as discussed previously. When an object is inserted in a collection using the *ADD* operation then

that operation will take place only in the event that the object is an instance of the member type of that collection. If object migration is allowed such that the object can actually "gain" types then the object representation must change and the user may have to supply additional attribute values for the object. Then the above Z-schema could be extended to have as an additional input value a list of new attribute values and to specify the forms of metamorphosis required.

We conclude this section by noting that the above proposal is a simple mechanism for controlling the forms of object evolution allowed. Clearly such a mechanism is limited in terms of expression of the possible evolutionary paths that entities may take in the real world. If a more powerful and general evolution model is required, then a simple extension of the collection model will not suffice. Rather some evolution model should be supported on top of the collection model. One possibility would be to support a constraint model that incorporated a form of dynamic constraints that could express evolutionary constraints. Alternatively, a separate mechanism specifically designed to cater for the control of the evolutionary aspects of objects could be supported: for example, a mechanism similar to that of scripts proposed in the TAXIS language [BMW84] which are used to specify the life-cycles of objects.

6.3 Schema Evolution

Schema evolution involves changes to the description of a database. These may arise either because of changes to the application reality that the database represents or because of changes in the requirements of the application system. In [VH91], Ventrone and Heiler examine one particular form of change - domain evolution. Although they focus on only one of the many forms of schema evolution, their paper provides a useful overview of the various reasons for change and the problems of handling such changes.

Sometimes schema evolution is confused with type evolution, but strictly type evolution is only part of schema evolution. This is particularly true in systems such as the one proposed here where the typing system is separated from the classification system and constraints are not part of type definitions.

Our aim in this section is not to describe detailed proposals for the support of schema evolution but rather to indicate the forms of schema evolution that arise in connection with the collection model.

Banerjee et al [BCKK87] describe a general taxonomy for schema changes in object-oriented systems. Here we present a very simple taxonomy with a view to distinguishing between changes to the type system and changes to the database structure in terms of the collections and their conceptual dependencies.

1. *Type Changes*
   (a) add a new type definition
   (b) remove a type definition
   (c) update a type definition
2. *Collection Changes*
   (a) add a new collection
   (b) remove a collection
   (c) change the member type of a collection
3. *Database Changes*
   (a) update a collection family
   (b) change the source/target of a binary collection
   (c) update the cardinality constraints on the source/target of a binary collection

The problems of schema evolution, like those of object evolution, are first of all to decide on what controls there should be on the forms of evolution supported and then to provide mechanisms to support the evolutionary process. For example, the view that a database can evolve such that collections can gain new subcollections but not supercollections would restrict the forms of evolution to be supported. As already seen in the previous section, in practice the restrictions on the forms of evolution supported are often not so much a matter of policy but rather a matter of limitations in terms of system support.

If values created during the existence of one schema state are to be retained and accessed during the existence of a future schema state, then the schema states must also be retained. The question then arises as to how to interpret a query over the present and past schema states.

As stated previously, the issue of schema evolution is very closely connected to the problem of type evolution. In the taxonomy given above, clearly those categorised as type changes correspond to type evolution and if these are to be supported, then this must be done at the level of the persistent object system. The ability for a collection to change its member type may also be regarded as a type evolution issue.

While the addition of new collections should not be problematic, other changes such as the alteration of classification structures or changes to cardinality constraints present difficulties not only in terms of engineering but also in terms of interpretation. Although these changes are not in a strict sense type evolution, they are typical of the sorts of issues that have been addressed by researchers working in the areas of both

type and schema evolution. An annotated bibliography of significant papers in these areas has been produced recently by Roddick [Rod92].

The most significant papers in connection with evolution in object-oriented databases are those reporting proposals for the ORION [BCKK87] and GemStone [PS87] systems and the work by Zdonik and Skarra [SZ86], [SZ87] which arises from supporting versioning of data.

The two general approaches that have been proposed for schema evolution can be categorised as conversion at schema update and conversion at retrieval. GemStone [PS87] adopts the former approach and converts object values in accordance with the new type definition at the time that the type definition changes. The latter approach is adopted both in the ORION system [BCKK87] and the proposals of Zdonik and Skarra.

Chapter 7

Realising the Collection Model

The collection model is a general model for the structure and operation of a database system which comprises interrelated collections of objects. While it was designed in the context of a particular project, Comandos, the proposed model is not dependent upon the Comandos architecture [CBHdP93]. Neither does the model assume a particular style of use or application.

This can be likened to the relational data model which has been implemented on a whole variety of platforms. Some implementations have been devised for single-user database applications on personal computers while others are intended for large commercial multi-user systems on a large mainframe or possibly distributed over a network of machines. Similarly the access to the database could take various forms such as through a general programming language with calls to the database management system or by some form of end-user direct manipulation graphical query language.

Likewise a particular realisation of the collection model could be as sophisticated and complex or as naive and simple as the intended applications and style of usage requires. The model is independent of the implementation platform and could support various forms of access languages.

In this chapter, two realisations of the collection model are described. The first, COLLEEN, is a prototype single-user database management system based on the collection model. It was developed primarily as system to prototype the collection model and to be used as a platform for experimentation in the areas of constraint management, evolution and database design tools.

The second realisation was done in the context of the Esprit project Comandos [CBHdP93], [HN91a], [HN91b]. The Comandos project is concerned with the development of an integrated support environment for the construction and management of

distributed applications involving persistent data. The Comandos platform is based on an object-oriented view of the environment and is provided in the framework of multi-vendor distributed systems. Potential applications include Office Information Systems, CAD/CAM systems and software development systems. Typically, such systems will involve large collections of complex objects of the same type and these collections will be interconnected: it is the role of the Object Data Management Services (ODMS) to provide mechanisms to support the management of these collections. The ODMS and its associated tools are based on the proposed collection model.

The contrast in these two systems both in terms of their scale of implementation and style of usage demonstrates the generality of the collection model and its independence from the underlying type system and implementation platform.

The first section provides an overview of COLLEEN and discusses some general issues raised by the system in terms of further experimentation that could be carried out using the system. Then the second section presents a brief overview of the general Comandos architecture and the implementation strategy of the ODMS. We note that the implementation of the ODMS was undertaken by the Comandos ODMS research group at the University of Glasgow and that section 2 draws on a number of articles that have been co-authored by members of this group.

7.1 COLLEEN

COLLEEN is a single-user database management system which provides a single environment for both schema and data manipulation. It was implemented as a prototype to test the ideas of the collection model and also as a platform for experiments in the area of database design tools, constraint management and evolutionary models. The system therefore employs a naive representation of objects and collections and has no support for the management of large collections.

The system was implemented on the Macintosh in the language LPA MacProlog [LPA91]. LPA MacProlog is an extension of Prolog which provides system defined predicates for the management of windows, menus and dialogue boxes. It therefore provides an ideal environment for the fast prototyping of systems and their user interfaces.

The top level interface to COLLEEN consists of pull-down menus in the style of Macintosh applications. There are four top level menus which allow the user to select from the four categories of operations - database, schema, data and query operations. To present an overview of the system, we shall consider each of these categories in turn.

Database Operations

A user works in the context of a single database at any time. The system stores the name of a database, a short textual description of its contents and the name of the file in which the database is saved. The database menu has options to open, create, save and close a database. Initially, only the open and create options are active.

If the user opts to open a database, then they are asked to select from a menu of existing database names and the corresponding database is opened. The other menus for schema, data and query operations are then installed.

Schema Operations

The schema operations allow the user to create and update a BROOM model of a database. The schema consists of descriptions of types, collections, relations and collection families.

A very simple type system is supported in which all types are object types comprising one or more attributes. An attribute can be of type string or integer. In creating a type, the user is asked to supply a list of attribute names and their types.

If the user wishes to create a collection, then once the name of the collection has been specified, a form is displayed in which the user specifies whether the collection is unary or binary, whether it has set or bag behaviour, if it is ordered and whether it is a kind or role.

For unary collections, a member type must be specified and the user is asked to select a type from a menu of existing types. A menu item "**...new type**" can be selected if the member type does not exist in the schema and the user must then create the type.

In COLLEEN, independent binary collections are prohibited. Binary collections can exist only as representations of relationships between a source and target collection. Therefore, if a collection is specified as binary, then the system will ask the user to select the source and target collections from a menu of existing collections. A menu item "**...new collection**" can be selected if the source or target collection does not exist in the schema and the user must then create the collection.

If a collection is specified as ordered, then once the member type of the collection is established, the user is asked to select a list of one or more attributes of that type which specify the ordering. In other words, the ordering of collections is based on the usual lexical orderings of string and integer values. If a collection is binary, then the selected attributes may appear from the member types of the source and target collections.

A collection family is specified by selecting a set of parent collections, a set of child collections and then selecting a constraint condition. The parent collections must have compatible member types. The child collections must then be selected from those collections which have member types that are subtypes of all of the member types of the parents. As before, these menus contain an option "**...new collection**" which can be selected if one or more collections of the family do not exist in the schema.

The system is designed so that the schema information can be provided in any order. A user could work bottom up by providing all type information, then all collection information and finally all collection family information. Alternatively, they could work top down by defining collection families and the system would automatically request collection and type definitions as and when required.

Data Operations

Data operations may involve either objects or relationships. An object may be created, updated, deleted or it may migrate between collections. A relationship between two values may be created or deleted.

When an object is created, the type of the object must be selected and then a form is generated to supply the attribute values of that object. The user must then select one or more collections with compatible member types and the object will be inserted in these collections. The object will be inserted automatically in any collections of which these selected collections are subcollections.

An update operation on an object displays a form for that object's attribute values which the user can then edit. The question is how the user specifies the object to be updated. This is done by means of a query expression. If the expression returns more than one object, the user will be notified and if confirmation is obtained each of these objects will be updated in turn. Objects to be migrated or deleted are specified in a similar manner.

An object may migrate between two collections of compatible member types. If the migration requires that the object change its type, then a form will be generated to show the new type of the object and the user must supply any new attribute values that are required.

When an object is created, the user is asked whether they wish to create any relationship involving that object. If so the user is asked to specify the name of the binary collection and then a query expression to identify the objects to which it will be related. If a relationship is to be created on existing objects, then the user must identify the binary collection and then query expressions for both the source and target objects.

In any of these operations, if the constraints on the database are violated then the operation will be disallowed and the effect of any intermediary updates undone.

Query Operations

A query may be expressed in terms of the collection algebra. Each query has a name and the result of evaluating a query is a temporary collection of that name. The collection is temporary in that it is not saved when the database is saved. A query may be explicitly saved in the database in which case the query expression, and not the value of the query, is saved in the database when the database is saved. The result of query evaluation remains in the database until the database is closed. This means that if the database is updated, then the query will not be re-evaluated automatically. However, the user can request re-evaluation at any time. A query expression may reference other queries.

Issues

Although the COLLEEN system is very simple, it does raise a number of interesting issues. The method of creation of a schema and also of objects is very straightforward and also very flexible. The user has the freedom to specify types, collections, relationships and collection families in any order. The system will then use the inherent constraints of the model to request any necessary further input from the user. For example, if a user specifies a new binary collection, then the system will insist that source and target collections be specified, if necessary creating new collections. The creation of new collections may in turn result in the creation of new types. It might be argued that it is better to impose a fixed order for specification of the various sorts of elements of a schema. However, we feel that this can be too restrictive and prefer this more flexible approach. In any case, the current system provides an ideal framework for monitoring users in order to analyse their characteristics in terms of their preferred order of schema input and also the number of updates to the schema that result.

The process of identifying a particular object to be updated or deleted is rather cumbersome: the same is true in the specification of relationships between objects. This highlights a problem of object-oriented database systems: How can you identify a specific object of interest? In relational systems, a tuple can be identified uniquely by the values it contains. However, the feature of an object based system is that two or more objects may have the same set of attribute values and the system can distinguish them by their unique object identifiers. But these object identifiers are not meant to be visible at the user level. The user will identify an object by means of a combination of its attribute values and its relationships to other objects. Then, as with relational systems, it may be useful to support the notion of key. A key will

be a set of attributes which can be used to uniquely identify objects. Then object keys would provide a more convenient way for users to specify an object of interest.

Note that object keys are not the same as object identifiers in that the former is visible at the user level while the latter is invisible to the user. Not all object types will have associated keys but rather only those which are likely to require direct retrieval by a user. These keys will often correspond to some semantic property of the application domain. For example, most systems that deal with person entities require some form of unique identification such as Social Security Number and as such there should be a corresponding key attribute of person in the application system. If we also had objects to represent contracts of persons, then it is possible that these would have no uniquely identifying attribute but rather would be identified through the associated person and possibly attributes of contract such as date.

Support for object keys is, of course, not only a matter of convenience: it is also a useful form of uniqueness constraint on attribute values. We may want to ensure that no two projects are given the same name. This raises another interesting question. What should be the scope of such a constraint? Should the constraint apply to a particular collection, to all collections containing objects of the corresponding type or to all objects of that type created within the system?

Having a single environment for both schema and data, and allowing free movement between operations on these, immediately raises the issue of schema evolution. The current system is very restrictive in the sense that it prohibits changes to types and collections that are inhabited. In other words, if there are any existing instances of a type, or members of a collection, then the descriptions of these cannot be changed. Thus, in effect schema evolution is not supported other than through the addition of schema information in terms of creating types and collections. COLLEEN would provide an excellent platform for experimenting with various proposals for schema evolution mechanisms.

The constraint model and management in COLLEEN is very basic. As stated previously, one of the objectives of producing the prototype was to allow experimentation in these areas. In particular, some experiments have been carried out in connection with automatic update propagations and system initiated dialogues to specify actions to be taken in the event of constraint violations.

7.2 The Comandos ODMS

The Comandos project has integrated operating system, programming language and database technologies to produce a platform that supports the development of applications involving access to reliable, persistent data in a distributed and multi-

language environment. The system is open in the sense that it supports access to pre-existing applications and information systems interworking with non-Comandos environments.

An overview of the general Comandos architecture is given in figure 7.1. A detailed description of the Comandos architecture is given in [CBHdP93].

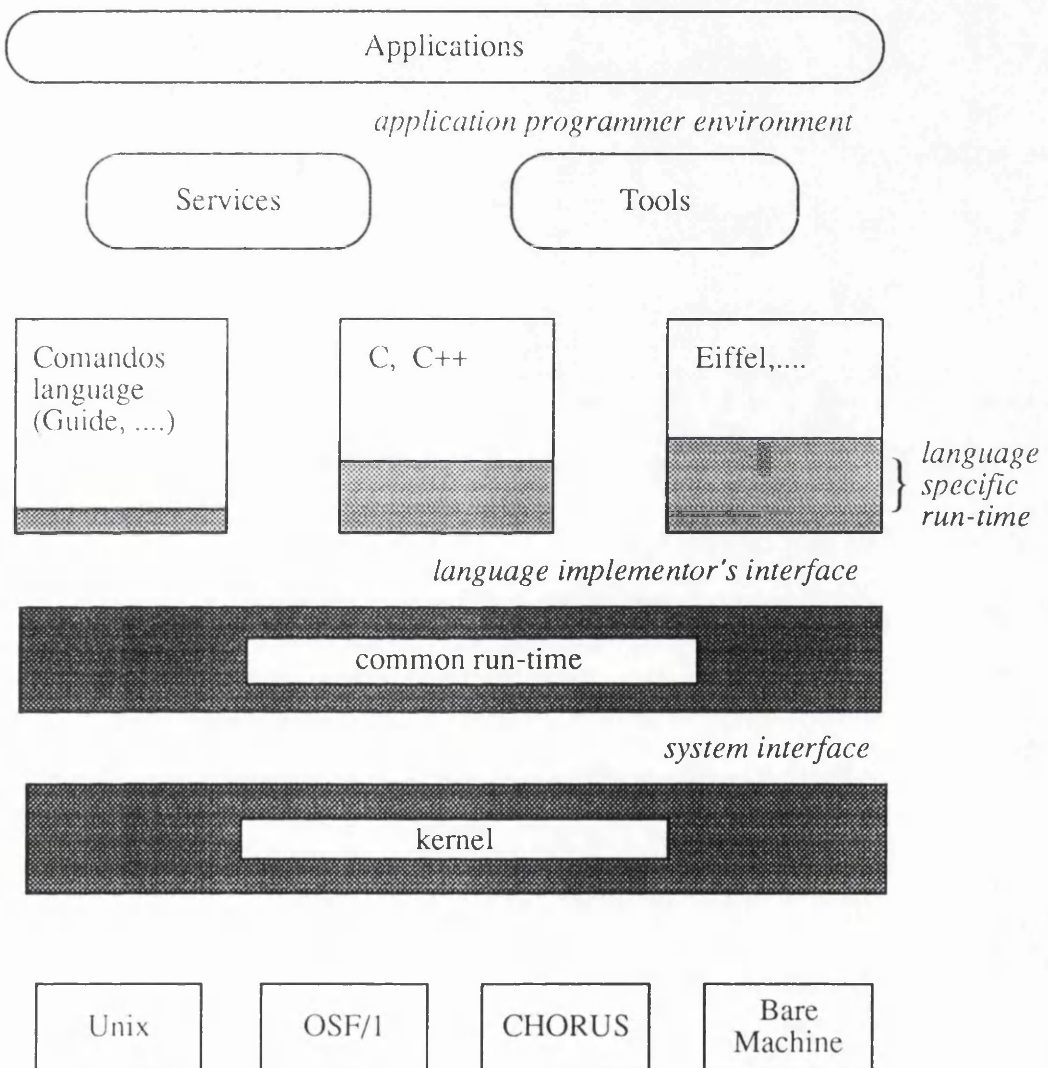

Figure 7.1: The Comandos Architecture

The Comandos Virtual Machine consists of a kernel, a common run-time system and a set of language specific run-time environments. The basic Comandos virtual machine is extended with a set of services and administration tools. The services include the Object Data Management System (ODMS) for the management and efficient retrieval

of collections of objects, a Distributed Directory Service for global object naming and a Type Manager for the management of user-defined types.

Both transient objects - those whose lifetime is limited to the execution of a program - and persistent objects - those whose lifetime is independent of that of the programs which use them - are provided within a unified framework. In Comandos, persistence is defined by reachability from a specified set of root objects. The Storage Subsystem of the kernel is responsible for the maintenance of a distributed persistent store. Persistent root objects are those having a known global name. A local object may be promoted to be a global root object thereby providing it with a global name and allowing it to persist beyond program execution.

Objects may be either atomic or non-atomic. Atomic objects are subject to a transaction mechanism which ensures their reliability and durability. The system cannot guarantee the consistency of non-atomic objects. A non-atomic object may be promoted to an atomic object but the reverse procedure is not possible. A programmer may choose to make an object non-atomic to reduce the overheads of the transaction mechanisms - however, the programmer must then realise that there is a risk that the object will become corrupted or that the persistent store will become inconsistent. Generally, it would be advised that all persistent objects be atomic.

The Comandos transaction model is derived from that provided in the RelaX project [SKME89]. It provides support for nested distributed transactions based on a two phase read/write locking protocol with a lock promotion scheme. Certain default mechanisms are supported but greater flexibility and control can be achieved by programmer control through calls to the transaction management subsystem.

Full distribution transparency is provided by the Comandos Virtual Machine. However, sometimes full transparency is not desirable and in such cases the application programmer may have control over distribution mechanisms.

The Comandos system supports multi-language working and this is achieved through the services of Type Manager which manages type information using a canonical type representation. The currently supported languages include two extensions of C++ which support persistence and distribution (C** and EC++), Eiffel and an object-oriented language, Guide, which was designed and implemented within the context of the Comandos project.

The ODMS has been realised as a layered, portable toolkit. Each layer corresponds to a level at which a programmer may wish to use the toolkit. These layers are outlined from the bottom up as follows:

The Aggregate Layer This layer provides a series of data storage abstractions, or "basic access methods", including B+-tree, dynamic hash table and list. It also

supports the concept of elements within an aggregate being indexed on some property of the element.

The Bulk Layer This layer provides abstractions for bulk data types i.e. set, bag, sequence and relation. For any instance of a bulk object, an implementation structure is chosen from the Aggregate Layer. Note that relations can have set, bag or sequence behaviour.

The Collection Layer It is this layer which realises the collection model. It provides the abstractions of the Bulk Layer augmented with multiple implementations, which are either extensional or intensional, constraint maintenance and query language support.

An application programmer wishing to use the full functionality of the ODMS will use the toolkit at the level of the Collection Layer. However, if a programmer wishes to use the abstractions of the Bulk or Aggregate Layers, then they may choose to access the appropriate level of the toolkit. For example, if a database engineer wanted to build an implementation of some data model other than the collection model, then they could use the access methods of the Aggregate Layer to support their storage structures, and possibly some or all of the bulk abstractions provided by the Bulk Layer. In this way, the ODMS could be regarded as an extensible database system in that it provides a toolkit to support the construction of database management systems. Further, since the abstractions are provided by the means of a class library, it is possible to extend the library classes with access methods or specialised forms of collections tailored to specific application requirements.

Logical collections at the Collection Layer can have multiple representations, some of which are extensional and some intensional. Extensional representations are explicit Aggregate Layer access methods. Intensional representations are similar to the notion of a view in that they correspond to query expressions.

Queries are represented by active query objects. The idea is that queries may be represented by abstract syntax trees, and, being in the object-oriented context, functionality may be associated with these trees. Active queries are self-evaluating and self-optimising. A query is evaluated by evaluating the root node, which in general involves scans, membership checks etc. of subordinate nodes. Evaluation can be performed lazily so that data is obtained on demand thereby minimising the need for intermediate results.

Optimisation involves transforming the syntax tree into a processing tree where the leaf nodes correspond to physical representations of collections. Active query objects, like any other objects, may persist and subsequently be retrieved and have operations invoked on them. Consequently, it is possible to optimise and evaluate such queries as and when required. Thus depending on what information is available about the collections, and changes that are made to the representations of collections, one can

choose to optimise at compile-time, run-time or at other appropriate times such as after bulk loading data into a database.

Active queries provide a framework in which queries from a range of sources may be handled uniformly. Thus, the approach supports queries derived from embedded query expressions in programs, from end-users in the form of ad hoc queries and from view definitions. Further details of active queries are given in [BHN92].

BROOM classification structures are represented by means of exclusion terms. An exclusion term is a collection expression which should evaluate to the empty collection in the case that the associated constraint is satisfied. For example, given the collection family F = ({Persons}, {Staff, Students}, partition), then the associated exclusion terms would be:

```
Staff − Persons
Students − Persons
Staff ∩ Students
Persons − (Staff ∪ Students)
```

If the partition constraint on F is satisfied, then each of the above collection expressions should evaluate to the empty collection.

Then the approach for the maintenance of classification structures can be outlined as follows. A semantic transaction is some sequence of operations that involves one or more update operations on collections. At the end of a semantic transaction, all exclusion terms involving one or more of the updated collections are used to check for semantic violations. If no violations are detected then the updates will take effect. Semantic transactions are atomic in effect and ensure the semantic integrity of the classification structure. However, they do not ensure the physical integrity of the database which is the concern of the Comandos transaction services.

The atomicity of semantic transactions is achieved through the use of differential representations of collections. A collection C is represented by a triple (C^0, C^+, C^-) where C^0 represents the extension of C at the beginning of a semantic transaction, C^+ represents the additions to C and C^- represents the deletions from C. On the basis of these differential representations, differential expressions can be derived which support the efficient computation of the differential representation associated with exclusion terms. If the differential representation of an exclusion term has a non-empty collection for the additions, then this indicates that the exclusion term does not evaluate to the empty collection and the associated constraint has been violated. Then all the updated collections can be restored to their values at the start of the semantic transaction. Further details of this mechanism for the detection of constraint violations is given in [BNHW93].

The development strategy for the ODMS was to first produce a prototype version on top of the object database management system Ontos [Ont91]. The prototype used the Ontos aggregates of set, list, bag and dictionary to realise the BROOM collections. This prototype provided a platform on which to develop database applications in terms of the BROOM model and thereby evaluate the semantic capabilities of the model and its support for application programming. The application systems which were developed on top of the prototype were a library system, a computer resources management system and a festival booking system. Details of this prototype are given in [Wal91].

At this stage, a prototype embedding of a query language based on the collection algebra was produced. A preprocessor written in Prolog was used to translate application programs written in C++/Ontos code with embedded collection expressions to C++/Ontos code. This work was undertaken in part by an MSc student and is reported in [Ong91].

The next stage in development was to implement the lower layers of the ODMS. Thus instead of relying on Ontos aggregates to realise the Aggregate and Bulk Layers, these were implemented directly on top of a persistent object store. Since the various Comandos kernels were still undergoing development, the Aggregate and Bulk Layers were implemented on top of the persistent store of the Exodus system using the persistent programming language E [CDG+89]. These layers were then ported on top of the Comandos kernel, Amadeus, developed at Trinity College, Dublin and the Comandos kernel, IK, developed at INESC, Lisbon [CBHdP93].

The Collection Layer realised in the Ontos prototype was then ported on top of the Aggregate and Bulk Layers and then enhanced with support for constraint management and active queries as outlined above.

As a result of these efforts, the ODMS provides a portable toolkit which has been implemented on a number of platforms and is such that it could be ported to other platforms based on versions of persistent C++ with relative ease.

Current work being undertaken on the ODMS includes the provision of a high-level data definition and storage language, DSDL, and a full embedding of a query language based on the collection model. The system provides a framework for query optimisation but a full investigation of optimisation strategies is still required. Various end-user database tools have been developed and these include a graphical BROOM schema editor and a General Forms Manager.

A further application that has been implemented on top of the ODMS is that of the Comandos Type Manager. The canonical type model of the Type Manager was specified in terms of the BROOM model [Cam91]. Based on this, a version of the Type Manager has now been implemented using the ODMS.

Chapter 8

Conclusions

This thesis addresses the general issue of how to provide data management services in object-oriented systems. To this end, a general object data model, called the collection model, is proposed and this provides a suitable foundation for the construction of such services. The main contributions of this data model stem from its generality and its semantic modelling capabilities.

The first section of this chapter summarises the contributions of the research reported in this thesis. The contributions lie not only in the details of the proposed collection model, but also in the general approach to the process of data model design and the realisation of object models in object-oriented systems.

The chapter concludes with a discussion of current and future research activities based on the collection model and its realisation in the context of the Comandos environment [CBHdP93].

8.1 Contributions

The contributions of this research are best described under the headings of statements of policy that are considered fundamental to the design of object data models to support data management services. Indeed most of these statements apply universally to the design of data models intended to support database systems.

A very general policy statement is given initially and this is followed by consideration of particular aspects of database system requirements.

- *Support Data Modelling in Object-Oriented Database Systems.*

Most existing object-oriented database management systems appear to have tackled the problem of providing support for database activities in a bottom-up manner; they extend existing object-oriented technologies with facilities for data management. Mechanisms are added to support persistence, transactions and efficient access to collections of objects. However, it is rare that these systems support the notion of a database as a complex structure of interrelated collections of objects. Even in the cases where this notion of a database is supported, it appears to be added as something of an afterthought. Certain restricted forms of conceptual dependencies may be supported, but the precise forms of these and the associated semantics are often unclear.

The problem should be tackled top-down rather than bottom-up. It is vital that one begins with a specification of data modelling requirements and then considers the general constructs required to support these requirements. From this, a general object data model can be designed which can be supported on a number of implementation platforms. For a given implementation platform, it may be necessary to impose certain restrictions to take account of the limitations of the platform. But the data model designer should not set out with these limitations in mind.

The top-down approach was adopted in the design of a data model for the Comandos ODMS. Out of this arose the proposed collection model. Whatever the arguments for or against certain features of the model, it is important to appreciate the desirability of the fact that it is a general model. Further, it is a model that is true to data modelling philosophy of the past two decades: it provides rich semantic constructs and supports physical data independence.

- *Support the Classification of Entities and their Representations.*

There is confusion in many systems about the use of typing in relation to classification. As discussed in Chapter 2, the significant distinction is between modelling the classification of entities that arise in the application reality and the classification of representation values in the database. Two entities may have common forms of representation in the database but they may be classified differently in the application reality. The classification of representation values is according to a set of necessary and sufficient conditions that specify the form of the representation. However, modelling the classification of entities in the real world is more difficult in that it is not always possible, or practical, to define membership conditions for a particular classification.

It is important to recognise that these two forms of classification sit side by side in a database system. The user or database designer should not be presented with a

model in which these two issues are confused.

The collection model has a clear separation of the two forms of classification. The classification of representation values is achieved by the type model. A type specifies a set of necessary and sufficient conditions that define the membership of that type. The classification of entities is represented by object groupings known as collections. The former activity is referred to as typing and the latter as classification.

This separation of typing and classification allows the collection model to be divorced from the underlying type system. The collection model is concerned with the management of object references rather than with the objects themselves. This view dispenses with the problems encountered in many object-oriented database systems where objects reside in collections and it is therefore not possible to have an object located in more than one collection. Further problems arise in such systems with the representation of objects in collections that have more than one parent collection. The problem of object representation should reside at the object level and be the concern of the type model and the persistent store.

- *Provide Rich Semantic Modelling Constructs.*

Over the past two decades, significant advances have been made in data modelling research in terms of the semantic modelling capabilities of data models. Much of this research has drawn on work in the areas of cognitive modelling and knowledge representation.

In addition, the use of these models has spread. The entity-relationship data model [Che76], and its variants, have proved enormously successful and are widely-used not only in database communities but also in software engineering communities.

Even the relational systems have taken note of the importance of providing good semantic modelling support and extensions both to the relational model itself [Cod79] and to relational systems [SK91] have been proposed to incorporate semantic modelling constructs.

It is a great pity that with respect to semantic modelling, many of the object-oriented database systems appear to have neglected these advances and have taken a step backwards. In general, they rely on the object-oriented programming concepts of objects with behaviour and subtyping to provide the semantic modelling capabilities of their models.

The collection model combines features of the entity-relationship models and the semantic data models which adopt an entity-based approach. It is similar to the entity-relationship models in that it provides direct support for the representation of

both entity categories and relationships. It also supports rich classification structures as advocated in many semantic data models and knowledge representation languages.

- *Produce Clear and Complete Specifications of Data Models.*

The development of a database application system should be based around a conceptual model of the corresponding application reality. The first stage is therefore to construct a description of that reality in terms of the supported data model. Because of the in-built nature of the data model in many existing object-oriented database systems, it is difficult to isolate the data model component for use in this first stage of system development.

It is important that a clear and complete specification of the data model be given. This specification must meet the needs of both the users and implementors of the model. A clear informal overview, with examples to demonstrate its use, should be provided to satisfy the needs of those who wish to learn about the main features of the model and to use it. A complete formal specification is required for the implementors of the model and for any clarification of model semantics. Thus, the description of the model should be neither purely informal nor purely formal. This point is stressed because many presentations of object data models tend to satisfy one or other need but rarely satisfy both. Either there is a lack of a precise description of the structure and semantics, or, the description is mainly formal and gives little insight into the motivation behind the model and examples of its use.

Further, an intermediate form of specification is useful both as a means of documentation and also as an aid to the production of the formal specification. For the collection model, a meta-circular description was employed for this intermediate stage of specification. It proved very useful in clarifying design decisions, testing the modelling capabilities of the model and also in the production of the formal specification. It is also useful in the development of tools to support the model and in developing systems with a uniform treatment of metadata and data.

- *Support High-Level Query Languages.*

A database management system should support one or more high-level query languages. For example, a system might support a query language embedding in C++, a report generation language and a graphical query language. A three level language structure can be employed to support both multiple high-level query languages and also multiple implementation platforms. A high-level language is translated into a target language which, in turn, is translated into a methods language. The target

language provides operations on the logical constructs of the data model. A particular realisation of the data model provides a methods language which operates on the physical structures of the database.

The target language of the collection model is provided by the collection algebra. This algebra could support a number of high-level query languages and, as reported in the next section, this is a topic of current research within the Comandos project.

Optimisation can be performed at all three language levels but often the most significant optimisations can be made at the higher levels where the the optimisation techniques are better able to utilise semantic information of the application domain.

In object-oriented database systems which omit support for the notion of a database structure, the attention to efficiency addresses access to individual objects or single collections of objects. The collection model supports database structures involving multiple collections of objects, and operations on these collections. Then in the realisations of the collection model, optimisations can be made at a higher-level which means that they tend to be more global and less localised.

- *Support Database System Evolution*

A database evolves in the course of time. In chapter 6, the three basic forms of evolution were examined. These are: the creation and deletion of entity representations; the evolution of objects required to represent the role changes that entities undergo in their life cycle; and the evolution of the structure and operation of the database system in line with changes to the application system itself.

It is important that all three forms of evolution be supported in database systems; it should not be left to the application programmer to cope with problems of change. There are two main aspects of evolution: firstly, supporting the necessary changes, and, secondly, controlling the forms of change allowed. The discussion in chapter 6 indicates the various forms of support required for evolution. In particular, support for object evolution is discussed in detail. A simple extension to the collection model to control object migration within a classification structure is proposed.

8.2 Further Work

As discussed in Chapter 7, the collection model has been realised in the Comandos project in terms of the Comandos Object Data Management Services (ODMS) [CBHdP93]. One of the main achievements of the ODMS work is the production of

an excellent platform for further research and development activities. This is mainly a consequence of the layered, open architecture which facilitates the development, or even replacement, of individual layers.

The collection model, and its realisation in terms of the Comandos ODMS, is seen as a beginning rather than an end. It opens the way for further interesting research work of countless varieties. There are three general directions of research. These are: supporting users of the model, providing additional and enhanced realisations of the model, and, extending the model. Here, overviews are given of three research topics currently under investigation within the Comandos project.

Database Tools

Database tools are required to support a range of users and user activities. Most of the existing graphical database tools support either schema input and display, or end-user activities of browsing and querying. Two important areas often neglected are support for the actual design process and support for the application programmer in the construction of interfaces for particular database application systems. A further activity for which there seems to be little support is that of the documenter who wishes to produce schema diagrams for inclusion in reports and papers.

An examination of the various database tools that have been proposed reveals two main requirements for the visualisation of data. The first of these concerns the display of schema information that represents the overall structure of the database in terms of collections of values and their interrelationships. Such information is generally represented as a graph with one or more node types and one or more arc types. The second form of visualisation of data concerns the display of data values. These values may have an arbitrarily complex structure and further may not have a direct representation in the database but are rather constructed from a number of stored values.

Within the Comandos ODMS group, two major graphical utilities have been designed and developed and these can form the basis for tools to support a range of user activities. The utilities are a graphical schema editor and a general interactive forms utility.

The graphical schema editor supports the input and display of database schemas. In part, this is achieved by building the schema editor around a generalised graph editor which deals with arbitrary forms of nodes and arcs. Then for a particular data model, the specific forms of nodes and arcs are specified along with the constraints on these that specify the valid schema structures of the model.

The general interactive forms utility deals with the display of complex values. It can be used both in read and write modes and can therefore also support data input and

database updates. The utility can be accessed in two ways. It can be used directly as a tool for the interactive design of forms and also as a library of classes to support the use of forms from an application program. Note that the interactive design of forms is useful not only for the construction of database application specific interfaces but also for the construction of interfaces for database tools. Indeed, the interface to the interactive forms manager was constructed using itself. An important point is that the general interactive forms utility is independent of the data model of a particular system.

A database design tool for BROOM has been constructed using these two utilities. The design tool supports the input of schema information through a graphical schema editor and forms interface. A schema, or part of a schema, can be selected and a corresponding Postscript file generated. Then BROOM schema diagrams can be constructed, edited with the tool and then included in documents by means of the generated Postscript file.

As a consequence of the increase in personal computers and the decentralisation of computing resources, an increasing number of non-experts are faced with the task of database design. It is important that tools should be developed to support such users in their design activities. The BROOM database design tool could be extended to support the actual design process by means of interactive dialogue sessions. A prototype of such a tool, Interactive Database Designer System (IDDS), is described in [Nor93]. The system guides the user through the design activity by eliciting information through question-answer sessions along with menu selection and form editing. In particular, the system generates classification structures by asking the user various questions about the members of a collection at the time that the collection is created. The system also encourages reusability by allowing the users to base their designs on one or more existing database designs.

Other proposals for tools to be constructed using the schema editor and general interactive forms utility are a browser and a graphical querying system. Details of the utilities and the database design tool for BROOM is given in the Comandos ODMS working paper [DKNH92].

Query Systems

The current status of the ODMS might be considered as providing a framework for query systems. The mechanisms are there for the representation, evaluation and optimisation of queries. These mechanisms are general in that they will support both end-user ad-hoc querying and queries embedded in application programs. The approach and framework has been proved through the support of prototype query languages and their embeddings.

Having established this framework there are three main areas for further research:

Query Languages The collection algebra provides an appropriate target language for high-level query languages. The design of suitable query languages is under investigation by the Comandos ODMS group. In particular, there is a proposal for a query language EQUAL, and work on the implementation of an embedding of this language in C++ is in progress. The ODMS group is also considering fourth generation languages and graphical query languages.

Query Processing The framework supports both the interpretation and compilation approaches to query processing. There are plans for further investigations into the relative merits of the two approaches.

Query Optimisation A query is represented as an active object which is self-optimising. The mechanisms are in place for query optimisation but, at present, only very simple optimisations take place. A complete investigation of optimisation strategies is required. These optimisations can be based both on the algebraic properties of the collection algebra and on the set of available physical representations of collections.

Constraint Systems

The BROOM model has a fixed number of structural constraints that represent conceptual dependencies among collections. These take two general forms. Firstly, there is the collection family construct with its subcollection, disjoint, cover, partition and intersection constraints. Secondly, there are constraints associated with relations that specify a source and target collection for binary collections and the associated cardinality constraints.

In chapter 7, a brief description of a general mechanism for checking violations of these constraints under update was given. This mechanism is based on a differential representation of collections in terms of an initial collection of values together with a collection of insertions and a collection of deletions. Details of this mechanism are given in [BNHW93]. The Comandos ODMS therefore has a mechanism for checking for constraint violation and taking a recovery action by undoing all the updates and restoring the collection to its state at the start of the update sequence that forms part of a logically atomic update operation.

The question arises of whether recovery from violation could be handled in some way other than undoing updates. The system could use the dependencies expressed by the constraints to attempt to restore consistency by the propagation of updates. For example, if an object is inserted in a collection and it is not already a member of that collection's supercollection, then it could be inserted automatically into that supercollection. Then programming in terms of update operations becomes a form of "programming by constraints". Effectively, this means that the user or application programmer is presented with a higher-level of update operation since a single

user-specified operation can generate a number of updates. The problem with such a system is that the effects of update propagation could be unexpected and quite drastic. A mistake by the user could result in the deletion of large sections of the database - and not simply in the deletion of the wrong object. Therefore, there should be some form of control on propagations. A confirmation dialogue could take place with the user in interactive systems, but, the case of non-interactive systems proves more difficult. This whole area requires further investigation.

An alternative approach to the problem of constraint violation is to aim for avoidance rather than detection and recovery. A transaction designer tool can be used to check for possible constraint violations. The application programmer is informed of potential violations and they must then add the appropriate actions to ensure consistency at the end of the transaction. Work in this area has been done by Stemple and Sheard [SS88]. The approach is currently under investigation by members of the Comandos ODMS group for use in the context of BROOM and the ODMS.

There are a number of other issues in connection with constraints that are ripe for investigation in the context of the collection model and, in particular, the ODMS. These include the use of constraint information in the design process. Based on constraints it is possible to transform BROOM schemas into equivalent schemas. This raises the issue of schema equivalence and the question as to what measures can be used to judge that one schema is preferred over another. It would also be interesting to design a general constraint model for the ODMS which would be layered on top of the collection model as indicated in section 2.2.

Bibliography

[AB84] S. Abiteboul and N. Bidoit. Non First Normal Form Relations to Represent Hierarchically Organized Data. In *Proceedings ACM SIGACT/SIGMOD Symposium on Principles of Database Systems*, pages 191–200. ACM, 1984.

[AB88] S. Abiteboul and C. Beeri. On the Power of Languages for the Manipulation of Complex Objects. Technical Report 846, INRIA, Paris, 1988.

[ABC+83] M. Atkinson, P. Bailey, K. Chisholm, P. Cockshott, and R. Morrison. An Approach to Persistent Programming. *The Computer Journal*, 26(4), 1983.

[Abr74] J. R. Abrial. Data Semantics. In J. W. Klimbie and K. L. Koffeman, editors, *Data Base Management*, pages 1–59. North-Holland, 1974.

[ACC81] M. P. Atkinson, K. J. Chisholm, and W. P. Cockshott. PS-algol: an Algol with a Persistent Heap. *ACM SIGPLAN Notices*, 17(7), July 1981.

[ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: A Strongly-Typed, Interactive Conceptual Language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.

[AG89] R. Agrawal and N. Gehani. ODE: The Language and the Data Model. In *Proceedings ACM SIGMOD Intl. Conf. on Management of Data*, 1989.

[AGO91] A. Albano, G. Ghelli, and R. Orsini. A Relationship Mechanism for a Strongly Typed Object-Oriented Database Programming Language. In *Proceedings of Very Large Database Conference*, 1991.

[AH87] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12(4):525–565, 1987.

[And91] J. Andersen. Operations on sets in an Object-Oriented Database. Master's thesis, Institute of Informatics, University of Oslo, 1991.

[Arm89] D. M. Armstrong. *Universals: An Opinionated Introduction.* Westview Press, 1989.

[Atk90] R. Atkins. *Artspeak: a guide to contemporary Ideas, Movements, and Buzzwords.* Abbeville Press, 1990.

[BCD89] F. Bancilhon, S. Cluet, and C. Delobel. A Query Language for the O_2 Object-Oriented Database System. In R. Hull, R. Morrison, and D. Stemple, editors, *Proceedings of 2nd Intl. Workshop on Database Programming Languages.* Morgan Kaufmann, 1989.

[BCG+87] J. Banerjee, H. T. Chou, J. F. Garza, W. Kim, D. Woelk, N. Ballou, and H. J. Kim. Data Model Issues for Object-oriented Applications. *ACM Transactions on Office Information Systems*, 5(1), 1987.

[BCKK87] J. Banerjee, H.-T. Chou, H. J. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD Record*, 16(3):311–322, 1987.

[Bec88] W. Bechtel. *Philosophy of Mind: An Overview for Cognitive Science.* Lawrence Erlbaum Associates, 1988.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[BHN92] S. M. Blott, D. J. Harper, and M. C. Norrie. Active Queries - a Lazy Approach to Query Evaluation in OODBs. In *Proceedings of the First International Conference on Data and Knowledge Management*, Baltimore, Maryland, USA, November 1992.

[BK91] P. Barclay and J. Kennedy. Regaining the Conceptual Level in Object Oriented Data Modelling. In M. S. Jackson and A. E. Robinson, editors, *Aspects of Database Systems*, pages 269–305. Butterworth-Heinemann, 1991.

[BM89] O. Boucelma and J. L. Maitre. Querying complex-object databases: the LIFOO functional language. Technical report, Universite de Marseille, France, 1989.

[BMW84] A. Borgida, J. Mylopolous, and H. K. T. Wong. Generalization/Specialization as a Basis for Software Specification. In M. L. Brodie, J. Mylopoulos, and J. W. Schmidt, editors, *On Conceptual Modelling.* Springer-Verlag, 1984.

[BNHW93] S. M. Blott, M. C. Norrie, D. J. Harper, and A. D. M. Walker. Detecting semantic violations in generalised classification structures. In *Proceedings of the First International Conference on Intelligent and Cooperative Information Systems, Rotterdam.* IEEE Press, May 1993.

[BOS91] P. Butterworth, A. Otis, and J. Stein. The GemStone Object Database Management System. *Communications ACM*, 34(10):64–77, October 1991.

[Bra90] S. E. Bratsberg. FOOD: Supporting explicit relations in a Fully Object-Oriented Database. In *Proceedings IFIP TC2 Conference on Object Oriented Databases, Windermere, U.K.*, 1990.

[Bro81] M. L. Brodie. Association: A Database Abstraction for Semantic Modelling. In *Proceedings 2nd Intl. Entity-Relationship Conference*, October 1981.

[Cam91] J. Campin. Using the Comandos Data Model for Type Management. In D. J. Harper and M. C. Norrie, editors, *The Glasgow Collection of Comandos Papers*. Dept of Computing Science, University of Glasgow, Research Report, CSC/91/R16, 1991.

[Cat91a] R. G. G. Cattell. Next-Generation Database Systems. *Communications ACM*, 34(10), October 1991.

[Cat91b] R. G. G. Cattell. *Object Data Management: Object-oriented and Extended Relational Database Systems.* Addison-Wesley, 1991.

[CBHdP93] V. J. Cahill, R. Balter, N. Harris, and X. Rousset de Pina, editors. *The Comandos Distributed Application Platform.* Springer-Verlag, 1993.

[CDG+89] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. Vandenburg. The EXODUS Extensible DBMS Project: An Overview. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems.* Morgan-Kaufmann, 1989.

[CDLR90] S. Cluet, C. Delobel, C. Lecluse, and P. Richard. RELOOP, an Algebra Based Query Language for an Object-Oriented Database System. *Data and Knowledge Engineering*, 5:333–352, 1990.

[Che76] P. P. Chen. The Entity-Relationship Model - Towards a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[COD71] *CODASYL Database Task Group Report*, 1971.

[Cod79] E. F. Codd. Extending the Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems*, 4(4):397–434, 1979.

[CP84] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems.* McGraw-Hill, 1984.

[Deu91] O. Deux. The O_2 System. *Communications of the ACM*, 34(10):34–48, October 1991.

[DG90] O. Diaz and P. M. D. Gray. Semantic-rich User-defined Relationship as a Main Constructor in Object Oriented Database. In *Proceeding IFIP TC2 Conference on Object Oriented Databases, Windermere, U.K.*, 1990.

[Dil90] A. Diller. *Z : An Introduction to Formal Methods.* Wiley, 1990.

[DKNH92] C. Dunlop, A. Kosmynin, M. C. Norrie, and D. J. Harper. A Generic Toolkit for the Construction of Graphical Database Tools, 1992. Comandos ODMS Working Paper.

[EWH85] R. Elmasri, J. Weeldreyer, and A. Hevner. The Category Concept: An Extension to the Entity-Relationship Model. *Data and Knowledge Engineering*, 1(1), May 1985.

[FBC+87] D. H. Fishman, D. Beech, H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derret, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. C. Shan. Iris: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems*, 5(1):48–69, January 1987.

[Ghe90] G. Ghelli. A Class Abstraction for a Hierarchical Type System. In *Proceedings of 2nd Intl. Conf. on Database Theory, ICDT'90*, LNCS 470, pages 56–70. Springer Verlag, December 1990.

[GR85] A. Goldberg and D. Robson. *SMALLTALK-80: The Language and its Implementation.* Addison Wesley, 1985.

[Gra81] P. M. D. Gray. The GROUP_BY Operation in Relational Algebra. In *Proceedings BNCOD-1*, pages 84–98. Pentech Press, 1981.

[HK87] R. Hull and R. King. Semantic Data Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 19(3):201–260, 1987.

[HM81] M. Hammer and D. McLeod. Database Description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.

[HN91a] D. J. Harper and M. C. Norrie. Data Management for Object-Oriented Systems. In M. S. Jackson and A. E. Robinson, editors, *Aspects of Database Systems*, pages 69–92. Butterworth-Heinemann, 1991.

[HN91b] D. J. Harper and M. C. Norrie. The Glasgow Collection of Comandos Papers. Technical Report CSC/91/R16, University of Glasgow, Dept of Computing Science, University of Glasgow, Scotland, September 1991.

[JS82] G. Jaeschke and H.-J. Schek. Remarks on the algebra of non-first-normal-form relations. In *Proceedings ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, pages 124–138, 1982.

[KBC+89] W. Kim, N. Ballou, H. T. Chou, J. F. Garza, and D. Woelk. Features of the ORION Object-Oriented Database System. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications.* ACM Press, 1989.

[Kri77] S. Kripke. Identity and Necessity. In P. Schwarz, editor, *Naming, Necessity, and Natural Kinds.* Cornell University Press, 1977.

[Lak87] G. Lakoff. *Women, fire, and dangerous things. What categories reveal about the mind.* University of Chicago Press, 1987.

[Lel88] W. Leler. *Constraint Programming Languages.* Addison-Wesley, 1988.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, October 1991.

[LLPS91] G. M. Lohman, B. Lindsay, H. Pirahesh, and K. B. Schiefer. Extensions to Starburst: Objects, Types, Functions, and Rules. *Communications of the ACM*, 34(10):94–109, October 1991.

[LP91] M. Levene and A. Poulovassilis. An object-oriented data model formalised through hypergraphs. *Data and Knowledge Engineering*, 6:205–224, 1991.

[LPA91] Logic Programming Associates Ltd. *MacPROLOG 4.0 Reference Manual*, 1991.

[LRV88] C. Lecluse, P. Richard, and F. Velez. O_2, an Object-Oriented Data Model. In *Proceedings SIGMOD*, 1988.

[LS76] B. W. Lampson and H. E. Sturgis. Reflections on Operating System Design. *Communications of the ACM*, 19(5):251–265, May 1976.

[LS92] C. Laasch and M. H. Scholl. Generic Update Operations Keeping Object-Oriented Databases Consistent. In *Proceedings 2nd GI-Workshop on Information Systems and AI*, February 1992.

[MBCD89] R. Morrison, A. Brown, R. Connor, and A. Dearle. Napier88 Reference Manual. Technical Report PPRR-77-89, Universities of Glasgow and St. Andrews, 1989.

[MBW80] J. Mylopoulos, P. A. Bernstein, and H. K. T. Wong. A Language Facility for Designing Interactive Database-Intensive Systems. *ACM Transactions on Database Systems*, 5(2):185–207, June 1980.

[McG77] W. C. McGee. The information management system IMS/VS. *IBM System Journal*, 16, 1977.

[MD91] F. Manola and U. Dayal. An Overview of PDM: An Object-Oriented Data Model. In K. R. Dittrich, U. Dayal, and A. P. Buchmann, editors, *On Object-Oriented Database Systems.* Springer-Verlag, 1991.

[Nor93] M. C. Norrie. An Interactive System for Object-Oriented Design. In R. L. Cooper, editor, *Interfaces to Databases*, Workshops in Computing. Springer-Verlag, 1993.

[NQZ90] R. Nassif, Y. Qiu, and J. Zhu. Extending the Object-Oriented Paradigm to Support Relationships and Constraints. In *Proceedings IFIP TC2 Conference on Object Oriented Databases, Windermere, U.K.*, 1990.

[Ong91] S. H. Ong. Development of an AQL Preprocessor. Master's thesis, Dept of Computing Science, University of Glasgow, 1991.

[Ont91] ONTOS, Inc. *ONTOS Reference Manual*, 1991.

[PL93] A. Poulovassilis and M. Levene. A Nested-Graph Model for the Representation and Manipulation of Complex Objects. *ACM Transactions on Information Systems*, to appear, 1993.

[PM88] J. Peckham and F. Maryanski. Semantic Data Models. *ACM Computing Surveys*, 20(3):153–189, September 1988.

[PS87] D. J. Penney and J. Stein. Class Modification in the GemStone Object-Oriented DBMS. In *Proceedings OOPSLA '87*, pages 111–117, 1987.

[PST91] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z.* Prentice Hall, 1991.

[PT86] P. Pistor and Traunmuller. A database language for sets, lists, and tables. *Information Systems*, 11(4):323–336, December 1986.

[Put77a] H. Putnam. Is Semantics Possible? In P. Schwarz, editor, *Naming, Necessity, and Natural Kinds.* Cornell University Press, 1977.

[Put77b] H. Putnam. Meaning and Reference. In P. Schwarz, editor, *Naming, Necessity, and Natural Kinds.* Cornell University Press, 1977.

[Qui68] R. Quillian. Semantic Memory. In M. Minsky, editor, *Semantic Information Processing.* MIT Press, 1968.

[Qui77] W. V. Quine. Natural Kinds. In P. Schwarz, editor, *Naming, Necessity, and Natural Kinds.* Cornell University Press, 1977.

[Rap68] B. Raphael. A computer program for semantic information retrieval. In M. Minsky, editor, *Semantic Information Processing.* MIT Press, 1968.

[RKB87] M. A. Roth, H. F. Korth, and D. S. Batory. SQL/NF : A Query Language for $\neg$ 1NF Relational Databases. *Information Systems*, 12(1):99–114, March 1987.

[RKS88] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended Algebra and Calculus for Nested Relational Databases. *ACM Transactions on Database Systems*, 13(4):389–417, December 1988.

[Rod92] J. F. Roddick. Schema Evolution in Database Systems - An Annotated Bibliography. Technical Report CIS-92-004, University of South Australia, School of Computer and Information Science, University of South Australia, SA 5095, 1992.

[Ros75] E. Rosch. Cognitive representations of semantic categories. *Journal of Experimental Psychology: General*, 104:192–233, 1975.

[RS87] L. A. Rowe and M. R. Stonebraker. The POSTGRES Data Model. In *Proceedings VLDB'87*, pages 83–96, 1987.

[Rum87] J. Rumbaugh. Relations as Semantic Constructs in an Object-Oriented Language. In *Proceedings OOPSLA*, pages 466–481, 1987.

[Rus56] B. Russell. Mathematical Logic as Based on The Theory of Types. In R. C. Marsh, editor, *Bertrand Russell: Logic and Knowledge.* George Allen and Unwin, 1956.

[Ryl49] G. Ryle. *The Concept of Mind.* Barnes and Noble Books, 1949.

[SFG+91] N. A. Stillings, M. H. Feinstein, J. L. Garfield, E. L. Rissland, D. A. Rosenbaum, S. E. Weisler, and L. Baker-Ward. *Cognitive Science: An Introduction.* MIT Press, 1991.

[Shi81] D. W. Shipman. The Functional Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, 1981.

[SK91] M. Stonebraker and G. Kemnitz. The POSTGRES Next Generation Database Management System. *Communications ACM*, 34(10):78–92, October 1991.

[SKME89] R. Schumann, R. Kroeger, M. Mock, and E.Nett. Recovery Management in the RelaX Distributed Transaction Layer. In *Proceedings 8th Symposium on Reliable Distributed Systems*, Seattle, USA, October 1989.

[SLR+92] M. H. Scholl, C. Laasch, C. Rich, H. J. Schek, and M. Tresch. The COCOON object model. Technical report, Department of Computer Science, ETH Zurich, 1992.

[Som67] F. Sommers. Types and Ontology. In P. F. Strawson, editor, *Philosophical Logic.* Oxford University Press, 1967.

[Sow84] J. F. Sowa. *Conceptual Structures : Information Processing in Mind and Machine.* Addison-Wesley, 1984.

[Spi89] J. M. Spivey. *The Z Notation.* Prentice Hall, 1989.

[SS77] J. M. Smith and D. C. Smith. Database abstractions: Aggregation and generalization. *ACM Transactions on Database Systems*, 2(2):105–133, 1977.

[SS86] H. J. Schek and M. H. Scholl. The Relational Model with Relation-Valued Attributes. *Information Systems*, 11(2):137–147, June 1986.

[SS88] T. Sheard and D. Stemple. Automatic Verification of Database Transaction Safety. Technical Report COINS 88-29, Dept of Computer and Information Science, University of Massachusetts at Amherst, 1988.

[SS90] M. H. Scholl and H. J. Schek. A relational object model. In S. Abiteboul and P. C. Kanellakis, editors, *Proceedings Intl. Conf. on Database Theory, ICDT'90*, LNCS 470, pages 89–105. Springer-Verlag, 1990.

[SS91] H.-J. Schek and M. H. Scholl. From Relations and Nested Relations to Object Models. In M. S. Jackson and A. E. Robinson, editors, *Aspects of Database Systems*, pages 202–225. Butterworth-Heinemann, 1991.

[SST92] M. H. Scholl, H.-J. Schek, and M. Tresch. Object Algebra and Views for Multi-Objectbases. In *Proceedings Intl. Workshop on Distributed Object Management*, pages 202–225, 1992.

[SSW80] P. Scheuermann, G. Schiffner, and H. Weber. Abstraction Capabilities and Invariant Properties Modeling within the Entity-Relationship Approach. In P. Chen, editor, *Entity-Relationship Approach to Systems Analysis and Design*. North-Holland, 1980.

[Sta72] H. Staniland. *Universals*. Macmillan Press, 1972.

[Str87] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1987.

[Str90] D. D. Straube. *Queries and Query Processing in Object-Oriented Database Systems*. PhD thesis, University of Alberta, Edmonton, Alberta, Canada, December 1990.

[SWK76] M. Stonebraker, E. Wong, and P. Kreps. The Design and Implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, 1976.

[SZ86] A. H. Skarra and S. B. Zdonik. The Management of Changing Types in an Object-Oriented Database. In *Proceedings OOPSLA'86*, pages 483–495, 1986.

[SZ87] A.H. Skarra and S. B. Zdonik. Type evolution in an object-oriented database. In B. Shriver, editor, *Research directions in object-oriented programming*, pages 393–416. MIT Press, 1987.

[SZ89] G. M. Shaw and S. B. Zdonik. An Object-Oriented Query Algebra. In *Proceedings of the 2nd Workshop on Database Programming Languages.* Morgan Kaufmann, June 1989.

[SZ90a] G. M. Shaw and S. B. Zdonik. A Query Algebra for Object-Oriented Databases. In *Proceedings of the Sixth Intl Conf on Data Engineering,* February 1990.

[SZ90b] G. M. Shaw and S. B. Zdonik. Object-Oriented Queries: Equivalence and Optimization. In J.-M. Nicolas W. Kim and S. Nishio, editors, *Proceedings of Deductive and Object-Oriented Databases*, pages 281–295. Elsevier Science Publishers B.V. (North-Holland), 1990.

[TF76] R. W. Taylor and R. L. Frank. CODASYL data-base management systems. *ACM Computing Surveys*, 8, 1976.

[TL82] D. Tsichritzis and F. H. Lochovsky. *Data Models.* Prentice-Hall, 1982.

[TYF86] T. Teorey, D. Yang, and J. Fry. A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model. *ACM Computing Surveys*, 18(2), June 1986.

[VD90a] S. Vandenberg and D. DeWitt. Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance. Technical TR 987, University of Wisconsin, Computer Sciences Department, University of Wisconcin, Madison, Wisconsin, 1990.

[VD90b] S. L. Vandenberg and C. J. DeWitt. An Algebra for Complex Objects with Arrays and Identity. Technical Report TR 918, University of Wisconsin, Computer Sciences Department, University of Wisconsin, Madison, WI 53706, March 1990.

[VD91] S. L. Vandenberg and D. J. DeWitt. Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance. In *Proceedings of SIGMOD 91*, 1991.

[VH91] V. Ventrone and S. Heiler. Semantic Heterogeneity. *SIGMOD RECORD*, 20(4), December 1991.

[Wal91] A. Walker. The Object Data Management Prototype. In D. J. Harper and M. C. Norrie, editors, *The Glasgow Collection of Comandos Papers.* Dept of Computing Science, University of Glasgow, Research Report CSC/91/R16, 1991.

[Wit53] L. Wittgenstein. *Philosophical Investigations.* MacMillan, 1953.

[WT90] D. A. Watt and P. Trinder. Towards a Theory of Bulk Data Types. Technical Report FIDE/91/26, University of Glasgow, FIDE Coordinator, Dept of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, U.K., 1990.

[Zam75] A. V. Zamulin. On the Database-Oriented Programming Language. *Programmirovanie (in Russian)*, (5):23–31, 1975.

[Zam78] A. V. Zamulin. BOYAZ - A Database-Oriented Programming Language. *Algoritmy i organizacija economicheskich zadach (in Russian)*, (12):40–67, 1978.

[Zam89] A. V. Zamulin. The Database Programming Language Atlant. Technical Report CSC/89/R13, University of Glasgow, Dept of Computing Science, Glasgow G12 8QQ, Scotland, U.K., June 1989.

[Zdo87] S. B. Zdonik. Can Objects Change Type? Can Type Objects Change? In *Proceedings Workshop on Database Programming Languages, Altair, France*, 1987.

[ZM90] S. B. Zdonik and D. Maier, editors. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann, 1990.

GLASGOW
UNIVERSITY