A Configurable Data Modelling System

Qin Zhenzhou

A thesis submitted to the Faculty of Science,

University of Glasgow

for the degree of Doctor of Philosophy

December 1995

Abstract

User interaction facilities are usually the weakest component of database management systems (DBMS). They are typically few in number and poor in quality as compared both to other features of DBMS and to user interaction facilities for other kinds of software. One cause for this is that adding further mechanisms to a DBMS requires tedious and repetitive programming effort in the context of a complex system.

The Configurable Data Modelling System (CDMS) attempts to get round this problem by providing an environment in which interaction facilities can be built more straightforwardly.

The CDMS considers a user interaction facility to be the pairing of a conceptual data model with a concrete user interface. The CDMS provides a generic data model, comprising elements for describing data structures, constraints and behaviour, together with one menu driven system for creating conceptual models as instances of the generic model and another for creating user interfaces to each data model thus generated.

This thesis describes the important features of the system. The thesis discusses the main difficulty in creating such a system; that is, obtaining a consistent and coherent analysis of all of the components which might be housed in a DBMS.

Apart from a theoretical analysis of the relevant issues, the research presented in this thesis has also established a prototype to test the theory. The research has been undertaken in a persistent programming environment. Persistent language technology has enabled the construction of a sophisticated and well-integrated CDMS. At the same time, the research has enhanced persistent programming environments with models, methodologies and tools that are crucial to the exploitation of persistent programming in construction and maintenance of long-lived, data-intensive application systems.

Acknowledgements

First of all I am indebted to my supervisor Richard Cooper for his continuous guidance, encouragement, enthusiasm and source of ideas. His support, whatever the issue, is very much appreciated. I am also grateful to Malcolm Atkinson, Ray Welland and David Harper for their advice and support.

I would also like to acknowledge my colleagues in Glasgow, in particular Paul Philbrow, Phil Trinder, Moira Norrie, David Kerr, Daniel Chan, David England, Dag Sjoberg, João Lopes and Artur Serrano. The discussions with them have helped the understanding of the issues of this research. Some of the work was performed collaboratively with Lay Khim Tan, Yng Tai, Daphne Lee and Rowena Lum, whose collaboration I acknowledge with gratitude.

It is appreciated that the Napier88 system, provided by the St Andrews persistent programming team, led by Ron Morrison, is an excellent system, without which the implementation in this research would have been very difficult.

It is the financial support by COMANDOS II (Construction and Management of Distributed Open Systems, ESPRIT project 2071), FIDE (Fully Integrated Data Environment, ESPRIT project BRA6309), CDM (Configurable Data Modelling, SERC Research Grant H17671) and IOPT (Introduction of Process Technology, ICL project) that made this research possible.

My wife, He Yu, and my son, Qin Jian, have also shown a level of love and support which is exceptional and without which this thesis would not have come to be.

Contents

1 Introduction 1

1.1 Database and Database Access 1
1.2 Configurable Data Modelling 3
1.3 Research Procedure 7
1.4 Structure of Thesis 8

2 The Usability of Database Systems 10

2.1 Databases and Database Systems 10
2.1.1 What is a Database? 10
2.1.2 The Architecture of Conventional DBMS 12
2.1.3 The Activities Required 14
2.2 User Interface to DBMS 15
2.2.1 Styles of Interface 15
2.2.2 Database Programming Languages 16
2.2.3 Interface Independence 17
2.2.3.1 Dialogue Component and Computational Component 17
2.2.3.2 The Advantage of Interface Independence 18
2.2.3.3 Requirements Arising from Dialogue Independence 19
2.2.4 Configuring the User Interface 21
2.3 Data Modelling 22
2.3.1 What Is a Data Model? 22
2.3.2 An Overview of Data Models 24
2.3.3 The Object-Oriented Approach 32
2.3.4 The Components of Data Models 35
2.3.5 Configuring Data Modelling 35
2.4 Federated Databases 36
2.4.1 The Characteristics of Federated Database Systems 36
2.4.2 Data Sharing Approaches 37
2.4.3 Database Conversion 37

2.4.4 Schema Transformation and Transaction Translation 40
2.5 Conclusions 46

3 Configurable Data Modelling System 48

3.1 Motivation and Feasibility 48
3.2 Data Modelling Process and Methodology 49
3.2.1 A Database as an Instance of a Data Schema 49
3.2.2 A Data Schema as an Instance of a Data Model 52
3.2.3 The Global Model as Generalisation of Data Models 53
3.3 CDMS Structure 55
3.4 Modelling Primitives and Data Model Configuration 58
3.5 Dealing with Constraints in the CDMS 60
3.6 Dealing with Behaviour in the CDMS 62
3.7 Interaction Elements and Dialogue Primitives 62
3.8 CDMS Functionality 64
3.9 Related Work 65

4 Constraints in the CDMS 68

4.1 Some Basic Concepts of Constraints 70
4.1.1 The Role of Constraints in a DBMS 70
4.1.2 Constraints as Predicates 71
4.1.3 The Management of Constraints 71
4.1.4 Placement of Constraints 72
4.2 Constraints in the CDMS Architecture 73
4.2.1 Some Introductory Examples 74
4.2.2 Constraint Architecture in the CDMS 75
4.2.3 Further Examples 79
4.3 Varieties of Integrity Constraints 84
4.3.1 Schema-Data Constraints 85
4.3.2 Model-Data constraints 91
4.3.3 Global Data Constraints 95
4.3.4 Metadata Constraints 96
4.3.5 Meta-metadata Constraints 99
4.4 Constraint Configuration in the CDMS 101
4.4.1 Constraint Configuration 101
4.4.2 Constraint Specification 105

4.5 Constraint Management 108
4.5.1 Some Concepts of Constraint Management 108
4.5.2 Simple Constraint management 110
4.5.3 Constraint Management using Transactions 111
4.6 Summary 113

5 Behavioural Issues 115

5.1 Active Objects in Semantic Data Modelling 116
5.2 Transactions as Active Objects 119
5.2.1 Database Transactions 119
5.2.2 Transactions and Concurrency Control 119
5.2.3 Transactions in Relation to Constraints 120
5.2.4 Conclusion 121
5.3 Process Support System 121
5.3.1 Role-Activity Diagram 123
5.3.2 Process Management Language 126
5.3.3 Process Support Environment 128
5.3.4 Implementing a Process Model 130
5.3.5 Conclusions 131
5.4 Computation in Database Systems 131
5.5 Active Database Systems 134
5.6 Summary of Behavioural Issues 135
5.7 The Behavioural Aspect of CDMS 136
5.7.1 Categorisation of Active Objects 136
5.7.2 Examples of Configuration of Active Objects 138
5.7.3 Using the Configured Functionality 141
5.8 Conclusions 142

6 The Platform for Implementing the CDMS 143

6.1 Persistent Programming Systems 143
6.2 PS-algol 145
6.2.1 Overview 145
6.2.2 The PS-algol Type System 146
6.2.3 Using PS-algol 147
6.2.4 Discussion 148

6.3 Napier88 151
6.3.1 Overview 151
6.3.2 The Napier88 Type System 152
6.3.3 Use of Napier88 154
6.3.4 Discussion 154
6.4 Napier88 Library 155
6.4.1 The Napier88 Compiler 155
6.4.2 The Bulk Values 155
6.4.3 WIN 156
6.5 The Advantages of the Persistent Approach 158

7 The Design and Implementation 161

7.1 Implementation Overview 162
7.2 Program Architecture 163
7.3 The Global Model 168
7.4 The Instantiation Process 168
7.4.1 The Creation of a Data Model 168
7.4.2 Adding the User Interface 169
7.4.3 Creating a Schema and a Database 170
7.5 Storage Structure 170
7.6 The User Interface to the CDMS Program 173
7.7 Current Status 178
7.8 Discussion 179

8 The Application of the CDMS 180

8.1 Introduction 180
8.2 Meta-metadata Syntax 181
8.3 Constructing the Relational Model 185
8.4 Constructing the Entity-Relationship Model 189
8.5 Constructing the Functional Data Model 195
8.6 Constructing IFO 198
8.7 Conclusions 205

9 Conclusions 206

9.1 Contributions 207

9.2 Further Work 209

Bibliography 212

List of Figures

Figure 2.1 External Dialogue and Internal Dialogue 18
Figure 2 2 User Interface Configuration 21
Figure 2.3 A Schema in the Semantic Binary Data Model 25
Figure 2.4 A Schema in the Entity-Relationship Model 26
Figure 2.5 A Schema in the Functional Data Model 29
Figure 2.6 A Schema in IFO 31
Figure 2.7 Database Conversion Approach 38
Figure 2.8 Database Systems with Database Conversion Approach 39
Figure 2.9 Schema Transformation Approach 40
Figure 2.10 Single ML to Single ML Mapping 42
Figure 2.11 Multiple MLs to Single ML Mapping 43
Figure 2.12 Single ML to Multiple MLs Mapping 44
Figure 2.13 Multiple MLs to Multiple MLs Mapping 45

Figure 3.1 A Library Schema 50
Figure 3.2 A Library Database 51
Figure 3.3 A Simplified ER Model 53
Figure 3.4 The Global Data Model 54
Figure 3.5 Combination Ability of Connection Types 55
Figure 3.6 Overall Structure of the CDMS 56
Figure 3.7 Specialisation of Generic Data Modelling Primitives 59
Figure 3.8 Interface Definition 63

Figure 4.1 Constraint Structure in the CDMS 77
Figure 4.2 Constraint Specification 79
Figure 4.3 Schema Inherent Data Constraints 80
Figure 4.4 Model Implicit Data Constraints 81
Figure 4.5 Model Inherent Data Constraints 82
Figure 4.6 Model Inherent Metadata Constraints 83
Figure 4.7 Global Inherent Meta-metadata Constraints 84
Figure 4.8 Data Constraints in a Simple ER Schema 85

Figure 4.9 A Database Using the Simple ER Schema 85
Figure 4.10 An Example of Disjoint Specialisation 88
Figure 4.11 An Example of Covering Specialisation 89
Figure 4.12 An Example of Disjoint Covering Specialisation 89
Figure 4.13 An Example of Connection Classes Combination Cardinality 90
Figure 4.14 Connection Class Cardinality at Type 91
Figure 4.15 Generalisation of the IFO Model 93
Figure 4.16 Generalisation in an IFO Schema 94
Figure 4.17 Grouping of the IFO Model 94
Figure 4.18 Aggregating of the IFO Model 95
Figure 4.19 A Global Data Constraint 96
Figure 4.20 Model-Metadata Constraints 97
Figure 4.21 An Invalid Pattern in IFO Schema 99
Figure 4.22 A Global Meta-Metadata Constraint 100
Figure 4.23 Configuration of Data Constraints 102
Figure 4.24 Configuration of Metadata Constraints 104
Figure 4.25 Specification of Constraints in a Model 106
Figure 4.26 Specification of Constraints in a Schema 107
Figure 4.27 Constraint Suspension and Re-Imposition 111

Figure 5.1 A Sample RAD 125
Figure 5.2 RAD Notations 126
Figure 5.3 A Sample PML Program 128
Figure 5.4 An Action Agenda Window 130
Figure 5.5 A Dialogue Window 130
Figure 5.6 A Library Schema 133
Figure 5.7 A Library Database 133
Figure 5.8 Active Objects in the CDMS 137
Figure 5.9 Transaction 138
Figure 5.10 Thread, Role and Process 139
Figure 5.11 Model Manipulation Schema in the CDMS 140

Figure 6.1 Declarations in PS-algol 147
Figure 6.2 Declarations in Napier88 152

Figure 7.1 Program Architecture 164
Figure 7.2 Software Modules 166
Figure 7.3 Storage Structure 171

Figure 7.4 The CDMS Main Menu 174
Figure 7.5 A Schema Editing Window 176
Figure 7.6 A Schema Display Window 177
Figure 7.7 An Interface Editing Window 178

Figure 8.1 Constructs 182
Figure 8.2 Metadata Constraints 182
Figure 8.3 Data Constraints (Part I) 183
Figure 8.4 Data Constraints (Part II) 184
Figure 8.5 The Relational Model - Constructs 187
Figure 8.6 The Relational Model - Metadata Constraints 188
Figure 8.7 The Relational Model - Data Constraints 188
Figure 8.8 The Entity-Relationship Model - Constructs 191
Figure 8.9 The Entity-Relationship Model - Metadata Constraints 192
Figure 8.10 The Entity-Relationship Model - Data Constraints 193
Figure 8.11 Exclusive Alternative Relationship 194
Figure 8.12 The Functional Data Model - Constructs 196
Figure 8.13 The Functional Data Model - Metadata Constraints 197
Figure 8.14 The Functional Data Model - Data Constraints 197
Figure 8.15 IFO - Constructs 201
Figure 8.16 IFO - Metadata Constraints 202
Figure 8.17 IFO - Data Constraints (Part I) 203
Figure 8.18 IFO - Data Constraints (Part II) 204

Figure 9.1 Comparison between the CDMS and FDSs 208

1 Introduction

This thesis develops a theory of semantic data modeling, which is based on the **decomposability** and **reconfigurability** of semantic data models and human-computer interfaces. The theory takes **constraints** as special values and treats both data **structure** and **behaviour** consistently.

A prototype system is established as the theory is developed. The system enables its user to configure data models as well as interfaces, to specify data schemata, and to manipulate data, all in a way which keeps the model apparent.

1.1 Database and Database Access

Database technology grew out of an attempt to facilitate the development of data intensive applications. The technology factors out the common elements of such applications, so as to raise data processing quality and reduce redundant work in the implementation of the applications. Database technology has made a major impact on the growing use of computers.

A **Database Management System** (DBMS) provides centralised functionality for storing, retrieving and updating data in various ways. The architecture of a DBMS can be discussed in a three-layer framework, which consists of a physical layer, a conceptual layer and an external layer. The **physical layer** deals with the details of data storage; the **conceptual layer** handles the centralised functionality of the system; the **external layer** provides various ways in which different users access the relevant data.

Data, mirroring a particular part of the real world or a **miniworld**, is organised in an appropriate data schema; a **data schema**, reflecting the relatively stable

structure of the miniworld, is described in an appropriate data model; a **data model** abstracts the common constructs of various miniworlds. Conversely, a data model acts as a sort of language which is used to describe a variety of data schemata, while a data schema provides a framework so that the relevant data can be organised in a comprehensible manner.

One way of thinking about the relationships among model, schema and database is that a data schema is an instance of the set of structures describable using the relevant data model, and a database occurrence is an instance of the set describable using the relevant data schema. In other words, a schema consists of specialisations of components of a model, whereas a database consists of specialisations of components of a schema.

A trend of recent database research has been the development of **semantic data models**. A semantic data model captures the semantics of an application by using constructs close to those used by human beings to recognise and describe the real world, thus provides richer data structuring capabilities. Various miniworlds possess distinct properties. Apart from this, it is possible to describe the same miniworld from different points of view. It is therefore natural that a variety of semantic data models have evolved, which differ from each other primarily in the orientation and extent of detail with which data can be defined [Hull and King, 1987; Peckham and Maryanski, 1988]. Each data model, as with any particular language, may have its distinct advantages and disadvantages, meeting special needs of a certain category of application circumstances, and being preferred by a certain group of database users. In most cases, however, a broad spectrum of alternatives will probably be available.

A DBMS usually provides a very restricted set of **user interfaces** and **data models** to allow its users to access the relevant schemata and data; that is, to define and modify the data schemata, as well as to store, retrieve and update the data. For instance, a relational DBMS might provide a query language and form-interface built on the relational model together with a schema design tool built on some flavour of ER model. Such a system therefore has three interaction facilities. A model coupled with an interface forms an **interaction facility** to the functionality of a DBMS. Conventionally, in creating a DBMS, each implementation of a particular interaction facility requires a separate programming effort.

The Laguna Beach Report and its follow-up state that there is very little research on 'investigating better end user interfaces to databases or better application development tools' [Laguna Beach, 1989; Stonebraker *et al*, 1993]. It seems to be a crucial handicap

that constructing a system requires 'a mammoth amount of low-level code', with each implementation for each interface facility being a distinct programming task. In other words, the techniques with which DBMSs are implemented are not sufficient for treating data management and interface management consistently.

One kind of tool which has been designed to overcome precisely this problem is the **User Interface Management System** (UIMS) [Hartson and Hix, 1989]. A UIMS is a piece of software which deals with **human-computer interaction** (HCI) in a systematic way. This approach attempts to isolate the **dialogue component** as a structured set of **interaction elements** such as 'input a string', 'output an integer' and so on. The principal idea here is to sort out the basic **dialogue primitives** which may be appropriately selected and coupled with interaction elements to produce suitable interface for a particular application. The UIMS is designed to allow the user interface to be customisable. When UIMS methodology is combined with knowledge of database functionality, it successfully attaches interface configurability to a DBMS [King and Novak, 1989; Brown *et al*, 1990].

Cooper extended this idea a step further. In his original paper [Cooper, 1990], he indicated that Hull and King [Hull and King, 1987] demonstrated that 'semantic data models are also based on a common set of operations, which could be regarded as primitive to them in just the same way as "get a string" is to the interface.' Then he stated that 'Given a sufficient set of primitive operations, the user could "build" a data model suitable to the application in hand, rather than be constrained by the data models actually present in the system'.

The research presented here is an attempt to manage the set of interaction facilities available in a DBMS by enabling them to be created by configuring out of components, both the model and the user interface.

1.2 Configurable Data Modelling

In [Cooper, 1990], Cooper outlined the methodology for designing and constructing a system that provides generic data definition, data manipulation and data retrieval operations, and generic dialogue primitives for realising these operations, as well as generic object types, relationships and constraints. A model designer bases his or her desired data model on such a system by deciding which generic data modelling primitives should be involved and how these should be specialised in a particular data model; an

interface designer bases his or her desired interface on such a system by deciding which dialogue primitive should be allocated to each interaction element for an operation.

The initial implementation, however, was very limited, dealing only with graphical operations for designing the structural aspects of a schema. To prove the full generality of the concepts, it is essential to add data manipulation and querying, and it is also essential to manage constraints and behaviour. In order to accomplish the latter, it is necessary to undertake analysis of constraints and behavioural components which is similar to that achieved by Hull and King for the structural aspects.

The intention of creating the CDMS is to set up a system which is expected to have the ability to allow its user to describe his or her data structure using suitable data models, which themselves can be defined within the system, and the ability to allow its user to access his or her schema and data through suitable interfaces, which themselves can also be defined within the system. In this way, a single system can be used for a broad spectrum of purposes by a wide variety of users whose levels of skill and points of interest will be different. Some usage will involve very detailed facilities for complex operations, while other usage will be more superficial. Although it is not possible to construct a complete set of data modelling primitives and dialogue primitives, the CDMS will be based on generalisation of a number of prominent data models and typical interface styles. In short, the CDMS is expected to provide proper centralised facilities supporting a number of interaction facilities, which are built with reference to a central data model and can thus be created in a single implementation.

In summary, the purpose of the research presented in this thesis is to facilitate the organisation of and access to semantic or conceptual databases. For this purpose, a practical theory has been developed. The foundation of the theory, as has been mentioned, is decomposability and reconfigurability of data models and interfaces. The theory takes the form of a meta-model called the **global model**. The global model is made up of highly abstract and unconstrained constructs for each of the categories of information which can be modelled using CDMS. The methodology for using the global model is built around a **four level architecture**: the global level on the top, the data level at the bottom, and the model and schema levels between. This theory of semantic data models used in the design not only deals with the conventional static values but also extends this ability to dealing with behaviour and constraints, which are all treated here as values in a consistent manner. Theoretically, this research will provide a framework against which the comparison of various semantic data models could be conducted; practically, the

implementation of the idea offers a relatively easy way to incorporate the behaviour of various semantic data models into a DBMS.

The extension from the purely structural aspects is necessary to provide a complete account of data modelling. Constraints are vital since they carry much of the semantics of an application. The purpose of data modelling is to allow the clear specification of the miniworld. Constraints should form part of that clear specification rather than be buried in application code.

Therefore, this research emphasises the integration of the treatment of constraints within the overall framework. One of the most difficult problems in setting up the desired system is that there are many categories of constraints which occur scattered throughout a DBMS, some being inherent properties of the DBMS, others being implicitly or explicitly specified by individual users. Textbooks tend to spread descriptions of the categories of constraints available throughout accounts of the various features of database systems. Integrity constraints get the most thorough treatment, for example, in section 20.2 of [Elmasri and Navathe, 1989]. Ricardo provides a shorter account which includes the statement 'Most database systems are lacking in their ability to express constraints' [Ricardo, 1990]. Without such ability, unfortunately, database users would have to carry out *ad hoc* programming of their application to ensure that all aspects of the domain being automated are captured. In order to manage fully the information required by a number of data intensive applications, the CDMS must be able to manage constraints in a systematic way.

Similarly, the behavioural aspect of the application needs a clear specification. The shift to object orientation is a response to the understanding that complex applications require a system in which the behaviour is more coherently integrated. The CDMS approach includes the provision of behavioural constructs at the global level which may be included into the data model. This has several specific advantages:

- the operations which provide access to the data model can be configured;
- constraint management can be configured; and
- behavioural components can be included in the model - for instance triggers, transactions and exceptions could be included in a particular model.

Similar to the idea that a data model can be seen as a language to specify various data schemata, the **global model** of the CDMS can be seen as a language to specify various data models and interfaces. As the global model is the highest level abstraction of all miniworlds and interfaces in the context of the CDMS, a proper set of specialisations of components of the global model will constitute a concrete data model or a concrete interface.

Due to the diversity of the ever changing real world, and the diversity of user views of this world, it is not feasible to invent an all-embracing system. Therefore, it is impractical to attempt to set up a single universal data model to encompass all database applications. Nevertheless, it is desirable and feasible to find a relatively comprehensive set of data modelling primitives and dialogue primitives and build, based on these, a practically usable system which will support most prominent data models and interface styles in present use. That is to say, the CDMS will be based on generalisation of most existing semantic data models and interface styles.

A prototype has been set up to demonstrate the strength of the theory. The prototype system involves structural data modelling constructs, relevant constraints, as well as minimal interface facilities, so that the system will permit its user to configure data models and interfaces, specify data schemata, and manipulate data, including constraints at corresponding levels, visibly in the sense described below. The operations which realise these functions, including transactions for constraint management, will appear in the system as 'active' values which may be defined in a similar way.

The prototype makes the architecture transparent by making visible the instantiation history of a value. Thus if a database value is shown, the user can also see which schema component it is an instance of, which model component the schema component is an instance of, and which global model component the model component is an instance of. The reason for this is that it is now possible to get a clear picture of how the different aspects of the DBMS are related. Clearly a DBMS product incorporating this architecture would not maintain or show this information, but, for the development of a theoretical tool, this approach is very revealing.

Using this prototype, a variety of data models may be constructed, for each of which, again, a number of different interfaces can be matched. In short, this prototype will support the theory by demonstrating the configuration of different interaction mechanisms. For each model, relevant schemata and data can be accessed through different interfaces.

Communication of schemata and data between different models should be achievable with no more difficulty than using a federation of heterogeneous databases.

Implementing a CDMS based on the global model will require less effort than the separate development of a large number of individual DBMSs based on different data models. Moreover, setting up a CDMS will benefit the following database research aspects.

1) It will promote research in the domain of semantic data modelling theory. Using a consistent language to describe various semantic data models, it abstracts their common characteristics and thus discloses essentials of their common nature.

2) In practice, it can be used as a platform to implement various semantic data models including those already existing as well as other novel models which are describable in the context of semantic data modelling. In this way, the best suitable data model and interface within the scope of the CDMS may be obtained to satisfy particular circumstances concerning specific users and applications.

3) It will offer a basis for further development of the CDMS itself as a result of the development of the theory and practice in data modelling and human-computer interaction. Thus, more sophisticated semantic data models, and other kinds of data models such as object-oriented data models, as well as new interface styles will potentially be supported in later versions of the CDMS.

1.3 Research Procedure

The start point for the present work was a pilot implementation of the CDMS concept by Cooper [Cooper, 1990]. This program, written in the persistent programming language PS-algol, allowed multiple data models to be configured for schema design and multiple graphical schema representation styles to be configured for each data model. The limitation was that this version involved neither ability of manipulating data nor ability of dealing with constraints or active objects.

The present work progressed by a technique of extracting general properties from individual data models and incorporating them into the CDMS prototype. Certain stages can thus be identified:

1) Implementation of the IFO data model, including data definition, manipulation and querying facilities, using the persistent programming language PS-algol.

2) Design of the **layered architecture**, which consists of **global level**, **model level**, **schema level** and **data level**. This architecture replaced Cooper's *ad hoc* architecture.

3) Re-engineering and implementing the prototype based on the design from stage 2 in the more powerful persistent programming language Napier88. The new system provided fuller coverage of data structuring components with simple data manipulation and querying facilities.

4) Extension of the IFO data model to include constraint handling facilities. In this way an understanding of the more general nature of constraints in a DBMS was obtained.

5) The development of a general model of constraints from stage 4.

6) An extension of the CDMS from stage 3 to manage constraints.

7) The development of a process modeling tool to gain an understanding of active objects.

8) The development of a general model of active objects from stage 7.

The obvious next stage is the extension of the CDMS from stage 6 to treat active objects, which is unfortunately beyond the scope of a single PhD project. However, its possibility is discussed in the conclusions.

1.4 Structure of Thesis

In the following chapters, the relevant work will be described and discussed.

Chapter 2 discusses the main issues of data modelling and human-computer interfaces, including principles of databases, database systems, database management systems, data modelling, interface independence, and federated database systems. A brief overview of data models and database programming languages is also given in this chapter.

Chapter 3 presents the main issues of the Configurable Data Modelling System systematically. A consistent CDMS structure is developed based on an analysis of data modelling practice. Sets of generic modelling primitives, interaction elements and dialogue primitives are described; the problems concerning constraints and active objects are proposed in the context of the CDMS.

Chapter 4 is dedicated to issues relating to constraints in the CDMS. Based on the analysis of varieties of constraints in the context of the DBMS, basic forms of constraints are elicited. Constraint specification, verification and management issues are then discussed.

Chapter 5 discusses the behavioural issues relevant to databases. Transaction systems are discussed as an introduction, followed by the presentation of a process support system. Active objects in semantic data modelling and the CDMS form the main body of this chapter.

Chapter 6 presents the platform for the implementation of the CDMS. This chapter describes the implementation tools, introducing PS-algol and Napier88, two persistent programming languages, and the relevant user interface tools.

Chapter 7 reports on an implementation of the CDMS. The implementation provides the ability of managing constraints as well as basic constructs consistently in the context of the CDMS. In this chapter the coherence and consistency of the CDMS structure is emphasised throughout.

Chapter 8 presents some applications of the CDMS. A number of data models are constructed within the CDMS to prove the usability of the platform.

Chapter 9 briefly describes the contributions of this work and proposes some potential future work as conclusions of the project.

2 The Usability of Database Systems

This chapter examines the nature of database systems, what they are used for and how they are used. The start point is the architecture of database systems and a description of the activities which must be carried out to use them. There then follows a discussion of the role of the user interface in the use of database systems. This is followed by a discussion of the use of data modelling in database design together with a survey of some of the leading data models. The chapter continues with a look at federated DBMSs as one example of the need for supporting multiple data models consistently, before summarising in order to motivate the work described in this thesis.

2.1 Databases and Database Systems

2.1.1 What Is a Database?

Databases play an important role in almost all areas in which computers are used, including scientific research, engineering, medicine, education, business, administration, and so on. Generally speaking, data may be generated and maintained either manually or by machine, but databases of large size and high complexity are feasible only with computers. The word 'database' is used so widely that we must clarify what a database is to avoid confusion.

A **database** is a collection of structured data which are well organised, implicitly meaningful, and able to be recorded, processed and retrieved. In practice, the following properties of a database should be emphasised:

- A database represents some aspect of the real world called a **miniworld**. A database is therefore a logically coherent collection of data values with some inherent meaning.

- A database is designed, built and populated with data for specific purposes. A database must therefore have an intended group of real users, including application programmers, and some preconceived applications.

- A database should be able to reflect changes in the relevant miniworld dynamically.

- A database is stored using the structures of an internal data model.

A **database management system** (DBMS) is a software system that enables users to create and maintain various databases. In other words, a DBMS is a collection of programs that facilitate the processes of defining, constructing, and manipulating data for various applications. Defining data implies specifying the types of data to be stored in the database. Constructing data is the process of storing data on some storage medium that is controlled by the DBMS. Manipulating data means such functions as querying a database to retrieve specific data, generating reports from the data, and updating a database to reflect changes in the miniworld. Databases together with the relevant software are referred to as a **database system.**

The database approach is distinguished from the traditional approach of programming with files by a number of fundamental characteristics. In traditional file processing, each user defines and implements the files needed for a specific application; while in the database approach, once a single repository of data is defined it can then be accessed as required by various users. The main characteristics of the database approach are the following:

1) **Self-containment**. A database system contains not only the data in a database but also a complete definition or description of the database structure, which is stored in the **system catalogue** (data dictionary) and called the **metadata**. The DBMS software then refers to the catalogue to find the structure of the data files, the types and storage formats of the data items, as well as any constraints on the data in a particular database. Thus the same software is able to access many different databases and work efficiently with various database applications.

2) **Program-data independence.** DBMS access programs are written to function independently of any specific files. Owing to the existence of the system catalogue, in which the structure of data files is stored, there is no need for the structure of any file to be embedded in the access programs. Nevertheless the structure of the metadata must be involved in the DBMS access programs to permit the programs to refer to the catalogue from time to time. By contrast, in traditional file processing data definition is typically part of the application programs, hence any changes to the structure of a file may undesirably require changing all programs which access that file.

3) **Data abstraction.** A DBMS provides users with a conceptual data representation that ignores many of the details of data storage. The database users only need to refer to the conceptual data representation, while the DBMS automatically extracts the details of file storage from the catalogue. The data model and the data schema are the main concepts which are closely related with data abstraction and these will be further investigated later.

4) Support of **multiple user views.** A database usually has many users who may be interested in different parts of the data in the same database. A view may be a subset of the data or it may contain virtual data that is derived from the database files but is not explicitly stored. A multi-user DBMS should therefore be able to provide facilities for defining multiple views of a database. Because the database approach aims at efficient redundancy control in defining and storing the data, it becomes very important to allow data to be shared satisfactorily by various users with maintenance facilities, including concurrency control, access authorisation, backup and recovery, etc, being available.

2.1.2 The Architecture of Conventional DBMS

A **three layer architecture** is widely used to separate the physical database and user applications, the layers being an internal layer, a conceptual layer and an external layer. The **internal layer** has an internal schema, which uses a physical data model to describe the physical details of storage structure and access paths for the database. At this level, the data is described in terms of the files and disk addresses where the data is to be found. The **conceptual layer** has a conceptual schema, which uses a conceptual or

implementation data model to describe the logical structure of the database, concentrating on such concepts as entities, relationships, and constraints, but hiding the physical details of storage structure and access paths for the database. The **external layer** or **view layer** includes a number of external schemata or user views, each of which describes the part of the database that a particular user or user group is interested in and hides the rest of the database.

The three schemata are all descriptions of the data, which actually exist only in the physical layer. Based on the three-layer architecture a DBMS transforms a request specified on an external schema into a request on the conceptual schema, then further into a request on the internal schema for processing on the stored data. Also based on the three-layer architecture a DBMS transforms a result found on an internal schema into a result on the conceptual schema, then further into a result on the external schema for viewing. The processes of transforming requests and results between layers are called mappings.

Data independence is the capacity to change the schema at one layer of a database system without having to change the schema at a higher layer.

There are two kinds of data independence. The first is **logical data independence** while the second is **physical data independence**. The former means the capacity of changing the conceptual schema without having to change the external schema or application programs; the latter means the capacity of changing the internal schema without having to change either the conceptual schema or the external schema.

The system catalogue of a layered DBMS must include information on how to map requests and data among different layers. When a schema at some layer is changed, the schema at the next higher layer should remain unchanged. The only thing having to be changed is the mapping between the two layers.

A substantial benefit promised by the three-layer architecture is that a conceptual schema can be realised by multiple equivalent internal schemata on the one hand, and can be revealed to the user by multiple external schemata on the other, without requiring the basic DBMS facilities to be programmed more than once. In practice, many valuable data structures for efficient physical storage have been developed, and the promise of multiple physical representations each of which produces efficient access in different situations has been fulfilled. Unfortunately the promise of multiple user interfaces has not been maintained to the same degree.

2.1.3 The Activities Required

Users need to perform a number of tasks with the database, which include:

- **Data definition**. Database administrators (DBAs) specify the conceptual and internal schemata of database applications, as well as the external schemata, i.e. various views, for particular users.

- **Data manipulation**. Database end-users populate the database and then update the data as needed.

- **Data retrieval**. Querying the database is obviously a major function for end-users.

The data definition activity involves the user in creating a description of the structure of the data. This description is called a **schema**. The schema is constructed in terms of a **data model**, i.e. a language with constructs which are primarily concerned with data structure but may also deal with constraints and even behavioural aspects of the data. As has been discussed previously a data model may concern itself with any of three levels of the architecture. Even within a level there may be more than one data model available for use.

Data manipulation and retrieval occur in the context of a data schema and allow values manipulated in the structures of the data schema. There will be operations to create, remove, update values, as well as operations which selectively return a subset of the values.

For each of these tasks the database system will provide suitable languages.

A DBA will have access to a **data definition language** (DDL) for specifying the conceptual schema, a **storage definition language** (SDL) for specifying the internal schema, and a **view definition language** (VDL) for specifying user views. Both DDL and SDL are also used to specify the mapping between conceptual schema and internal schema, while VDL is also used to specify the mapping between user views and conceptual schema. The end user will have access to a **data manipulation language** (DML). The languages may be presented to the user in a variety of interface styles and it is to these that attention is now turned.

2.2 User Interface to DBMS

2.2.1 Styles of Interface

The user **interfaces** through which a DBMS is manipulated come in a variety of styles: command language interfaces, menu-based interfaces, form-based interfaces, graphical interfaces, natural language interfaces, and so on.

- Query languages. The relevant statements in these languages may be used independently, or they may be embedded in a general-purpose programming language.

- Menus do away with the need to memorise the specific commands and syntax of a query language.

- Forms are usually designed and programmed for naive users as interfaces to canned transactions.

- Graphical interfaces allow data retrieval and manipulation diagrammatically.

- Natural language interfaces are expected to accept requests in English or some other human languages and attempt to understand and process them.

These are appropriate to different classes of users, of whom it is usual to distinguish:

- Naive end-users need to learn very little about the facilities provided by the DBMS; they have only to understand the kinds of standard transactions designed and implemented for their use.

- Casual users merely learn a few facilities that they may use repeatedly but infrequently.

- Sophisticated users need to learn most of the DBMS facilities in order to meet their complex requirements.

- The system analysts and programmers also design and compose some special interfaces for specific users.

- In addition, the DBA may have some specific interface to realise some restricted functions, for instance, changing a schema.

It is desirable for a variety of interfaces to be supplied for the same access functionality, so that various users and distinct applications will be able to communicate with the computer in their individually preferred and beneficial way. This is now potentially possible, but still rarely realised, because this usually asks for a great deal of *ad hoc* programming effort.

2.2.2 Database Programming Languages

For the design and support of complex applications, it has become increasingly clear that the kinds of interface described in the previous subsection are insufficient. Instead, a combination of programming language and database facilities is required. **Database Programming Languages** (DBPL) attempt to provide such a combination. Some of the kinds of DBPL are discussed briefly here, but the subject will be returned to in Chapter 6. This chapter describes the implementation framework used for this project and this framework requires a well engineered DBPL to enable the complex tasks involved to be tractably accomplished.

The following classes of DBPL can be distinguished:

- Relational DBPL's. These languages embed data types for relations in a programming language thus allowing the database to be seamlessly manipulated by a program [Schmidt, 1977].

- Polymorphic DBPL's. These languages, influenced by ML [Milner, 1984], provide mechanisms which allow generic operations to be produced which operate against a variety of data [Matthews, 1985].

- Persistent Programming Languages. These languages permit data of any type to be long-lived [PS-algol, 1987; Morrison *et al*, 1989], while conventional

programming languages restrict the types of long-lived data, e.g. serial files in Pascal.

- Object-Oriented (O-O) Database Languages. These are either the addition of a database to an O-O language [Maier *et al*, 1986; ServioLogic 1987] or the addition of a programming language to an O-O data model [Lecluse *et al*, 1988].

2.2.3 Interface Independence

The vigorous development of the computer industry and the rapid popularisation of computer applications has meant that realising the potential of a modern computer has become less limited by its computing power than by its power to communicate with human users, more and more of whom are now people other than computer professionals. The study of **human-computer interaction** has therefore been recognised to be of vital importance.

Interface independence is an important factor in improving human-computer interaction. The concept is a logical extension to machine independence, language independence, data independence, etc. It argues for a human-computer interface/dialogue independent of the rest of the application.

Strictly speaking, **human-computer dialogue** and **human-computer interface** are conceptually different. The former means the communication between a human user and a computer system, while the latter means the communication medium. But they will be used synonymously in this thesis just as in most of the literature, for the two terms are tied together closely in the development process.

2.2.3.1 Dialogue Component and Computational Component

Dialogue independence is a characteristic of an **interactive software system** that separates design of the human-computer dialogue software from design of the internal computational software of an application system. This separation requires that design decisions affecting only the human-computer dialogue should be isolated from those affecting only the computational structure of the application. Consequently, the appearance of the interface to the end user and choices of interaction styles used to extract inputs from

and present outputs to the user are transparent to the computational software, modifications in either causing minimal change in the other.

From the stand of a computer, the human-computer **dialogue component** includes the acceptance of inputs from the user and the presentation of outputs to the user; whereas the other parts of transformation between inputs and outputs comprise the **computational component**. Dialogue and computation will thus be distinguished from each other.

Separation of the dialogue component from the computational component requires a new interface between them and a new sort of dialogue through the interface. The computational component uses the dialogue component as an intermediary instead of communicating directly with the end-user.

The relationships among the major concepts mentioned above can be roughly illustrated by Figure 2.1.

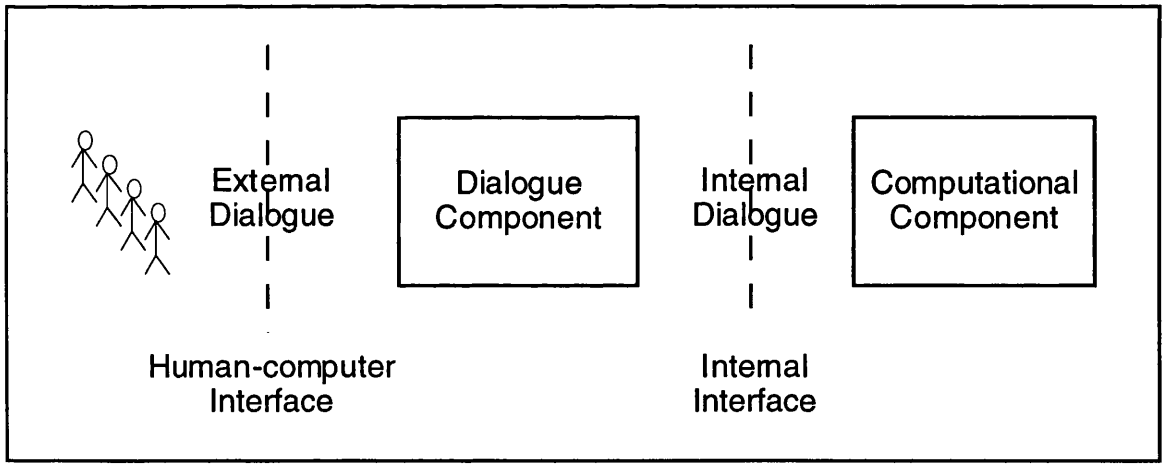

Figure 2.1 External Dialogue and Internal Dialogue

Internal dialogue is not readily understandable at execution time, but its representation at design time is essential to interface independence. As long as both dialogue component and computational component keep consistent with their common internal dialogue representation, they are free to change without affecting each other.

2.2.3.2 The Advantage of Interface Independence

Early approaches to the development of interactive systems involved close interspersion of dialogue and computational software and often resulted in an unsatisfactory

human-computer interface. Without dialogue independence, both the way in which dialogue was structured and the details of how it was conducted with the end-user were usually driven by the computational requirements of the application system. Knowledge of dialogue details and decisions about interaction styles were intermixed with the computational component, so that it could be very difficult as well as time-consuming to modify such a system as the design progresses.

In contrast, dialogue independence permits fast modification, iterative refinement, and easy maintenance of dialogues to meet users' ever-changing needs, because a system development approach based on this concept would allow changes to be limited to the interface component, which is relatively independent of the rest of the application system.

More attractively, dialogue independence permits more than one interface to be defined and utilised as required for the same underlying computational component to satisfy various users' preference. In other words, owing to the independent nature of the interface, the development of user configurable interface systems become a meaningful task.

Dialogue independence can also be used as a design abstraction to allow a top-down approach, focusing first on high-level design issues while postponing commitment to details.

2.2.3.3 Requirements Arising from Dialogue Independence

First and foremost, a new software development role, the **dialogue developer**, now becomes increasingly important. This role is carried out by a human factors specialist concerned with the design, implementation, and evaluation of the form, style, content, and sequencing within the human-computer interface. The dialogue developer is involved in the entire system life cycle, using an understanding of psychology and human factors principles to build, evaluate and refine iteratively an interface which permits effective human-computer communication.

Some dialogue development tools require that the dialogue interpreter and design tools possess considerable knowledge of the style of interaction. This in turn requires a great deal of programming of new interaction styles, techniques, and devices. Due to the separate existence of dialogue developers, the need for communication among

such developers and implementors will increase. Some new development methodologies are being developed to deal with this problem. An important concept in human-computer interface management is a methodology for interactive system development. This methodology considers interface management as an integral part of the overall development process and gives emphasis to evaluation in the development life cycle.

Separation of the dialogue part from the computational part leads to increased internal communication among components, which may cause a decrease in system performance. To some extent, this can be improved by adopting the system architectures which allow concurrent execution of the dialogue and computational components, as well as by using newly developed dialogue supporting hardware. Multi-threaded, direct manipulation dialogue is more effective than sequential dialogue, for end-users can directly, visually, and asynchronously perform operations on interface representations of application objects. However, in this case separation into components can be much more difficult to accomplish than for sequential dialogue, because the dialogue and computation are often more closely interleaved, and the two components often share a common data representation of the interface and application objects. Nevertheless, design decisions regarding the appearance and behaviour of the interface can often be kept independent of those for the software manipulating the corresponding data structures.

Representation of the human-computer interface is a mechanism for expressing and recording the results of dialogue development. Written programs, textual representation languages and graphical representation languages are various sorts of such representation measures. Unfortunately, language-oriented representational techniques are inappropriate for representing direct manipulation style dialogue, for there exist so many visual and other perceptional aspects requiring representation. A new mechanism is automated tools for interactive production of the interface representation.

Rapid prototyping as a system development technique provides at least one early version of the application system, showing the end-user what the interface and the whole system will look like before it is actually developed. Prototyping, as an extension of software simulation, increases communication among system designers, implementors, evaluators, and end-users. It is important to ensure that a high-quality system is developed through prototyping in situations where the system requirements are complex or uncertain.

The iterative nature of human-computer interface development alters the conventional linear development life cycle [Hartson and Hix, 1989]. A prototype helps to solve the problem of end-users' inability to give complete specifications to system

designers, and 'gives the end-user a more immediate sense of the proposed system' [Wasserman and Shewmake, 1982]. It discloses misunderstandings that occur between developers and end-users caused by their different backgrounds and experience [Gomma and Scott, 1981].

2.2.4 Configuring the User Interface

In the context of database systems, two important ideas modify the previous discussion. Firstly, the computational component of the software as far as the user is concerned is fairly restricted to the activities described in Section 2.1.3. The user interface only has to support database design, data manipulation and querying in the context of a particular data model* .

Secondly, the range of users is great, from those with no particular experience of computer systems to database specialists. Thus many kinds of interface are desirable. This particular mix of restricted computation with many users makes it particularly suitable for a configurable approach to provide the user interface. Instead of resorting to repetitive coding on an interface by interface basis, it would be better to provide a combinable set of primitives and frameworks out of which particular interfaces can be developed.

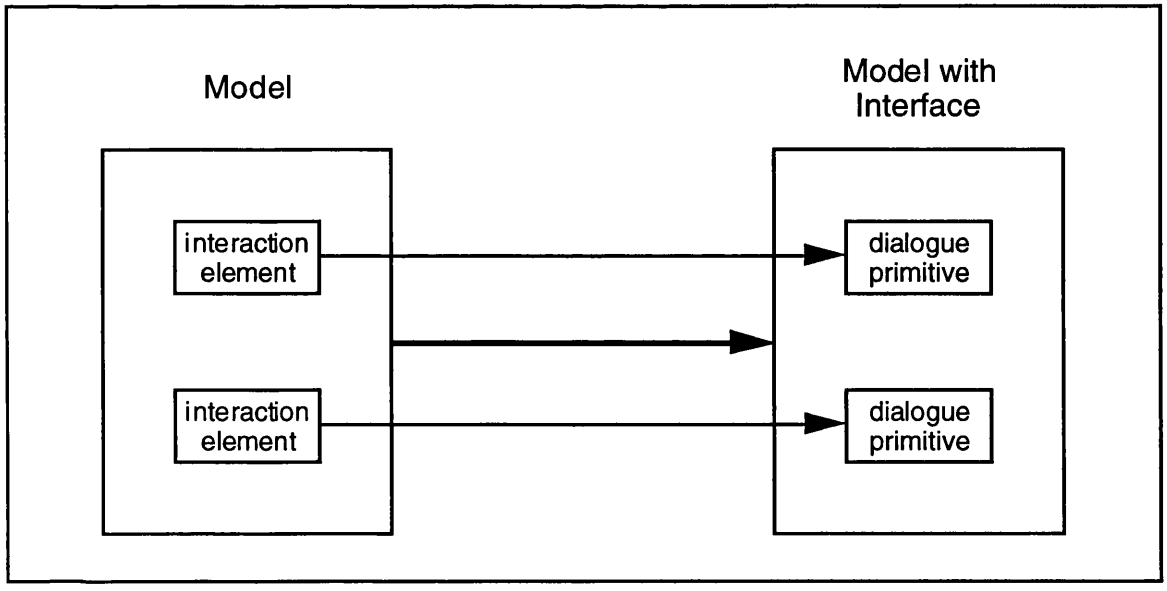

Figure 2.2 User Interface Configuration

* Of course there is a great deal more computation in a DBMS, namely storage structures, concurrency mechanisms, indexing and so on, but these do not involve close interaction with the user.

The architecture for such a system is shown in Figure 2.2. A data model includes a series of interaction elements, each of which will be replaced by a concrete dialogue primitive as a user interface to the model is configured. This issue will be further discussed briefly in Section 3.7. The next section, however, discusses the notion of a data model in detail.

2.3 Data Modelling

2.3.1 What Is a Data Model?

A **data model** is a group of concepts for describing the structure of a database, including the kinds of relationships and constraints which may hold on the data to be kept in the database. Some data models also include a collection of operations for specifying updates and retrievals on the values in the database. In other words, a data model is a set of abstraction mechanisms, with associated operators, used to define and manage data schemata. Data models are the main tool for providing data abstraction, since they hide details of data storage, which are not needed by ordinary database users.

There have been proposed a diversity of data models, which can be categorised into three kinds, based on the types of concepts they provide to describe the database structure. A high level or **conceptual data model** provides concepts that are close to the way in which many users perceive data, whereas a low level or **physical data model** provides concepts that describe the details of how data is stored in the computer memory. There is, between the two extremes, another class of data models called **implementation data models**, which hide some details of data storage, providing concepts which can be understood by end users, and which can also be directly implemented on a computer system. These kinds of data model thus correspond to the three layers in the DBMS architecture discussed in Section 2.1.2.

The concepts typical of a conceptual data model include entities, attributes and relationships. An **entity** is the representation of a class of real-world object in the database. An **attribute** is a property of an object. **Relationships** reflect various connections among real-world objects. Conceptual data models are usually referred to as **semantic data models** [Hull and King, 1987]. These models mainly describe objects and their inter-relationships and sometimes are referred to as **object-based** models. One

of the most popular high level data models is the **entity-relationship model** (ER) [Chen, 1976].

The implementation data models most widely used are the **hierarchical, network** and **relational** models. They are sometimes referred to as **record-based** models, because all of them use record structures to represent all data. Records are an intermediary structure. They are not a direct representation of physical storage since records may be fragmented about the disk or grouped in a variety of ways. On the other hand, a record could be considered a conceptual level structure in that records directly model many of the information structures which require handling. However, record-based models cannot be considered to be semantic data models since there is a wide variety of information which is not easy to deal with using record-based models.

A **data schema** is the description of a database structure in the language of a data model. Data schemata are not expected to change frequently, and are specified during the database design phase. A DBMS stores a data schema in the database catalogue as metadata. This allows the DBMS software to refer to it later in order to find out the structure of the relevant database as required. The data contained in the database at a particular moment in time is called a **database instance**, which usually changes much more frequently than does a data schema.

A data schema is called the **intension** of the database instances, and a database instance can be called an **extension** of the schema. A data schema is, in essence, a description of many database instances with the same structure. A schema for a library database may thus be used by a number of library systems. On the other hand, a data model can be appreciated as the corresponding database type system, which supports a number of data schemata by providing common data definition tools. The relationship among database, data schema, and data model will be further clarified in chapter 3, where suitable examples will be given.

Incidentally, it should be noted that the word 'model' is sometimes used to denote a schema by some authors in the sense that a schema is, in fact, a kind of abstraction of the relevant miniworld with only structural knowledge being considered. It is hence possible to speak of a library model, meaning the schema for library databases. However in this thesis the word 'model' will never be used in this sense to avoid confusion.

2.3.2 An Overview of Data Models

The **network model**, the **hierarchical model** and the **relational model**, that is, the so called **classical data models**, were developed in the early stage of data modelling practice. The Relational Model [Codd, 1970] proved to be an elegant framework, in which all kinds of data are described in the form of relations or tables, with columns named and typed, and rows undistinguished. The mathematical features of the model have encouraged a great deal of research leading to the optimisation of data storage and retrieval. Some standards have been established, for instance, the query language SQL which permits data to be shared between different DBMS. Many business applications can be described in terms of relations satisfactorily, so that the relational model has become an effective and efficient medium for data intensive applications which have a relatively simple nature.

However, the Relational Model is inadequate for many application areas, such as **computer-aided design** (CAD), **computer-aided manufacturing** (CAM), **computer-aided software engineering** (CASE), **computer-aided engineering** (CAE) and **office automation** (OA). It is extremely difficult to manipulate objects with complex structures using the relational model, which, being simple in its underlying structure, will inevitably enforce a heavy burden on software engineers. The point is, as Kent indicates [Kent, 1979], the relational model provides two mechanisms for relating two pieces of data: either they are in different fields of the same tuple, or they are in two tuples with a common field, which acts as a 'foreign key'. These two mechanisms are each used for a variety of purposes, thus inducing semantic overloading. Consequently, the information concerning one object may be distributed over a number of tuples, or may just be part of the same tuple, and so there is no simple mapping between real world objects and database values. In fact, the Relational Model fails to provide a consistent way of describing and identifying single objects. This in turn makes it difficult to construct values which refer to other values as components or attributes. It quite often becomes too difficult for someone to understand the function of a part of a relational schema which has been established by someone else.

The limitations of the classical data models led to the development of models which strive to capture more of the meaning of applications. **Semantic data models** [Hull and King, 1987; Peckham and Maryanski, 1988] provide more abstract concepts, greatly easing the description of the structure of a database. Thus, within the database context, powerful facilities are emerging for expressing complex data intensive real-world applications straightforwardly.

Most semantic data models concentrate on a description of the structure of the database, while others include also an active component. The latter allows a direct and consistent description of processes with which the database can be manipulated. Many of the ideas of semantic data models were incorporated into the **object-oriented** (O-O) **approach**, which will be further discussed in Subsection 2.3.3.

The first semantic data model was the **semantic binary data model** (SBDM) proposed by Abrial [Abrial, 1974]. This model was intended as a design tool for relational databases, introducing constructs for describing entity types called categories and binary relationships between categories. Each database design, such as the one shown in Figure 2.3, proceeds by using these two constructs: objects are grouped into categories (*book*, *title*, and so on) and then categories are interrelated in particular binary relationships (usually bidirectional). Thus in the figure, *borrows* and *is-borrowed-by* are inverse. Although the SBDM was a very simple modelling system, it set the precedent for modelling real-world notions in a straightforward manner. It should also be noted that the SBDM contained a behavioural element - for instance, it was possible to trigger a piece of code by the creation of an object. Thus even the earliest model aspired to capturing application behaviour.

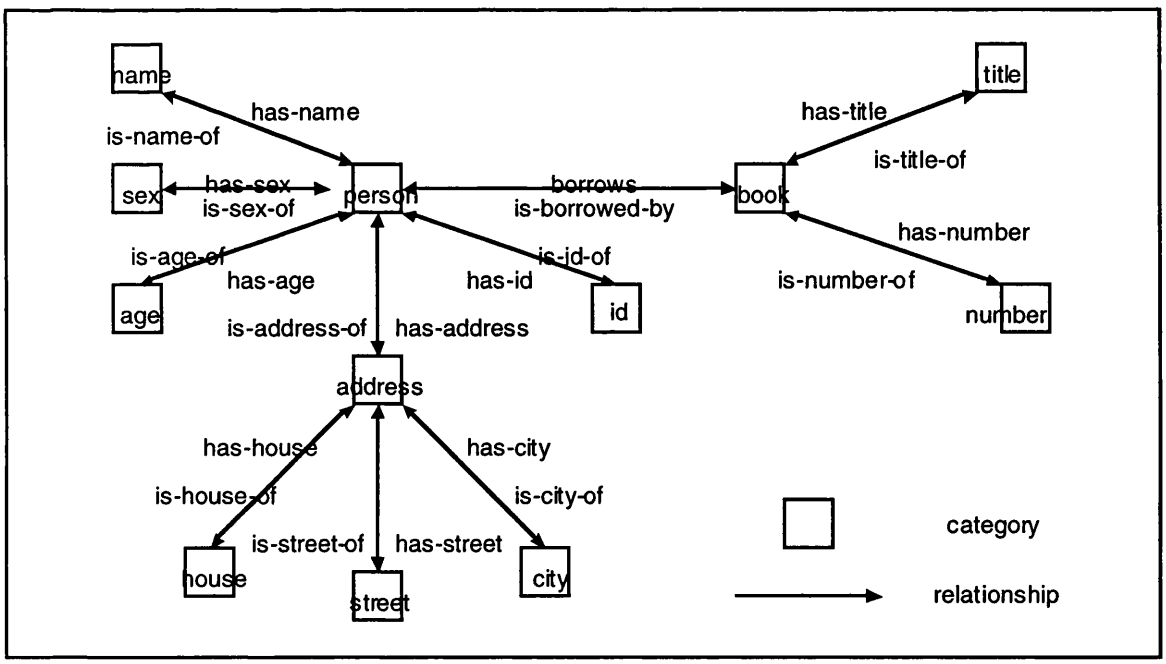

Figure 2.3 A Schema in the Semantic Binary Data Model

The **entity-relationship model** (ER), introduced in 1976 [Chen, 1976], extends the modelling power of the SBDM, although it loses the behavioural component. It was the model which first popularised the semantic data modelling concept and still

remains very popular. Being similar to the Bachman diagrams proposed for CODASYL databases [Bachman, 1969], the ER model has been used as an off-line graphical design tool for relational database systems and is increasingly used as the data definition interface to commercial systems.

In ER, entity sets are equivalent to categories in the SBDM, relationship sets interconnect entity sets, and atomic valued attributes are allowed on both entity sets and relationship sets. In addition, this model has the ability to represent different kinds of entity set. One kind has primary keys and is referred to as a strong entity set; another kind derives part of their keys from some other entity set and is referred to as a weak entity set. Entities belonging to a weak entity set are identified by being related to specific entities from another entity set in combination with some of their attribute values. This other entity set is called the identifying owner, and the relationship set that relates a weak entity set to its owner is called the identifying relationship of the weak entity set. All of these are easily transformed into a relational database schema [Teorey, 1986]. The ER model provides a relatively straightforward environment for modelling the structure of the objects in a database. It is easily mastered, but does not provide much depth of description. Figure 2.4 shows a data schema in the ER Model.

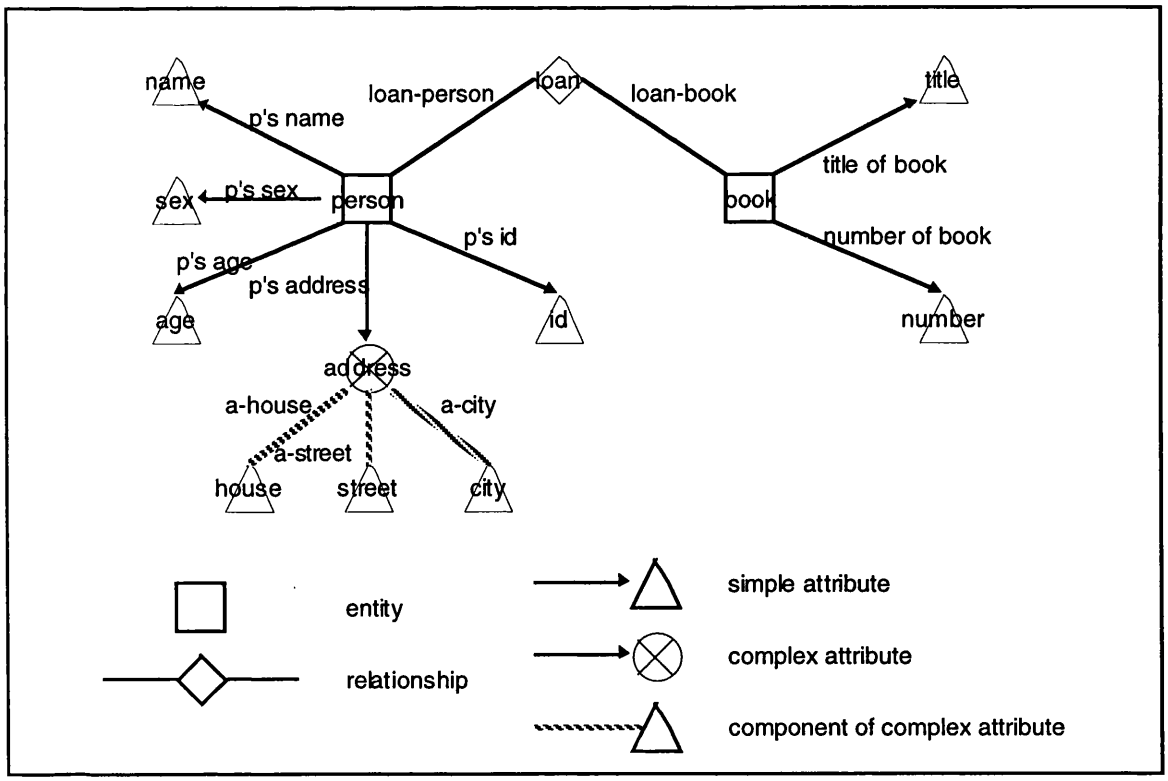

Figure 2.4 A Schema in the Entity-Relationship Model

The **semantic data model** (SDM) [Hammer and McLeod, 1981] is a very rich structural data model concentrating on the concepts of entity types and relationships. The fundamental modelling construct for entity types, called a class, is defined to have a name, a set of members, a description for documentation purposes, a set of member attributes affiliated to each member of the class (for instance address), and a set of class attributes affiliated to the class as a whole (for instance cardinality).

Classes are divided into base classes and non-base classes. A base class is defined independently, while a non-base class is defined with respect to other classes. If a class is a base class, then there will be stored data holding the instances of the class. In this case, whether duplicate members are permitted should be clarified and the set of member attributes which form the key, if appropriate, should be specified. A non-base class derives its instances through some form of calculation, such as: grouping, where a partitioning expression splits a class into a group of sub-classes; sub-classing, which is realised by a filtering membership predicate; as the intersection of two classes; as the range of some attribute on another class; or by user-definition.

An attribute can be either mandatory or optional, either single-valued or multi-valued, either changeable or not, either unique or not, either exhaustive or not, either derived or independent. There are also several sorts of derived data and constraint types, and there is an inheritance mechanism between the classes. The SDM is hence very rich in its modelling capacity, but using this data model sometimes causes difficulty in defining and perceiving the semantics.

In contrast, the **functional data model** (FDM) [Shipman, 1981], providing only a single modelling construct, describes the database in a set of functions mapping entities to entities. All concepts, including entity types, sub-typing, attributes, relationships, single-valued and multi-valued data, base data and derived data, even metadata, are described by functions. Functions in this model can have zero, one, or more arguments. Entity types are defined by functions with no arguments, whereas attributes of entities and relationships among entities are defined by functions with arguments. A single-argument attribute is represented by a single-argument function, whereas a multi-argument attribute is represented by a multi-argument function. On the other hand, functions can also be specified as single-valued or multi-valued. Entity types in this model are arranged in a type hierarchy with automatic inheritance of functions, including attributes and relationships, from a supertype to all of its subtypes. Derived entity types are defined by using functions which imitate aggregation, set union and intersection; while derived

attributes can be defined by using functions, function composition, or aggregating functions.

The **extended functional data model** (EFDM) [Gray *et al*, 1992], is the implementation based on the FDM by Kulkarni [Kulkarni and Atkinson, 1987]. The EFDM offers an interactive user interface to create and manipulate relevant information. The capabilities of defining both stored data and derived functions, integrity constraints and views are also provided by the EFDM. The EFDM describes the metadata with the same constructs used to describe the data. An example of multi-argument function is the function *grade*, which has two arguments, *student* and *course*, and which indicates that every student-course pair corresponds a string entity denoting the grade that the student gets for that course. The function grade, however, should only be defined for those courses in which the student is enroled. In EFDM, this sort of semantic rule can be described by a constraint.

Generally speaking, the EFDM offers the following constraints: ISA constraint, single-valued property constraint, and inverse property constraint; and it allows the following constraint to be specified explicitly:

- key constraint, which requires that an entity can be identified by a group of function values taken together;
- disjoint constraint, which requires that the objects of a group of entity types must not overlap;
- required property constraint, requires that a function must have values for each argument entity;
- fixed-value constraint, which requires that the values of a function cannot be changed once they have been created;
- range constraint, which requires that the values of a function can only be drawn from a limited range.

The EFDM does not offer for capturing such constraints as property induction constraint, unique property constraint and covering constraint, but the constraint mechanism of EFDM could be extended easily to handle these.

The FDM is a modelling system with both simplicity and great power. However, a lot of the semantic contents of databases are lost while using the FDM. Neither is it always clear from a schema what role a given function is playing.

Figure 2.5 shows a data schema in the FDM.

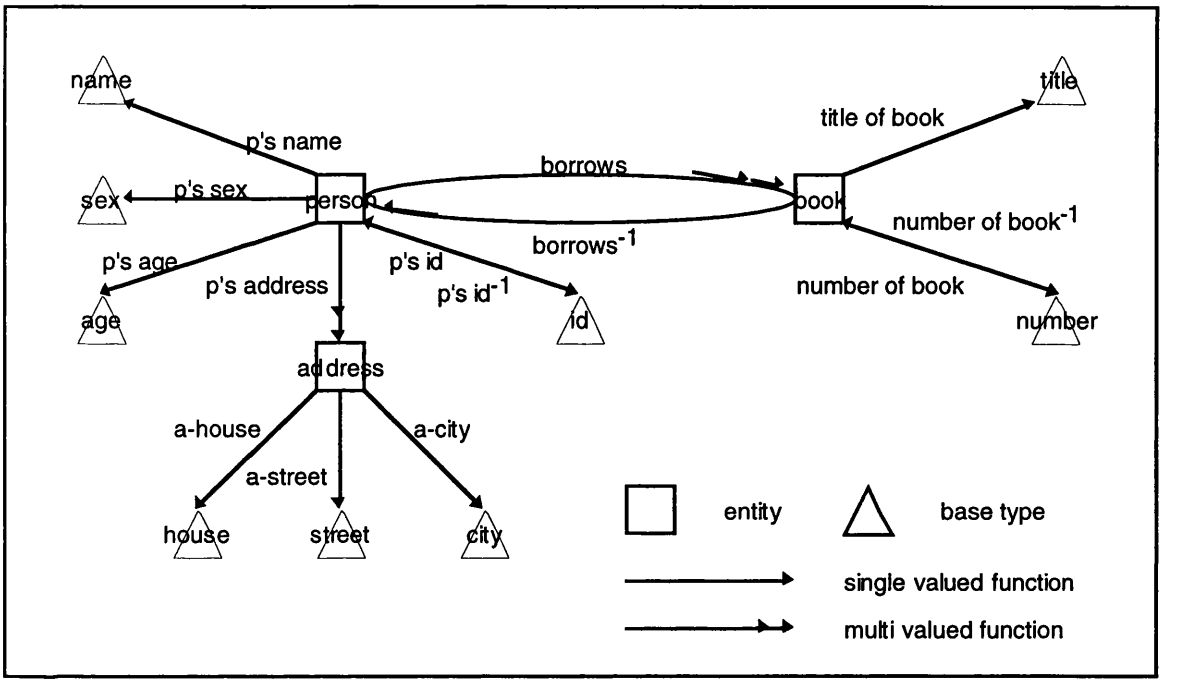

Figure 2.5 A Schema in the Functional Data Model

The **IFO model** proposed in [Abiteboul and Hull, 1987] has three basic constructs, namely objects, fragments (representing functional relationships) and ISA (class/subclass) relationships. IFO aims at producing a theoretical framework for examining the structural aspects of semantic data models by providing a sophisticated categorisation of types. The model incorporates attributes and type constructors for grouping and aggregation at a fundamental level and distinguishes between two kinds of ISA relationships. The model is also used to characterise the propagation of simple updates and to analyse formally the interplay between constructed types and ISA relationships. The IFO model is able to represent **atomic** object types and **constructed** types. Each of the atomic types corresponds to a class of non-aggregate objects in the world. There is also a distinction between atomic types that are **abstract** and those that are **printable**. Abstract types correspond typically to objects in the world that have no underlying structure (at least, relative to the point of view of the database designer or user) such as *person*, while printable types correspond to objects of predefined types that serve as the basis for input and output such as strings. There is another kind of atomic type - the **free** type. Free types are defined with reference to other types by inheritance. There are two complex type

constructors. **Sets** represent multi-valued objects, and **aggregates**, i.e. records, represent single objects consisting of component parts. An aggregate can be used to encapsulate information, for instance, *address* comprises *house*, *street* and *city*. A set or grouping construct is used to represent sets of objects of the same type, for instance, *class* represents sets of *student*. This situation is illustrated in Figure 2.6.

The types are connected by various relationships available in the model, including attribution, component (of a set or aggregate), specialisation and generalisation. Specialisation represents the notion that type X is a sub-type of Y in the sense that X inherits the properties of Y and all Xs are also Ys. The specialised type, X will be a free type, since it is defined relative to the type being specialised. On the other hand, generalisation creates a free type which is the super-type of a set of other types. It encompasses the notion that every instance of this type must also be an instance of one of the sub-types.

IFO allows some local constraints to be specified, such as that a relationship is 1:1 etc. IFO also allows constraints on specialisation relationships, for instance, the subtypes of a certain type must be disjoint or they should cover the supertype. There are also global constraints, for instance, the sub-type graph must be acyclic, no type could be the specialisation of more than one atomic type, and no free type could be specialised and generalised from some other types simultaneously. IFO places the concepts which were introduced in the preceding models into a relatively simple framework in which the intrinsic nature of the constructs is more clearly revealed. This represents the first step towards analysing data models formally [Abiteboul and Hull, 1987].

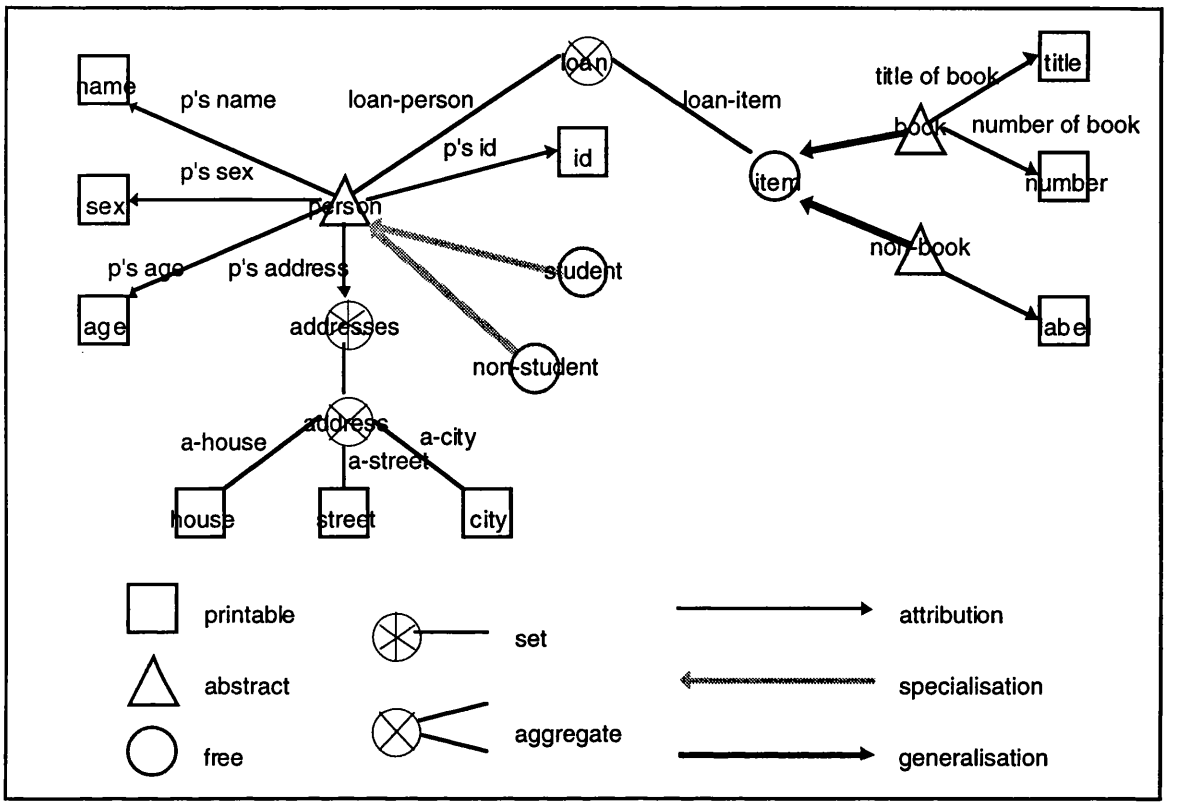

Figure 2.6 A Schema in IFO

For complex applications, it is crucial to use data models which capture the data structure in a more meaningful way than do the classical models, particularly for non-expert implementors. Furthermore, it would increasingly be valuable to use the same modelling tool (or an analogous one) to describe the **behavioural properties** of the database. Actually, the worlds of CAD, CAM, CASE, CAE and OA increasingly need the ability to describe behaviour in the same way as the structure.

Semantic data models, unfortunately, have tended not to address the behavioural aspects of a data application. The need for more programming power in this area resulted in the development of **database programming languages** (DBPL), which will be discussed in Chapter 7. Nevertheless, two examples of the kinds of data model which deal with **behaviour** are introduced in Chapter 5.

To gain understanding of how data models can be implemented, an IFO data model program was created [Cooper and Qin, 1990]. This program provides a graphical schema design tool for the creation and manipulation of IFO schemata, and further provides facilities for populating a database with data and for manipulating and querying the data using the same graphical interface.

2.3.3 The Object-Oriented Approach

The **O-O approach** absorbs the ideas of semantic data modelling and fixes them into a structure within which both the behavioural and structural components of an application can be described at once. The origins of the O-O approach are scattered among the areas of programming languages [Dahl and Nygard, 1966], database systems [Smith and Smith, 1977] and artificial intelligence [Hewitt *et al*, 1973]. Good surveys of the approach include, for instance, [Stefic and Bobrow, 1985], [Bancilhon, 1988] and [Meyer, 1988].

According to Meyer's definition, O-O design is the construction of software systems as structured collections of abstract data type implementations [Meyer, 1988]. This reveals the strengths and weaknesses of the approach simultaneously. The strengths come from its highly modular construction, while the weaknesses are due to the limitation of the structure of abstract data types. Just as there are information structures which do not readily lend themselves to modelling by relations, so there are some which are not best modelled by ADTs.

The main characteristics of an O-O system include the following:

- **Object identity** - the attributes of a particular object are grouped together; a unique system-defined identifier is given to each object.

- **Referencing** - an object may be referenced from any other object via its identity, and any alteration of the object will be visible to these references.

- **Classification** - data values with common structure and behaviour are classified into sets which are referred to as classes.

- **Encapsulation** - the descriptions of the data structure and behavioural aspects of a class are grouped together. The data structure is hidden, and the only way of accessing an object is via a set of operations (methods) which capture the object's behaviour and which are available publicly.

- **Sub-typing** - the ability to describe one class as being a more specialised form of one or more other classes.

- **Inheritance** - both the properties and the behaviour of super-classes are inherited by each of their sub-classes; they can be further refined in the sub-class.

- **Overriding** - the ability to replace inherited definitions by sub-class specific ones.

- **Deferred binding** - the ability to refer to the operations of an object, knowing that at run-time its class will determine which version of the operations will be used.

Some relevant examples of the approach will now be given.

Simula 67 [Dahl and Nygard, 1966], developed by Dahl and Nygard as a language for discrete event simulation, demonstrated the properties which are later associated with O-O languages. Simula 67, being an extension of Algol, has many features, including the class constructor, which was first seen in this language.

Smalltalk [Goldberg and Robson, 1983], developed at Xerox by a group led by Kay, Goldberg and Ingalls, used the same concepts as those in a dynamically typed system. A drawback of Smalltalk is lack of static type checking, so that type errors could not be detected at the earliest possible time.

The important contribution of Smalltalk, however, is to frame everything in terms of objects, which brings significant conceptual simplicity. It has also been implemented in an interpretive way that makes the class hierarchy open for browsing and modification at run-time. This greatly eases program debugging. In addition, the language contains the notion of a meta-class, by which it becomes possible to describe class operations in the same way as instance operations.

C++, designed by Bjorne Stroustrup of AT&T [Stroustrup, 1984], adds the notion of classes to C. The language provides complete encapsulation, although some of the operations in the interface may be declared to be friend operations. That is, the operations take the relevant object an extra parameter. C++ also provides a single inheritance hierarchy and virtual operations.

Objective C, being a sort of combination of C and Smalltalk, was proposed by Brad Cox [Cox, 1986]. This language also offers polymorphism and dynamic binding. In objective C all complex objects are declared to be of the same type, *ID*.

E [Richardson and Carey, 1987] was extended from C++. E adds a supplementary sort of class, referred to as the **dbclass**, with which implementation details, buffering and pointer control can be added. E attaches persistence to C++ by a special sort of class referred to as a **file**, and also a notion of generic classes derived from **CLU** [Liscov *et al*, 1977].

Eiffel [Meyer 1988] attempts to include both the best features of the above languages and modern concepts of software engineering. Eiffel, incorporating the typed class world of Simula within a much simpler architecture similar to Smalltalk, possesses the following characteristics:

- **Strong static typing**. Eiffel indicates that strong typing and O-O programming can be combined elegantly.

- **Assertions**. While a class may have invariants specified, an operation may have pre- and post-conditions specified. The involvement of assertions helps to ensure the correctness of class descriptions.

- **Exceptions**. If an operation fails then a **retry** clause will be executed. If this fails again or does not appear then the exception will be transmitted to the calling operation.

- **Genericity**. Classes with type parameters may be specified, such as STACK OF [T], denoting stacks of arbitrary type. Such classes should be instantiated by producing a type in place of the type parameter.

- **Multiple inheritance** with name clashes resolved by renaming.

- **Dynamic binding** and **feature overriding**. A method may be respecified in a sub-class and the implementation of the method for a given object will be selected at run-time.

- **Deferred classes**. The inheritance mechanism is enhanced to include a special form of the subtyping relationship. Some of the methods of a given

class may be specified to be deferred, meaning that implementations of this method will only appear in subclasses. Such classes, however, may not have direct instances.

Although many superficial weaknesses of O-O languages have been avoided, the limitations upon the ways in which application behaviour can be expressed is still apparent. In any O-O system, all behaviour must be expressed as method code and this can have serious effects on the directness of the modelling power.

2.3.4 The Components of Data Models

All of the data models discussed are similar in that they all provide a set of constructs each of which can be instantiated many times to create schema elements. The constructs are of four kinds:

1) types to hold values - such types may describe sets of values which are atomic or composite; stored data or derived; objects (i.e. have object identity) or not;

2) relationships between those types;

3) constraints on the ways in which the types can be populated or combined;

4) behavioural constructs which describe some aspect of how the values are used.

It is clear that among all the models discussed, at least the constructs of the first two of these kinds are all drawn from a very small set of basic constructs [Hull and King, 1987]. Moreover, as will be discussed in Chapters 4 and 5, there is reason to believe that the same holds true for constraints and for behaviour. A coherent account of all data models could therefore be created by describing them in terms of the few basic constructs. This is one aim of the work presented in this thesis.

2.3.5 Configuring Data Modelling

The reason for the wide variety of data models surveyed is that they are each suitable for particular applications, tasks or user groups. It would therefore be of value for a DBMS to support a variety of data models at the external level. However, just as for the

user interface, repetitive coding of multiple data models would be a high cost. Instead, a configurable approach is indicated.

The approach which the current work takes is to create a toolkit of modelling primitives out of which a data model can be configured. Just as for the user interface, this should be an appropriate approach, which will be described in detail in Chapter 3.

2.4 Federated Databases

Each database system is usually intended for a particular class of data processing tasks. In an organisation which involves multiple departments and multiple database systems, although the database systems within a department may be based on the same data model and the same data language, the systems throughout the departments of the organisation may well be based on different data models and different data languages. Thus even within organisations there is likely to be a need to manage a heterogeneous set of database systems. If information is to be shared between a number of organisations, clearly the need is even more pressing.

Federated database systems [Hsiao, 1992] are groups of heterogeneous database systems. Each of these database systems usually has its respective data model and data language, being supported by distinctive computer hardware, system software, and professional personnel. They are efficient in their own applications and effective in upholding their respective integrity constraints and security requirements. Federated database systems aim to provide data sharing and resource consolidation without violating the autonomy of individual database systems.

2.4.1 The Characteristics of Federated Database Systems

Generally speaking, federated database systems possess the following characteristics [Hsiao, 1992]:

1) **Multimodel** and **multilingual** support. Federated database systems encompass a number of databases in different data models and execute transactions written in various languages.

2) **Transparent access**. A user is able to access each of the heterogeneous databases as if it were the user's own database. In other words, a user does not need to understand the data model of a foreign database, nor does the user need to write transactions in the data language supported by a foreign database system.

3) **Local autonomy**. The owner of each of the heterogeneous databases shares the database with others without compromising the owner's integrity constraints, application specificities, and security requirements. In other words, despite multiple accesses and manipulations made by other users, the autonomy of each database is maintained.

4) Connection of different hardware platforms including the possible use of multiple special purpose **database machines**.

5) **Consolidation** of the various component databases into a single usable system with **concurrency control**, supporting the normal database functionalities.

2.4.2 Data Sharing Approaches

There are two basic approaches towards data sharing among federated, heterogeneous databases. These are (i) **database conversion**, and (ii) **schema transformation** and **transaction translation**, respectively. The former approach creates copies of the data by transforming the structure between systems. The latter merely transforms the schema and queries but leaves the data unchanged. These two approaches will now be discussed in turn.

2.4.3 Database Conversion

To make a database available to a user who is not familiar with the relevant data model, a theoretical but impractical solution is the database conversion approach, which directly converts the database into an equivalent database in the data model the user is familiar with. For example, if a user who is familiar with relational databases wishes to access a hierarchical database, a hierarchical-relational database converter must be employed. This must be done to access the hierarchical database system for the intended

database, to make a copy of the hierarchical database, to convert the copy into an equivalent relational database, to load the converted database into a relational database system, and to allow the relational database user to access the converted database by way of the relational database system. If the user wishes to update as well as to retrieve the data in the database, then upon the manipulation being completed, a relational-hierarchical database converter would in turn be employed to convert the updated relational database back into an equivalent hierarchical database, which will replace the original database in the hierarchical database system. This process, however, could be horribly inefficient. The main ideas regarding database conversion approach are illustrated in Figure 2.7.

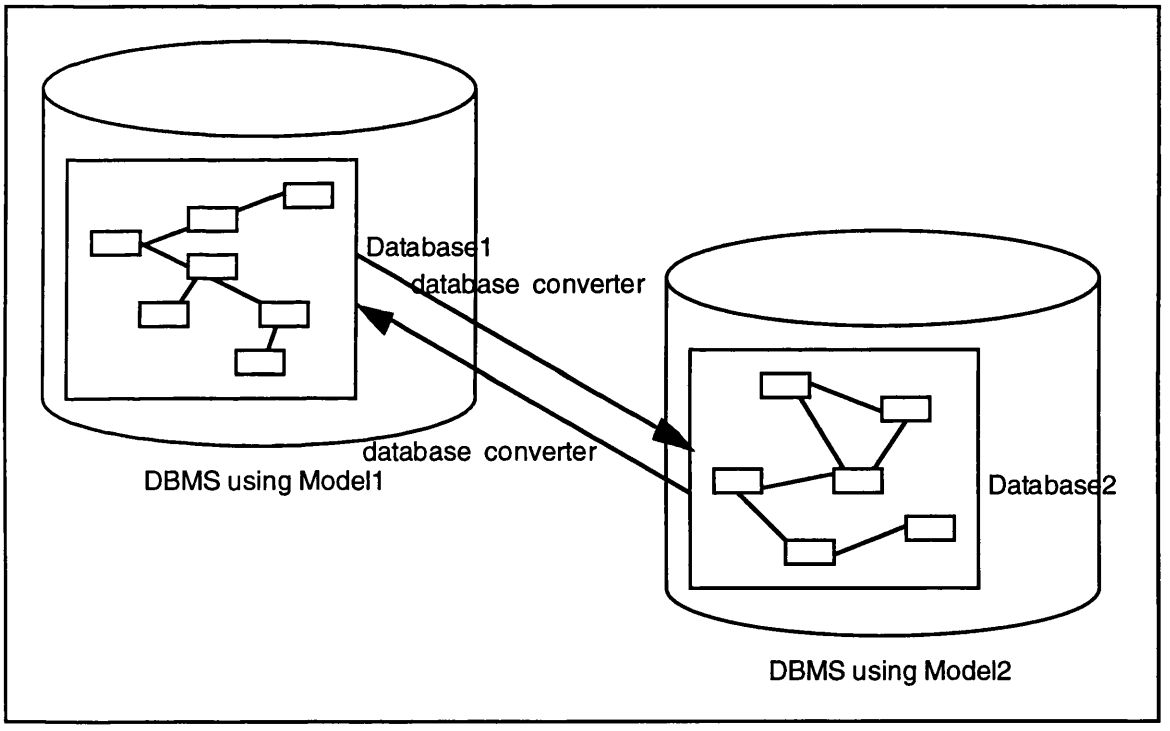

Figure 2.7 Database Conversion Approach

The database conversion approach provides transparent access to heterogeneous databases in the federation if a sufficient number of converters are provided, each of which converts databases from one model to another. Given *n* models, *n(n-1)* converters will be needed in order to provide full transparency. The conversion of a database in a certain data model to an equivalent database in a semantically richer data model is straightforward, since semantic constructs of the former are likely to be subsumed by semantic constructs of the latter. In addition, research results have proved that it is possible to convert a database in a semantically rich data model to an equivalent database in a semantically poorer data model, that is, equivalent databases in semantically poor data models can also be created for databases in semantically rich models. In this case, however, as has been pointed out for

the relational model [Teorey, 1986], it may be difficult for the user to understand the original intention of each part of the converted database.

An example of a federated database system with the database conversion approach is illustrated in Figure 2.8. The system, which permits full transparent access and manipulation, consists of three data models.

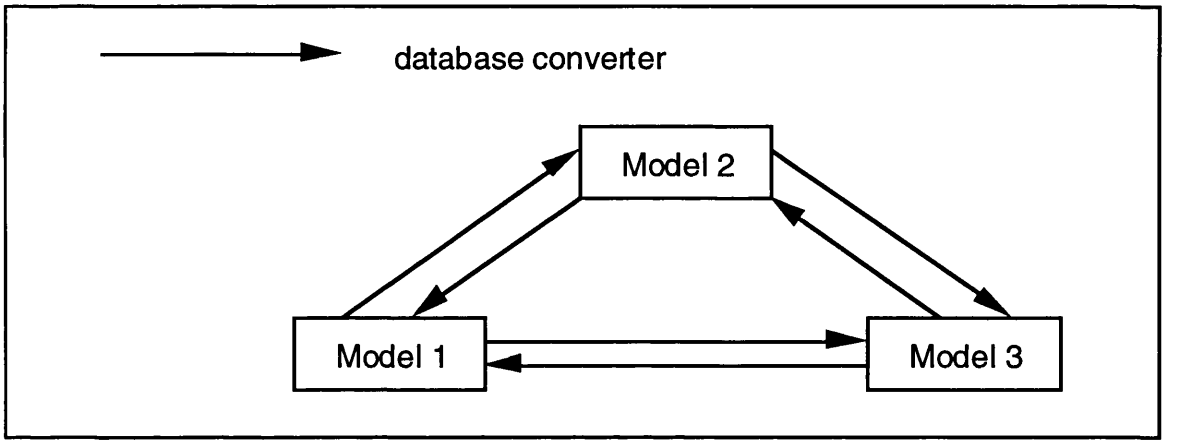

Figure 2.8 Database Systems with Database Conversion Approach

The semantic equivalence of the database in one data model and its converted version in another data model is determined by the converter. The semantics of one data model may be markedly different from the semantics of the other data model. If one desires to stay in one's own semantics, without learning the semantics of the other data model, one will have to accept the semantic equivalence provided by the database converter on the semantics of the other database in the light of one's own data model and data semantics.

Database converters generate multiple copies of the same database as required. The autonomy of the original database is safeguarded by the system which supports the database, but copies in different data models are not supported by the same system. It is therefore difficult if not impossible to enforce the same integrity constraints, application specificities and security requirements of the database on its copies. In addition, simultaneous updates by different database systems on copies of the same database are difficult to coordinate and control.

A way to minimise this difficulty is to ask a multimodel and multilingual professional, who understands integrity constraints, application specificities, and security requirements of the database in its native data model and native data language, to specify equivalent constraints, specificities, and requirements on its copies in foreign data models

and foreign data languages. The viability of local autonomies of federated databases thus rests with the availability of such an individual.

What is really required to improve this approach is some form of global management of the features of the individual databases. In such a system, local integrity constraints, application specificities, and security requirements would be made to hold on copies and thus be maintained automatically.

2.4.4 Schema Transformation and Transaction Translation

The schema transformation and transaction translation approach changes the access mechanisms instead of changing the structure of the data. This approach of data sharing among federated databases is more efficient than the database conversion approach.

In this approach, additional equivalent schemata for a database are generated as required based on data models other than the original data model, while any transaction written in a data language other than the original data language has to be translated back into an equivalent transaction in the original data language for execution. The main ideas concerning schema transformation and transaction translation approach are illustrated in Figure 2.9.

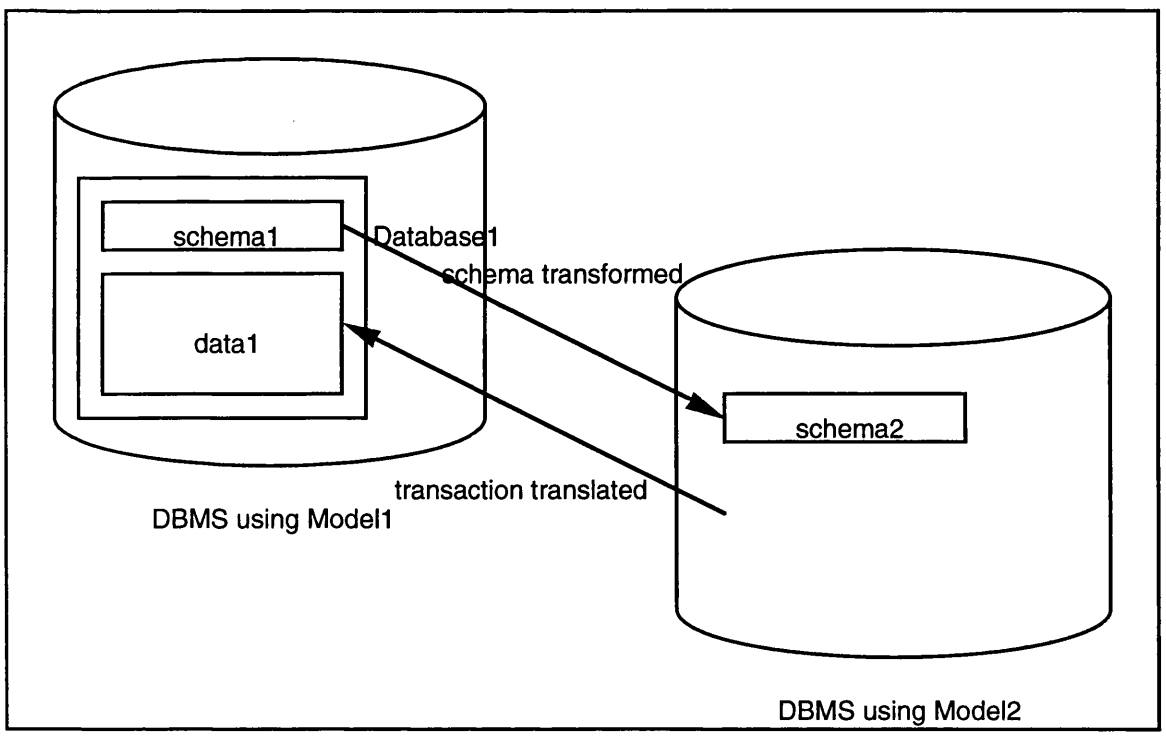

Figure 2.9 Schema Transformation Approach

Both capabilities of schema transformation and transaction translation are essential in this approach. The former enables a database to be viewed in different data models, while the latter permits a database to be manipulated in different data languages.

Comparing the schema transformation and transaction translation approach with the database conversion approach, two major differences are as follows:

1) In the conversion approach multiple copies of a database are produced and maintained, whereas in the transformation and translation approach only multiple schemata will be generated. The apparent existence of multiple copies is supported by the maintenance of virtual derived versions.

2) In the conversion approach suitable software is developed for the conversion of a database in one data model to an equivalent database in another data model, whereas in the transformation and translation approach proper software is developed for schema transformation and transaction translation among various data models and data languages.

There are four different **system architectures** to facilitate data sharing among federated, heterogeneous database systems using the schema transformation and transaction translation approach. These are:

- the single-model-and-language-to-single-model-and-language mapping (Single-ML-to-Single-ML mapping),

- the multiple-models-and-languages-to-single-model-and-language mapping (Multiple-MLs-to-Single-ML mapping),

- the single-model-and-language-to-multiple-models-and-languages mapping (Single-ML-to-Multiple-MLs mapping), and

- the multiple-models-and-languages-to-multiple-models-and-languages mapping (Multiple-MLs-to-Multiple-MLs mapping).

An example of a federated database system with the architecture of **Single-ML-to-Single-ML mapping** with the transformation and translation approach is

illustrated in Figure 2.10. The system consists of three data models and offers full transparent access and manipulation capability.

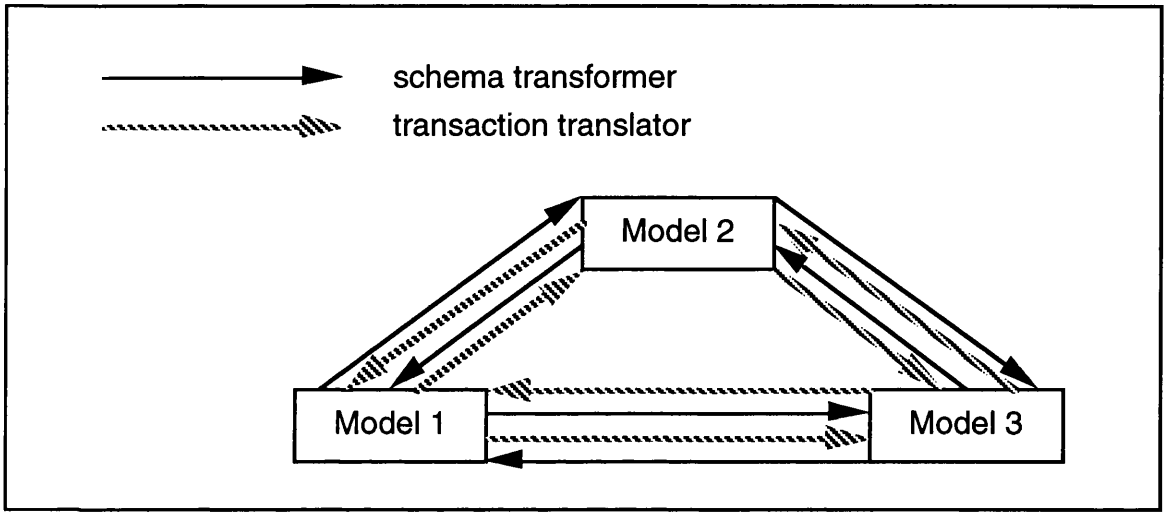

Figure 2.10 Single ML to Single ML Mapping

In order to achieve the maximum data sharing in such an architecture, if the heterogeneity of the federated databases is *n*, then *n(n-1)* schema transformers and *n(n-1)* transaction translators will be required. As long as this many transformers and translators are properly formulated and maintained, no user of the system will be faced with any difficulty when accessing and manipulating the data. In addition, since all changes to a database are made locally by the local database system, local autonomy can be upheld easily and effectively by the access and concurrency control mechanism of the local database system.

A disadvantage of this architecture is that the number of schema transformers and transaction translators in the federated system will become unacceptably large if the number of data models involved in the systems is itself large.

An example of a federated database system with the architecture of **Multiple-MLs-to-Single-ML mapping** with the transformation and translation approach is illustrated in Figure 2.11. The system also consists of three data models and offer full transparent access and manipulation capability.

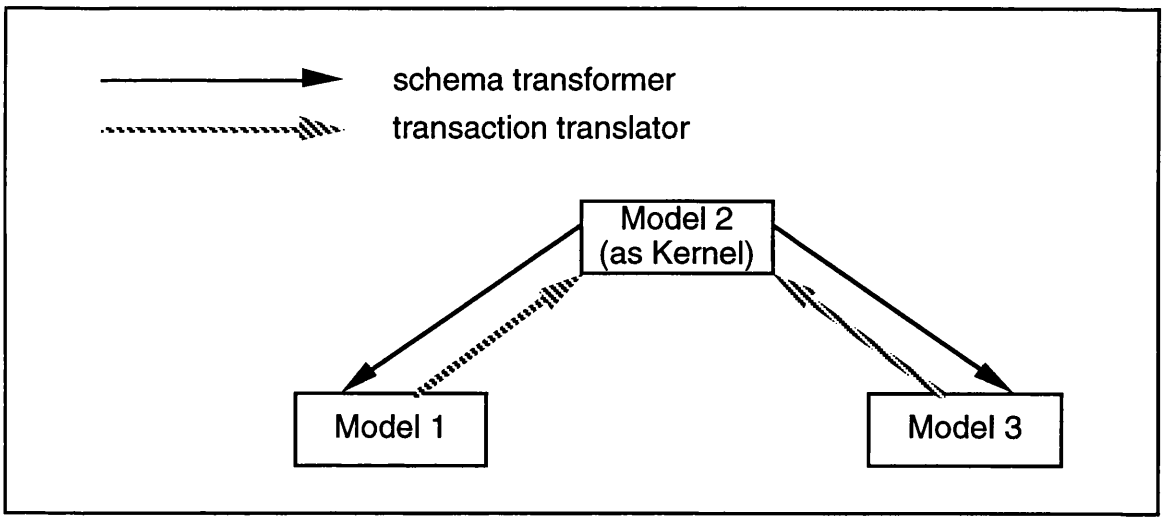

Figure 2.11 Multiple MLs to Single ML Mapping

In this architecture, a data model and data language pair acts as kernel model and kernel language, upon which the sole genuine database system in the 'federation' is based. The other data models just provide alternatives for the users to access and manipulate the data in the system. As illustrated, if the heterogeneity of the system is *n*, then only *n-1* schema transformers, and *n-1* transaction translators will be needed to provide fully transparent access and manipulation. Each transformer transforms a schema in the kernel data model to a schema in a particular non-kernel data model, whereas each translator translates a transaction written in the data language relating to a particular non-kernel data model to a transaction in the data language relating to the kernel data model. Owing to similar reasons to those mentioned for the Single-ML-to-Single-ML mapping architecture, if the relevant constraints, specifications, and requirements written in a non-kernel data language can be faithfully translated into equivalent constraints, specifications and requirements in the kernel data language of the system, then the local autonomy can be upheld by the access and concurrency control mechanism of the kernel database system.

An advantage of this architecture is that only a relatively small number of schema transformers and transaction translators are required in order to enable full transparency of data access and manipulation, while a disadvantage is that existing databases based on other database models have to be converted into equivalent databases in the kernel data model.

An example of a federated database system in the architecture of **Single-ML-to-Multiple-MLs mapping** with the transformation and translation approach is illustrated in Figure 2.12.

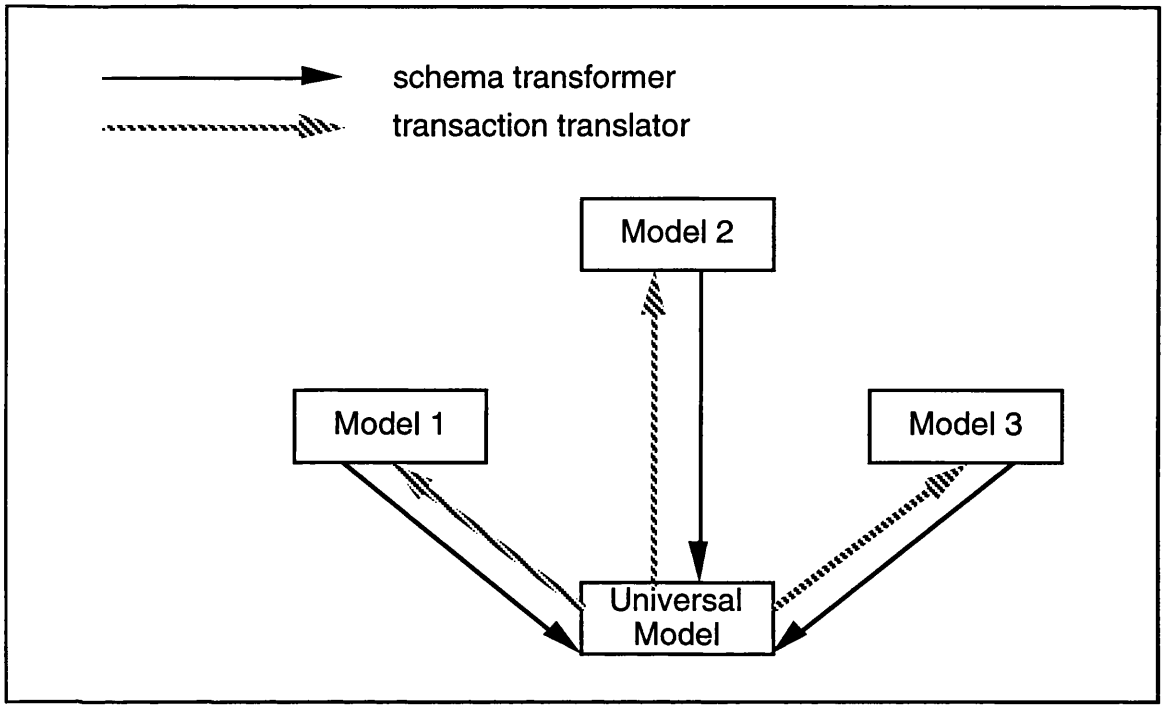

Figure 2.12 Single ML to Multiple MLs Mapping

In this architecture, a universal data model and a universal data language are created artificially and provided to all users for transparent access to and manipulation of the heterogeneous databases in the federation. With a user's familiarisation with both the universal model and the language, the federated database system allows the user to view any database in the federation as if it were in the universal data model and to manipulate any database by writing transactions in the universal data language. The data model and the data language are called universal because they are the only pair of model and language that provides transparent access to every database in the federation.

A user must learn the universal data model and universal data language to enjoy the benefit of transparent access to the databases, the models of which the user is not familiar with. This time, provided the heterogeneity of the federated database is *n*, then *n* schema transformers, and *n* transaction translators are required for transparency of access and manipulation in this way.

Again, since all changes to a database are made locally by the local database system, the local autonomy can be upheld by the access and concurrency control mechanism of the local database system.

Finally, Figure 2.13 shows an example of a federated database system, which consist of three data models and provide full transparency of access and manipulation, in

the architecture of **Multiple-MLs-to-Multiple-MLs mapping** with the schema transformation and transaction translation approach.

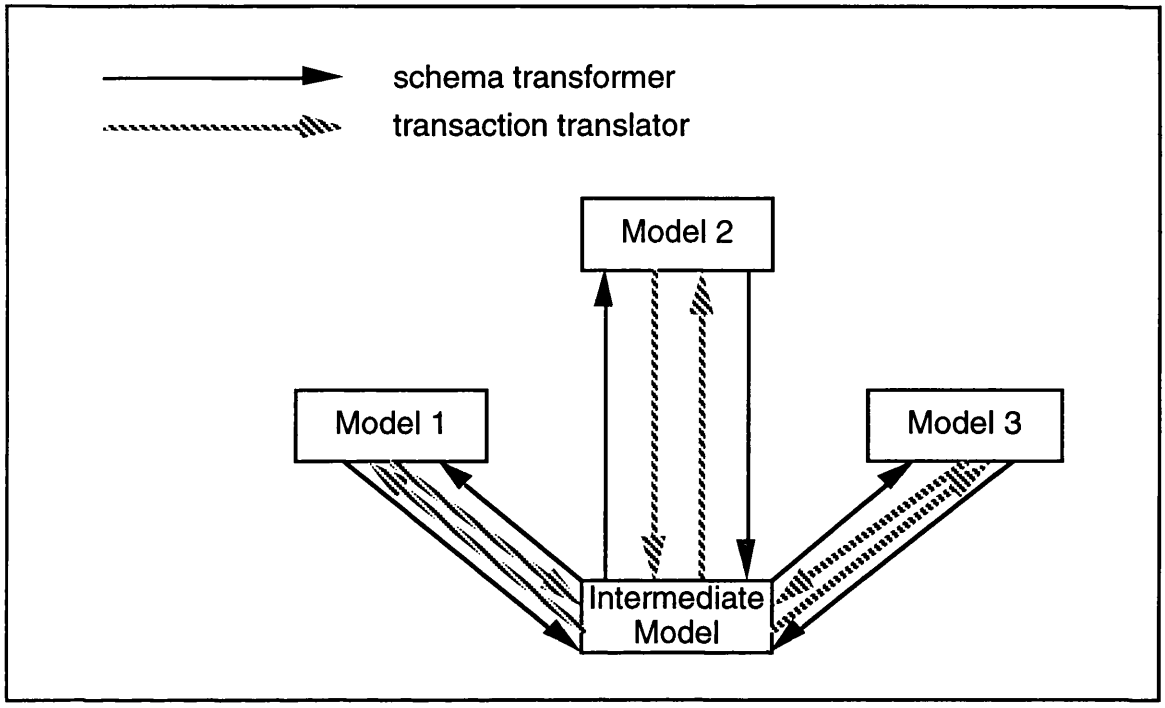

Figure 2.13 Multiple MLs to Multiple MLs Mapping

In this architecture, a special pair of data model and data language is created to act as intermediate model and intermediate language. The Multiple-MLs-to-Multiple-MLs mapping therefore consists of two consecutive stages, the first stage being a Multiple-MLs-to-Single-ML mapping and the second stage being a Single-ML-to-Multiple-MLs mapping. This architecture requires $2n$ schema transformers and $2n$ transaction translators to achieve the maximum data sharing, given that the heterogeneity of the federation is n.

The intermediate data model and data language used in this architecture are just conceptual and virtual, being sufficiently rich in semantics to subsume the others. Therefore, the intermediate data model and language are similar to the universal data model and language in the Single-ML-to-Multiple-MLs mapping architecture, but quite different from the kernel data model and language in the Multiple-MLs-to-Single-ML mapping architecture.

A goal for using an intermediate data model and data language is access and concurrency control. Since transparent accesses can only take place after the second stage of the mapping process, the mapping can validate the access request against the integrity constraints, application specificities, and security requirements of the given database at the end of the first stage of the mapping process. All requests, constraints, specificities, and

requirements are translated from the users' data model and language into the intermediate data model and language to facilitate the validation. As long as the transformation and translation preserve the semantics of the users' requests, constraints, specificities, and requirements in the semantics of the intermediate data model and language, the role of the intermediate data model and language in access and concurrency controls of the two-stage process is important and necessary.

In the work presented in this thesis, the existence of a universal data model promises support for multiple data models without the requirement of translators between each pair of models (requiring $n(n-1)$ translators), instead each model will have translators between it and the universal model (requiring only $2n$ translators to be built). Translation between two models then require two steps with the universal model appearing as an intermediate model.

2.5 Conclusions

A DBMS is a software system that supports data intensive applications. In the early stage, such applications were relatively easy to program as the structure of data used was kept simple and straightforward. As database technology progressed, however, not only have the requirements arising from the traditional applications increased, but new assistance is also demanded for more general and complicated application systems. These application systems automate complex information processing systems and support a broad spectrum of activities.

A distinguishing characteristic of data intensive applications is the representation of a large number of interrelated entities of the application realm. In the database system, each entity will be represented by a value. The term 'value' here is used in its most general sense, implying any valid data item of a particular system; that is, a value may be a simple one, such as a string or an integer, or a complex one, such as a record or an object.

For example, a particular person might be represented by the group of the string 'Jean' and the string '50 Grant Street'. Groups of values, which themselves are values, can also be used to represent relationships between entities. If a particular person entity is represented by the value p, and a particular book entity is represented by the value b, then the fact of that person borrowing that book may be represented by the group (p, d).

At the end-user level, there are facilities for describing the nature of the values which arise in a particular application and other facilities for manipulating and retrieving these facilities. Each of these facilities is provided in the context of the data model in which the user conceives the data description. The system will then map this description down into more physical models against which the DBMS is written.

Given the variety of users that a DBMS is expected to support, there is a clear need to support multiple data models, varying in their degree of detail and appropriateness to particular applications. One particular kind of database system in which the support of multiple models is vital is a federated DBMS. Here DBMS built on different data models are required to share data.

The next three chapters describe an approach to the consistent and coherent provision of multiple data models. Chapter 3 describes the approach in the context of structural aspects of the data, while Chapters 4 and 5 describe the management of constraints and behaviour respectively. Later chapters deal with the formalisation and implementation of these ideas.

3 Configurable Data Modelling System

This chapter supplies a systematic exposition of the concepts underlying a configurable data modelling system (CDMS).

Section 3.1 explains the motivation and feasibility. Section 3.2 analyses the data modelling process and methodology. Section 3.3 proposes the CDMS structure. Section 3.4 introduces data modelling primitives and data model configuration. Sections 3.5 and 3.6 contain, respectively, the approaches dealing with constraints and behaviour in the context of the CDMS. Section 3.7 describes interaction elements and dialogue primitives. Section 3.8 summarises the CDMS functionality. Section 3.9 describes some related work.

3.1 Motivation and Feasibility

Now that a number of semantic data models have been proposed, data abstraction can now be achieved directly and intuitively based on these user-friendly models. Nevertheless, a conventional database management system (DBMS) is usually built upon few specific data models, with few fixed interfaces, hence its usability is inevitably restricted. Apart from this, each implementation of a specific pair of model and interface for such a DBMS is traditionally a separate task. This leads to repetitive programming efforts, which incur labour wastage.

A **user interface management system** (UIMS) treats an interface as a structured group of **dialogue primitives**, which can be combined flexibly to form appropriate interfaces according to the designers' choices. By using the UIMS idea more precisely in the context of a DBMS, a range of interfaces can be built to the same data model [King and Novak, 1989]. However, the UIMS idea may be further extended to

support the construction of multiple data models out of generic **data modelling primitives**. Thus the varying needs of different groups of users can be satisfied.

Judicious analysis of data modelling shows that the various semantic data models are actually based on a small set of constructs - base values, complex objects, connections, constraints on their utilisation and combination, as well as data definition and data manipulation operations [Hull and King, 1987]. In essence, this variety of semantic data models is based on a small number of common concepts, which can be regarded as primitive components of the data models in just the same way as interaction elements are primitive to user interfaces.

Therefore, given appropriate sets of **data modelling primitives** and **dialogue primitives**, as well as appropriate construction regulations, data model implementors would be able to construct data models that suit various applications, and interface implementors would be able to construct appropriate interfaces to those data models. This would satisfy various users, without being restricted by the models and interfaces actually existing in a particular system or requiring the continual re-programming of similar functionality.

Such a system is referred to as a **configurable data modelling system**.

3.2 Data Modelling Process and Methodology

This section articulates the relationship among database, data schema, and data model, out of which arises the notion of a global data model.

3.2.1 A Database as an Instance of a Data Schema

To use DBMS technology, the structure of a miniworld is described in terms of a data model and is then referred to as a **data schema**. The schema is made up of a set of elements each of which is an instance of one of the constructs in a data model. The schema is then employed as the framework for the values which make up the relevant **database**.

To illustrate this, Figure 3.1 shows an ER schema, in which the framework of a library system is described in terms of related entities with attributes. A number of **base classes**, *name*, *id*, *sex*, *age*, *house*, *street*, *city*, *title* and *number*; a number of **complex**

classes, *person*, *book*, *address* and *loan*; and a number of **connection classes**, *person's name*, *person's id*, *person's sex*, *person's age*, *person's address*, *title of book*, *number of book*, *address-house*, *address-street*, *address-city*, *loan-person* and *loan-book* are involved in the schema.

This schema can be used to organise the basic information regarding members, books and loans relating members and books in a particular library. Broadly speaking, it can be used to organise the aforementioned sorts of information for any library into a database.

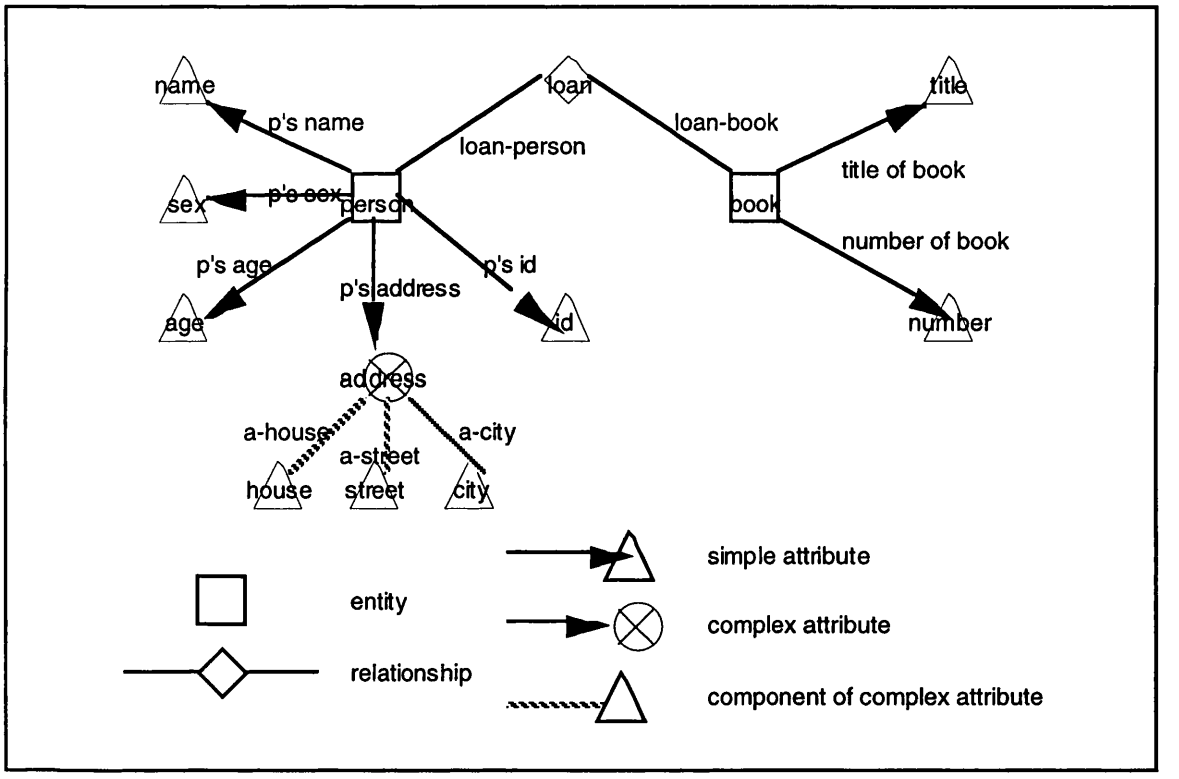

Figure 3.1 A Library Schema

A database that is made up of instances of the schema components of Figure 3.1 is shown in Figure 3.2, where, for example, *'Jean'* and *'George'* are instances of *name*, *true* and *false* (representing female and male respectively) are instances of *sex*. In addition, there are two instances of *person*, one instance of *address*, three instances of *loan*, and so on. The information expressed by the database is that both a girl named 'Jean', with id 1001, aged 18, living at 50 Grant Street, Glasgow and a boy named 'George', with id 1002, aged 20, living at the same address are members of the library; the library owns books 'Database' numbered 5001, 'Programming' numbered 5002 and 'Bridge' numbered 5003; 'Database' and 'Programming' are currently borrowed by Jean, and 'Bridge' by George.

In order to make the correspondence between an instance and the class it belongs to even clearer, the layout of Figure 3.2 deliberately resembles that of Figure 3.1. Thus, a number of **base instances**, a number of **complex instances** and a number of **connection instances** are contained in the database, which is itself an instance of the data schema.

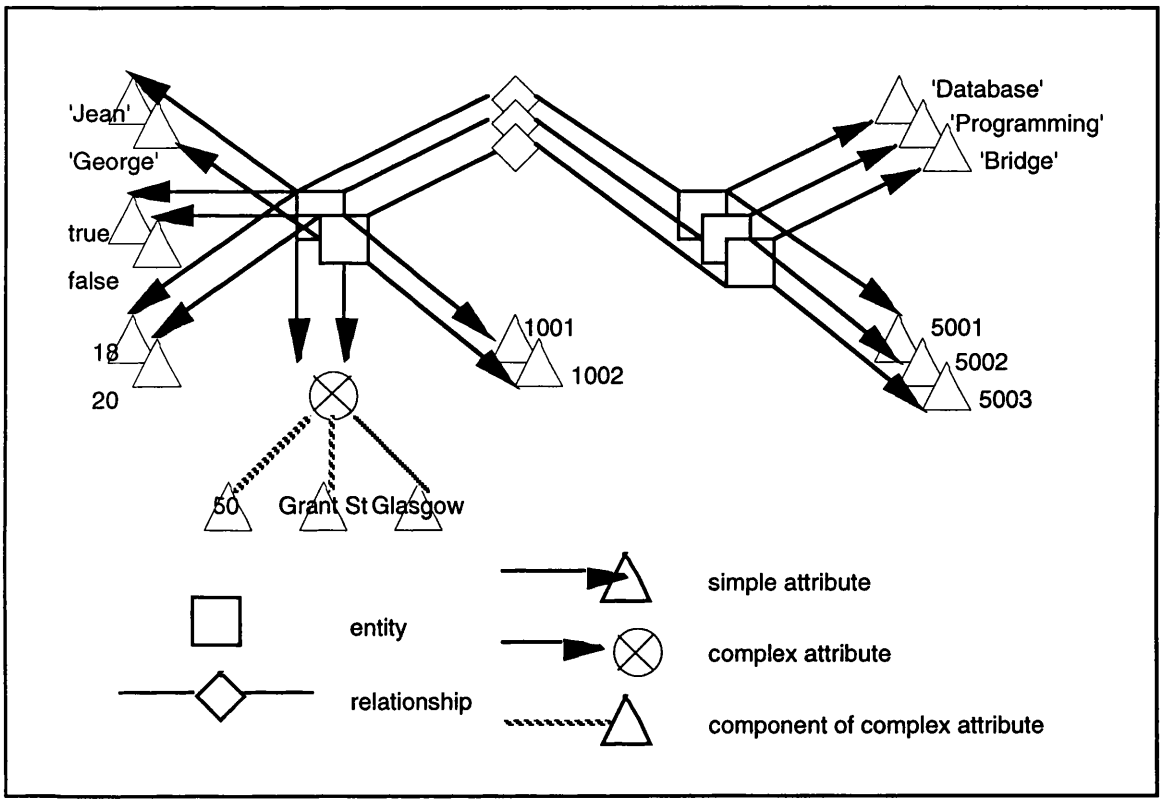

Figure 3.2 A Library Database

Figures 3.1 and 3.2 are usually referred to as the **intension** and the **extension** graphs of the database. The information which is available for manipulating the database can be thought of as the combination of these two graphs with an instantiation link between nodes in the intension graph and nodes in the extension graph. Thus each element of Figure 3.2 is an instance of an element of Figure 3.1.

Indeed, the database as a whole can be viewed as an instance of the schema taken as a whole - each value in the database being an instance of an element of the schema. The database represents one of the many possible states which correspond to the schema. A database is thus an instance of a schema.

3.2.2 A Data Schema as an Instance of a Data Model

The relationship between a data model and a data schema supported by the model parallels the relationship between a data schema and a database supported by the schema. A semantic data model defines sets of base types, complex types and connection types, of which base classes, complex classes and connection classes in data schemata are just instances.

A simplified ER model, which is sufficient to support the data schema represented in Figure 3.1 is illustrated in Figure 3.3. This model includes the **base type** *simple attribute*; **complex types** *entity*, *composite attribute*, *relationship*; and **connection types** *attributing*, *consisting*, *relating*. In the figure, a named triangle denotes a base type, a named symbol other than a triangle denotes a complex type, and a named edge style denotes a connection type. More precisely, each directed edge denotes a sub-connection type, each instance of which connects two instances of the relevant base/complex types in a particular data schema supported by the data model. This representation is consistent with the data schema representation of Figure 3.1.

Thus in the previous example, base classes *name*, *id*, *sex*, *age*, *house*, *street*, *city*, *title* and *number* are instances of the base type *simple attribute*; complex classes *person* and *book* are instances of the complex type *entity*; the complex class *address* is an instance of the complex type *composite attribute*; the complex class *loan* is an instance of the complex type *relationship*; connection classes *person's name* etc are instances of sub-connection type *attributing* (from *entity* to *simple attribute*); the connection class *person's address* is an instance of sub-connection type *attributing* (from *entity* to *composite attribute*); connection classes *address-house* etc are instances of sub-connection type *consisting* (from *complex attribute* to *simple attribute*); and connection classes *loan-person* and *loan-book* are instances of connection type *relating*.

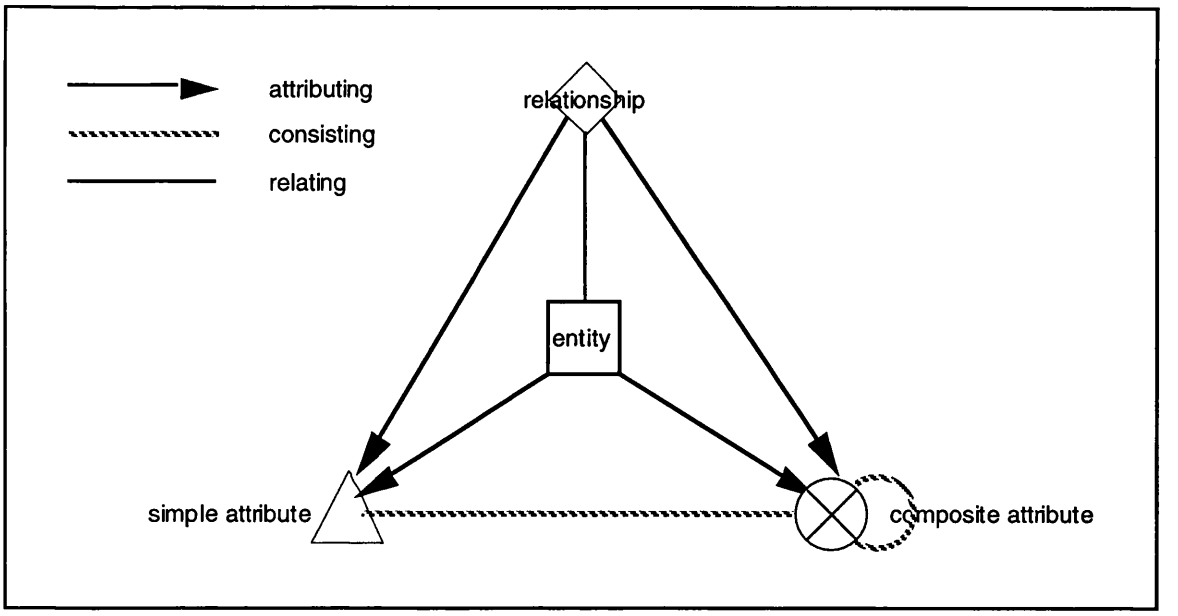

Figure 3.3 A Simplified ER Model

Just as a database occurrence is an instance of a data schema, a data schema is an instance of a data model. Many schemata may be created, modified and instantiated against one model, just as a number of databases may be established and maintained against one schema.

Various semantic data models are common in offering types to be specialised in order to generate classes as schemata are created. The difference between two particular models is intrinsically the difference between the number and nature of concrete types they offer. This difference allows various points of view to determine schema definition, and various extents of detail to be captured.

3.2.3 The Global Model as Generalisation of Data Models

A CDMS is designed to be a system which coherently and consistently supports a variety of data models. Thus a CDMS expects to house a majority of the semantic data models found in the literature in a single environment in such a way that data can potentially be shared among them. In order to achieve this, some underlying structure is required. The structure chosen is intended to provide a highly abstract, relatively unconstrained generic model of which all of the supported data models are instances. Thus this generic model, referred to as the **global data model**, stands in the same relation to the data models as each data model does to the schemata supported by it.

The **meta-base type**, **meta-complex type** and **meta-connection types** represent the main concepts of the global data model. Meta-types epitomise various types existing in various semantic data models, thus as a meta-type is specialised, a type will be created as an instance of it.

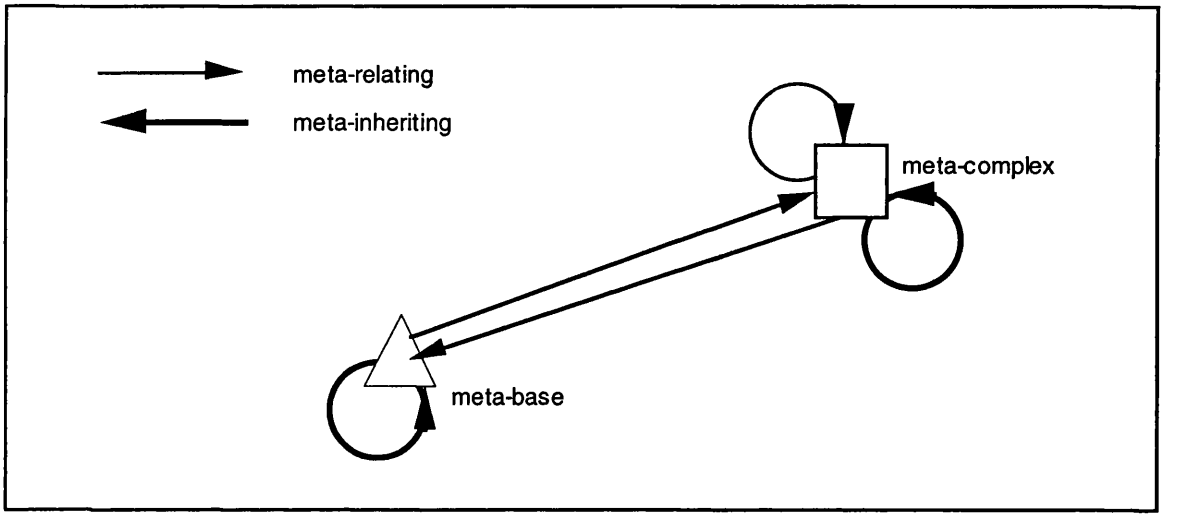

Figure 3.4 The Global Data Model

Figure 3.4 illustrates the global data model used to build the CDMS in this research. The global data model defines the sole meta-base type *meta-base*; the sole meta-complex type *meta-complex*; and meta-connection types *meta-relating* and *meta-inheriting*. In the figure, the triangle and the square denote *meta-base* and *meta-complex* respectively, while each named edge style denotes one of the two meta-connection types permitted by the global model - *meta-inheriting* describes an ISA relationship and *meta-relating* describes attribution and part-of relationships. This representation is consistent with the data schema representation illustrated in Figure 3.1, as well as consistent with the data model representation illustrated in Figure 3.3. The figure also shows the basic constraints which are built into the global model:

- inheriting can only link the same kind of value - thus base values cannot inherit from complex values and vice versa;

- there are no other connections between base types.

These are summarised in Figure 3.5.

Referring to the simplified ER model (Figure 3.3), which is an instance of the global data model, the base type *simple attribute* is specialised from *meta-base*, complex

types *entity*, *composite attribute* and *relationship* are specialised from *meta-complex*, and connection types *attributing*, *consisting* and *relating* are specialised from *meta-relating*.

relating types

from \ to	base type	complex type
base type	not allowed - a base type may not relate to a base type	allowed - a base type may relate to a complex type: there is reverse function from a base class to a complex class
complex type	allowed - a complex type may relate to a base type	allowed - a complex type may relate to a complex type

inheriting types

from \ to	base type	complex type
base type	allowed - a base type may inherit from a base type	not allowed - a base type may not inherit from a complex type
complex type	not allowed - a complex type may not inherit from a base type	allowed - a complex type may inherit from a complex type

Figure 3.5 Combination Ability of Connection Types

3.3 CDMS Structure

Based on the analysis in the previous section, a CDMS is built up using a **four level architecture**, which consists of the global level, model level, schema level and data level. In this architecture, a data model resides in the system as a consistent set of meta-metadata values in the same way that a data schema does as a consistent set of metadata values. The overall structure of the CDMS is therefore as illustrated in Figure 3.6. In the figure, the frameworks of real miniworlds *miniworld1* and *miniworld2* are represented, separately, by *Schema1* and *Schema2*, however, both the schemata are described by the same data model *Model1*

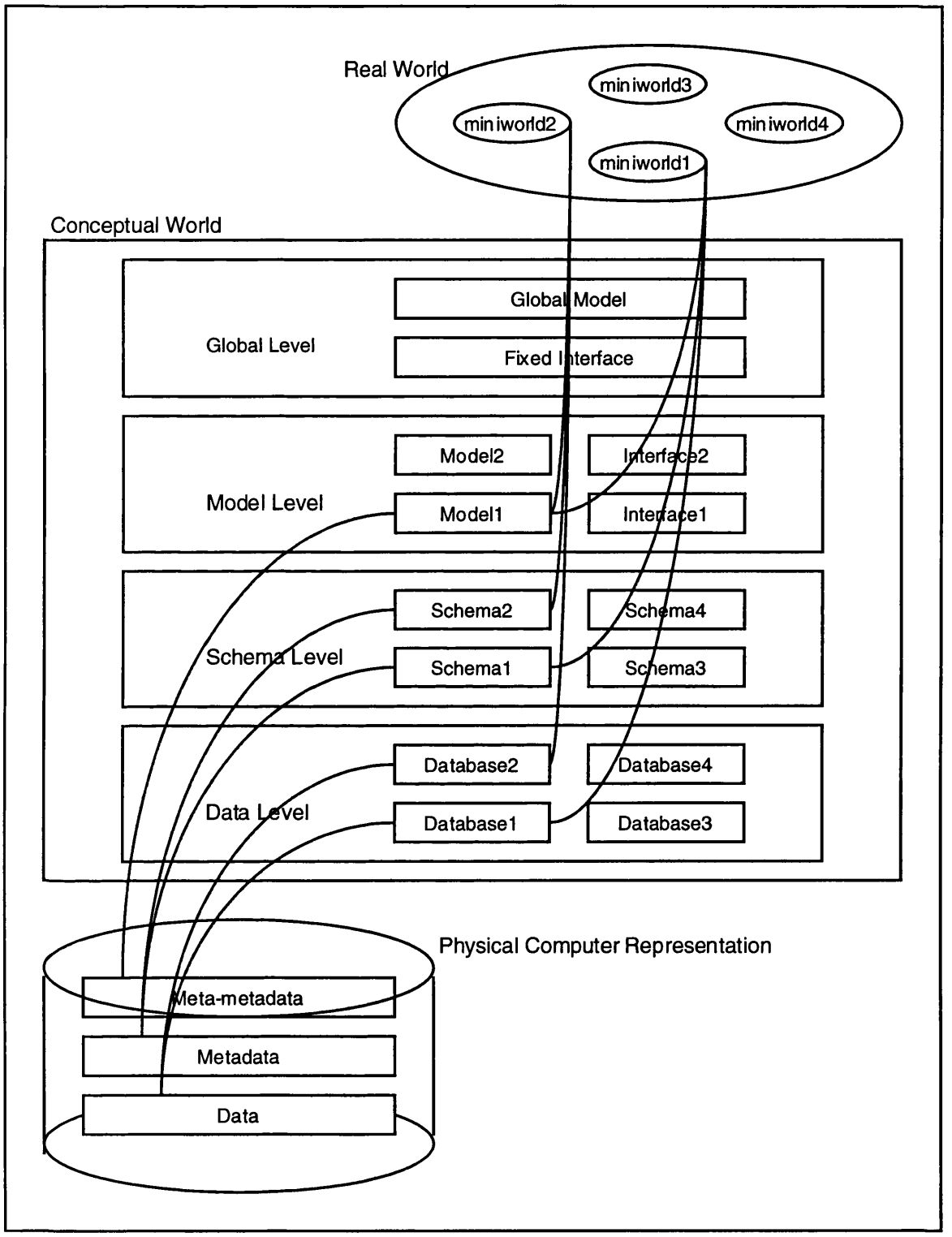

Figure 3.6 Overall Structure of the CDMS

The **global level** contains the sole global data model, the properties of which determine the fundamental characteristics and limitations of the CDMS. In other words, this level offers all generic data modelling primitives, which represent the most comprehensive concepts of the CDMS and from which components of data models can be

generated. An appropriate group of these specialised components constitute a specific data model.

The data models supported by the system will include most prominent semantic data models such as the Entity-Relationship Model (ER), the Semantic Data Model (SDM), the Functional Data Model (FDM) and IFO, as well as classical implementation data models such as the Relational Model, the Hierarchical Model and the Network Model, since these also have a partially conceptual aspect.

The global level also offers dialogue primitives, from which various interfaces to individual data models can be constructed. The interfaces supported by the system should include command languages, forms, graph-based interfaces and so on.

At the global level a database engineer (DBE) constructs various data models and interfaces as desired.

The **model level** contains all data models which have been constructed within the system. Each model is essentially a specialised hence restricted set of the components of the sole global data model. At this level a database administrator (DBA) defines and maintains various data schemata.

Each particular data model has at least one default user interface attached to it, while more interfaces may be added to meet various users' needs. In practice both model design and interface design are closely combined with each other. The default interface must be constructed at the same time as the relevant model is being defined, and this interface must not be removed unless the same model is removed. The interfaces are used to maintain schemata at the model level.

The **schema level** contains all data schemata which have been constructed. Each schema is a specialised set of the components of a certain data model. At this level a database end-user uses one of the interfaces which have been constructed to manipulate the relevant data.

The **data level** contains all databases which have been established. Each database can be considered a specialised set of the components of some data schema.

DBAs and database end-users interact with the CDMS via interfaces created by DBEs, while DBEs interact with the CDMS via the fixed interface attached to the global data model.

3.4 Modelling Primitives and Data Model Configuration

Data modelling primitives constitute the global data model. The global data model represents the highest abstraction of models, schemata and databases which can be supported within the CDMS, while the data modelling primitives represent the highest abstraction of all sorts of values which may exist in these models, schemata and databases. Data modelling primitives incorporate **meta-base type, meta-complex type, meta-connection types** and others.

The CDMS contains *meta-base* as the sole meta-base type. *Meta-base* represents the highest abstraction of all base values which are directly representable and alterable through human-computer interface, it can therefore be specialised as the base type *simple attribute* in the ER model, or *printable* in IFO.

The CDMS contains *meta-complex* as the sole meta-complex type. *Meta-complex* represents the highest abstraction of non-base objects, it can therefore be specialised as the complex types *entity*, *composite attribute* and *relationship* in the ER model, or *abstract*, *free*, *set* and *aggregate* in IFO. An *entity* in the ER model, an *abstract* or a *free* in IFO represents an object. A *set* in IFO represents a group of similar objects of its 'child', that is, *set* can be specialised as such a class that each instance of it is a set of instances of some other class. Types *composite attribute*, *relationship* in the ER model, and *aggregate* in IFO are, in essence, a Cartesian product over their 'children', thus, for example, *composite attribute* can be specialised as *address*, each instance of which consists of three instances, which are specialised from *house*, *street* and *city* respectively.

The CDMS incorporates *meta-relating* and *meta-inheriting* as meta-connection types. These can be specialised as meta-metadata values in the models to be defined, representing various sorts of connection in the real world. *Meta-relating* is used to describe various relationships among values/objects, while the basic meaning of *meta-inheriting* concerns duplicate values/objects. This semantics is reflected in Figures 3.4 and 3.5, by indicating combination patterns among the instances of meta-types as a model is constructed.

primitives	meta-metadata	metadata	data
meta-base	simple attribute	name	*'Jean'*
			'George'
		id	*1001*
			1002
		sex	*true*
			false
		age	*18*
			20
		house	*50*
		street	*'Grant Street'*
		city	*'Glasgow'*
		title	*'Database'*
			'Programming'
			'Bridge'
		number	*5001*
			5002
			5003
meta-complex	entity	person	person Jean
			person George
		book	book Database
			book Programming
			book Bridge
	composite attribute	address	address 50 Grant St, Glasgow
	relationship	loan	Jean borrowing Database
			Jean borrowing Programming
			George borrowing Bridge
meta-relating	attributing	person's name	Jean-*'Jean'*
			George-*'George'*
		person's id	Jean-*1001*
			George-*1002*
		person's sex	Jean-*true*
			George-*false*
		person's age	Jean-*18*
			George-*20*
		title of book	Database-*'Database'*
			Programming-*'Programming'*
			Bridge-*'Bridge'*
		number of book	Database-*5001*
			Programming-*5002*
			Bridge-*5003*
		person's address	Jean-50 Grant St, Glasgow
			George-50 Grant St, Glasgow
	consisting	address-house	50 Grant St, Glasgow-*50*
		address-street	50 Grant St, Glasgow-*'Grant Street'*
		address-city	50 Grant St, Glasgow-*'Glasgow'*
	relating	loan-person	Jean borrowing Database-Jean
			Jean borrowing Programming-Jean
			George borrowing Bridge-George
		loan-book	Jean borrowing Database-Database
			Jean borrowing Programming-Programming
			George borrowing Bridge-Bridge
meta-inheriting			

Figure 3.7 Specialisation of Generic Data Modelling Primitives

Figure 3.7 summarises the generic data modelling primitives, and gives an example of their specialisation routes in the context of an application of the CDMS. This example demonstrates the configuration of the Simplified Entity-Relationship Model.

As the Simplified ER Model, which was mentioned above, is configured, the construct simple attribute in the model is represented by *simple attribute* along with *attributing*; complex attribute by *complex attribute* along with *attributing*; component of complex attribute by *simple attribute* along with *consisting*; relationship by *relationship* along with *relating*; while entity by *entity* singly.

3.5 Dealing with Constraints in the CDMS

As has been indicated above, which elements exist in a data schema restrict the potential set of data values allowed in all databases framed by the schema; the constructs contained in a data model restrict metadata values in all data schemata defined by the model; and the constructs which have been provided in the global data model restrict which constructs can be placed in a data model. In other words, the limited presence of classes in particular data schemata, types in particular data models, and meta-types in the global data model represent constraints at the schema level, the model level and the global level of the CDMS in some sense.

Referring to Figure 3.1, due to the restricted number of classes in the schema which describes the structure of the library system, only very basic information can be kept in the relevant database. In fact, only each member person's name, id, sex, age and address (consisting of only house, street and city), the title and number of each book owned by the library, and the person and book in relation with each loan can be recorded in the database, while information on a person's education, position and salary, the value of a book, the starting date and due date of an individual loan, etc, have to be ignored. In order to contain more details relevant to a library system, the data schema must be expanded to involve more classes.

Referring to Figure 3.3, because the simplified ER model does not involve a type similar to *free* or *specialisation* in IFO, it is impossible to directly construct a data schema which involves, say, student and staff as subclasses of *person* using the ER model. Moreover, connection type *attributing* consists of only four sub-connection types: *attributing* (from *entity* to *simple attribute*), *attributing* (from *entity* to *composite attribute*), *attributing* (from *relationship* to *simple attribute*) and *attributing* (from *relationship* to

composite attribute). Therefore no class specialised from *composite attribute* and no class specialised from *simple attribute* can be connected by a class specialised from *attributing* at all. However, a class specialised from *consisting* may connect a class specialised from *composite attribute* and a class specialised from *simple attribute*, examples of which are that *address* is connected by *address-house*, *address-street* and *address-city* to *house*, *street* and *city* respectively. The reason is that connection type *consisting* includes the required sub-connection type *consisting* (from *composite attribute* to *simple attribute*).

Referring to Figure 3.4, which represents the global data model, the fact that *meta-inheriting* consists of only *meta-inheriting* (from *meta-base* to *meta-base*) and *meta-inheriting* (from *meta-complex* to *meta-complex*) reflects that an inherited value/object is essentially the same as the original. In addition, the fact that *meta-relating* is not directed from *meta-base* to *meta-base* reflects the independent nature of base values. According to the definition, *meta-base* should epitomise all values representable and changeable directly through the interface; nevertheless, as the technology is constantly developing, the spectrum of such values is potentially expandable. In the currently implemented CDMS base values include *bool*, *int*, *real* and *string* only. For a more complete implementation for multimedia applications, however, *sound* and *image*, etc, should not be excluded.

There are other sorts of constraints in the context of the CDMS. In fact, the meaning of data in a database is comprehended not only based on the structures that is accommodated and the names given to the structures, but also based on the constraints on how the structures can be populated. As a data model is designed, the choice of constructs provided is made to cope with the payoff between providing highly constrained structures or providing less constrained structures together with facilities for specifying further constraints on these structures. For instance, the ER model contains one connecting type between relationships and entities together with the possibility of asserting cardinality and participation constraints on this type. A different model might provide different connecting types for multi-valued and single-valued connections.

From the CDMS point of view, all constraints should be dealt with in a consistent way with other modelling primitives. The four level CDMS architecture gives a firm basis for such a way. The global model provides meta-constraint types which can be used either to create constrained constructs in the model or to embed constraint specification constructs in the model. Using this framework, how and where constraints arise can be distinguished properly. At each level there are constraints which are fixed at that level and facilities for imposing further constraints at the levels below. The relevant issues, however, will be explored in much more detail in Chapter 4.

3.6 Dealing with Behaviour in the CDMS

The requirement for covering behaviour initially appeared in the field of office automation, where the need to refer to certain activities as entities in their own right occurred naturally.

In the context of the CDMS, broadly speaking, any piece of executable program can be looked upon as a behavioural class, with an execution of it being considered an instance. It is a useful perspective to view a piece of program as a single denotable 'value' which can be manipulated, that is, created, stored, modified and removed in the same way as an ordinary entity.

It is in the style of the CDMS that the model definition, data definition and data manipulation operations be regarded as behavioural objects. In this way, all of the code modules which change data, including metadata and meta-metadata, will be managed in a unified manner, similar to the unification of the management of all of the constraints, which was proposed in the previous section. The effect of this is that the configuration of a data model can encompass not only control over the data structuring facilities, but also the operations with which users use these facilities.

Thus the CDMS provides meta-behavioural types in the global model which can be used either to embed code into a model or to allow behavioural constructs to be put into the model. The relevant issues will be discussed in detail in Chapter 5.

3.7 Interaction Elements and Dialogue Primitives

The CDMS provides dialogue primitives which facilitate the operations for model definition, data definition and data manipulation.

As a model has been defined, the construction of the default interface for the model should be requested by the system. More interfaces for an existing data model can be created by repeating the interface definition process.

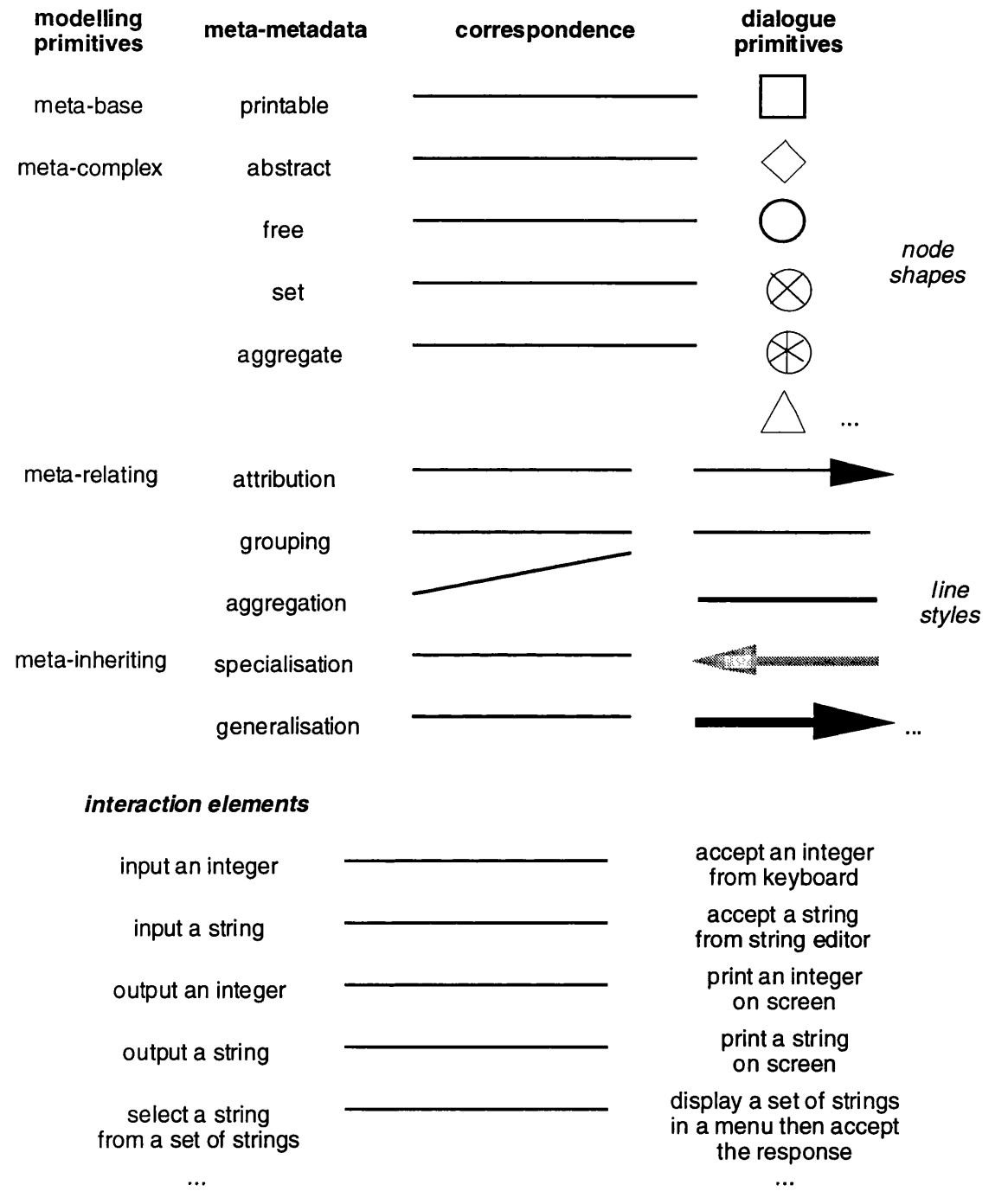

Figure 3.8 Interface Definition

An interface definition process is actually a process where a suitable dialogue primitive is allocated to each of the interaction element involved in the relevant data model (Figure 3.8). However, the details on the relevant issues are beyond the scope of this thesis.

Obviously, the symbols and line styles are limited to those stored in the system, and a mechanism which will avoid conflicting selections should be involved in the system design. For every data definition and data manipulation operation, the system provides the interaction element sequence, the associated action of a selection being either doing nothing or further providing an appropriate dialogue primitive menu for the DBE to select suitable dialogue primitives. Real operation procedures will then be generated according to the DBE's selections, and these will be embedded into the environment corresponding to the relevant model and interface, which will eventually become ready for use.

3.8 CDMS Functionality

The CDMS provides a component of a database system with which a variety of data modelling tools can be generated, each potentially with a variety of interfaces.

The facilities which are provided by the CDMS can be divided into four categories:

- model definition facility, intended for data model implementors;
- interface definition facility, intended for interface implementors;
- data definition facility, intended for data schema constructors;
- data manipulation facility, intended for end-users.

It seems appropriate to refer to both model and interface implementors as **database engineers** (DBEs), distinguished from **database administrators** (DBAs). Thus, a skilled DBE will construct data models and relevant interfaces through, say, a menu-driven program which permits the organisation of specialised generic modelling primitives and dialogue primitives through his or her thoughtful choices among the possibilities provided by the CDMS. These models and interfaces can then be utilised by DBAs for data definition and further by database end users for data manipulation.

3.9 Related Work

The idea of providing a system which permits the management of multiple data models and/or multiple user interfaces has attracted a fair amount of attention by researchers recently. In this section, a number of pieces of work in this area are reviewed.

The work of Roger King has concentrated on configuring the user interface in the context of an object oriented data model [King and Novak, 1993]. DBface is a toolkit which produces a visual framework from the database schema and then permits the user to specify operations over the data using state transitions. There are therefore two stages in the process of creating an interface. Firstly, a representation is created which defines three views: a data view defines how the database will be represented; a schema view defines how the classes and their connections will be seen; while a hierarchy view defines how the inheritance hierarchy is viewed. Secondly, the operations are created using components such as menus and primitive queries and updates. Thus DBface resembles the part of CDMS which deals with user interaction once the data model has been fixed.

The MOBIAS system developed at US West uses a meta-model to allow a variety of user interfaces to be developed and to co-exist on the same data [Durand *et al*, 1993]. MOBIAS has the same sort of four-level model as CDMS with the top level containing just two meta-constructs: the Data Object Type or DOT which subsumes all abstract and printable object types; and the Functional Object Type or FOT which subsumes all relationship kinds. Individual data models then have different kinds of DOT and FOT in a similar way to CDMS. By implementing user interaction primitives with respect to the meta-model, MOBIAS can support several different styles - a query language, natural language, forms and graphical interfaces - and these can then be used for different data models.

The DEDD (Design Environment for Deductive Databases) system is an extensible data model with a graphical interface [Radermacher, 1993]. The system presents a means to design new constructs and to provide graphical interaction tools for them. The creation of a new construct thus implies a semantic part - defining the nature of the construct and how it may be interacted with - and a graphical part - how it is visualised and how user actions initiate the interactions.

Atzeni and Torlone have explored a meta-modelling approach in the context of a project called INFOKIT [Atzeni and Torlone, 1993]. In a similar way to CDMS, they assert that the constructs in any data model can be reduced to a few categories: lexical types

(i.e. base types); entity types; aggregations; grouping constructs; functions; and generalisations. From this they propose a meta-model and a textual language for describing models in the meta-model. They then go on to tackle the difficult problem of schema translation between different models. This process is in two stages - from the source model to the meta-model is trivial since the meta-model subsumes any model, but from the meta-model down to the target model is more difficult, requiring that the model designer provides mechanisms for mapping each meta-construct into one or more constructs in the model.

At the University of Wisconsin, Yannis Ionnidis and his group have been developing a system, OPPOSUM, for permitting multiple graphical interfaces to co-exist on the same data [Haber *et al*, 1994, Haber *et al*, 1995]. OPPOSUM is built around a generic data model with primitives, abstract types, attributes and constraints. The connection between a model in OPPOSUM and a graphical interface to this is called a metaphor. The main contribution of the work seems to be a full analysis of the nature of such metaphors and how they may be evaluated.

One final area in which this kind of system emerges is in the design and implementation of Repositories [Bernstein, 1995]. A repository is a database which is intended to house design data emerging from a variety of tools and from a variety of environments. As such, a repository must handle multiple data models since they are designed to manage the data which is in pre-existing formats without requiring that the tools that create the data be re-implemented in any way. The repository system being developed by Microsoft Ltd uses the same four level architecture as the CDMS. It creates a meta-model which can cope with the models which underpin any of the applications whose data the repository is intended to handle. Clearly, given the wide variety of models which it can expect - RTF, SGML, HTML, ER, etc. - the successful design of the meta-model will be an impressive achievement.

To summarise, the work above is mostly motivated by the recognition that there is a great deal of commonality between the various data models. This commonality is then extracted and implemented as a highly abstract data model which can either be extended or instantiated. By implementing user interaction in the context of the most general model, it becomes possible to provide a great deal of the work of developing multiple user interfaces to multiple data models, without repetitive programming. Techniques for adding specific user interaction for specific operations complete the specification of the user interface.

None of the work described above seems to be advanced enough to deal with constraints or behaviour in a systematic manner as yet, however. All of the examples at best hint at constraint management and say nothing about behaviour. So the work proposed under the CDMS can be seen as an advance in these areas. On the other hand the CDMS work can be seen to lag particularly the work of Roger King and of Yannis Ionnidis in the area of user interface specification, but many of the techniques found in these papers seem to be directly relevant to the CDMS approach.

4 Constraints in the CDMS

Data in recent database applications such as design databases, multimedia and office information systems are extensively interrelated in various ways. To support this interrelation in an easy-to-use manner, rich conceptual structures are needed. As well as these structures, however, there is also a need to specify restrictions on the ways in which these structures may be used and combined. Support for integrity constraints in these applications needs more emphasis than in traditional applications. A systematic approach to constraint specification and enforcement is therefore required to replace the conventional unstructured approaches [Cooper and Qin, 1992].

The way in which constraints are treated in a traditional DBMS is deficient as a significant portion of the behaviour of a data model or a data schema is invisible and hence difficult to deal with. In addition, the management of constraints which are inherent to the data model is kept separate from that of constraints which are defined in the data schema, so they may not be handled in a unified way.

Deductive Database Systems promote constraints to be of primary importance [Naish and Thom, 1983]. They offer tools for specifying constraints and resolving their combinations. On the other hand, Frame Based Systems [Fikes and Kehler, 1985] provide slots for describing constraints on the structure of a database. Both have tended to be implemented in the context of untyped or dynamically typed environments. There is, however, a problem with maintaining database software which must wait until run-time before revealing that a piece of code is trying to use data of the wrong type. If the code is expected to be used over a long period of time, as is normal with database applications, then typing errors for infrequently used code may take years to turn up and consequently be very difficult to rectify. Also, type checking is necessary forever, rather than once only at compile-time.

In achieving the goal of managing multiple data models, each of which may embody different semantics, the issue of constraint management comes more sharply into focus, since the semantics of the data models will be largely expressed in terms of constraints. Thus there arises a requirement to manage both the constraints which are an inherent part of the data model and the constraints which are expressible in the data model together in a systematic way.

The specification and enforcement of constraints are essential functions of the CDMS. When a schema for a particular database application is created, one important task is to identify the constraints that must hold on the database framed by the schema. Similarly, when a model is constructed, it is crucial to declare the constraints that must hold on every schema that is to be defined by the model. **System semantic integrity** covers the techniques used to maintain the data model, the data schema and the database in a consistent state with respect to the constraints defined in the global model, model and schema. In order to prevent an inconsistent state occurring, the system should perform semantic integrity verification to determine whether any change will incur a constraint violation, and if this is the case, what appropriate action shall be taken consequently.

The CDMS is designed to provide a unified way to specify all constraints whether they act on the data, the schema or the model; whether they reside in the schema, the model or the global model; and whether they are defined as explicit constraints, are included by instantiating implicit constraints, or are inherent constraints. Since in the CDMS all modelling primitives, meta-metadata, metadata and data are treated as uniformly describable values and all possible constraints can be considered as predicates of one kind or another, this unified way to specify all constraints should eventually be able to be reached. The research starts by producing a categorisation of constraints and by identifying where in the architecture constraints shall reside.

In this chapter, Section 4.1 proposes some basic concepts concerning constraints; Section 4.2 presents the structure of integrity constraints in the CDMS; Section 4.3 categorises integrity constraints in the context of semantic data modelling; Section 4.4 describes the configuration of constraints within the CDMS framework; Section 4.5 discusses the methods for the management of constraints; and Section 4.6 summarises the issues presented in the whole chapter.

4.1 Some Basic Concepts of Constraints

A database stores information about some part of the real world, or a miniworld, and, as has been indicated earlier, a miniworld situation is always governed by some regulations or integrity constraints. Subsection 4.1.1 explains the role of constraints in a DBMS; Subsection 4.1.2 indicates that a constraint is in essence a predicate; Subsections 4.1.3 and 4.1.4 introduce constraint management and constraint placement respectively.

4.1.1 The Role of Constraints in a DBMS

In connection with the DBMS applications, it is usual to distinguish three sorts of integrity constraints that can be specified and enforced on the data schemata of the relevant data model [Elmasri and Navathe, 1989]. These are inherent constraints, implicit constraints and explicit constraints.

Inherent constraints, being part of the specification of the data model, automatically hold in all the schemata supported by the data model and hence do not need to be specified as a schema is defined. **Implicit constraints** are directly specifiable by facilities incorporated in the data model. In other words, each data model includes a possible set of implicit constraints that may be represented in a schema. **Explicit constraints** are not directly specifiable by measures incorporated in the data model, but may be explicitly specified and represented in a schema.

Take the ER model as an example, it is inherently established that in any ER schema every instance of a relationship class relates exactly one instance of each entity class participating in the relationship class in a specific role, while constraints such as key attributes on entity classes and structural constraints on relationship classes are implicitly specifiable in a particular ER schema. Explicit constraints add to an ER schema further restrictions such as a range restriction on an integer-valued attribute.

In IFO, an inherent constraint on specialisations/generalisations is that every instance in a subclass must also exist in its superclass, while such constraints as disjointness or coverage on specialisations are implicitly specifiable in a particular IFO schema.

Thus normally the use of an application is restricted partly by assertions made by the designer when setting up the schema and partly by decision forced by the data model. The CDMS brings the management of these together again.

4.1.2 Constraints as Predicates

Each constraint is essentially a predicate, that is, a boolean-valued function which takes as its argument a subset of the values held by the DBMS. The function value indicates whether or not the constraint has been violated. For instance,

$$a \in \text{age and } (a>0 \text{ and } a<120)$$

returns false if any *age* value lies outside the range 0 to 120.

Since the DBMS stores metadata as well as data, it is possible to hold constraints which limit the metadata. For instance,

$$\tau(X, Y) \text{ and } \tau \in \text{attribution and } (X \notin \text{attribute and } Y \notin \text{entity})$$

means that if Y is the class of an attribute of X then X must not be an attribute class and Y must not be an entity.

In the CDMS, the global model holds a number of templates for different kinds of constraint. These may be specialised to impose constraints at any of the levels of the CDMS architecture.

4.1.3 The Management of Constraints

Constraints are predicates which return a boolean value which indicates whether or not the constraint has been violated. The question arises what should happen if a constraint is violated. Ideally, there should be a constraint management component of a DBMS which enables DBMS users to specify clearly what should happen if a constraint is violated.

Some of the facilities which such a component could provide include:

- the ability to suspend the effect of a violation;
- the ability to re-impose the constraint;
- the ability to perform a check of all constraints;
- a clear mechanism for indicating how system should recover from constraint violation; and
- the ability to position constraint checks at particular points of the interaction with the database - check points, transaction boundaries and so on.

In fact, a DBMS typically provides very little in the way of systematic support for constraint management. Constraints are scattered through the system and each is dealt with in an *ad hoc* manner. By unifying the treatment of constraints, it is hoped that systematic constraint management is brought a step closer.

4.1.4 Placement of Constraints

In order to start to bring some order to constraint management, it is important to identify where in a DBMS the code which embodies each constraint is located. Traditional options are as follows:

1) In the **code implementing the data model**. This is where the inherent constraints and generic code for the implicit constraints will reside. In a program supporting the ER model, for example, there must be a part which forces attribute values to be printable and another part which has a generic form of structural constraints that is to be instantiated as appropriate.

2) In **assertions** specified by the user.

3) In **application code** as code fragments which provide checks on updates as well as inputs.

4) **Associated with computational values**. These constraints are a supplement to some piece of code and control its behaviour. Thus a

transaction, for instance, may have as part of its definition some kind of check, which can be further categorised as:

- **pre-conditions** or **guards**, which are restrictions that must hold if the code is to be executed;

- **post-conditions**, which if violated cause the effect of the code to be undone;

- **triggers**, which instruct the system to take actions when specified changes are attempted; and

- **exceptions**, which are error conditions that change the normal flow of control in a program.

The crucial issue is that constraints which are buried in code are not sensitive to the kind of management facilities which are required. In order to provide facilities which allow constraints to be suspended and re-imposed and to allow them to be visible to the user, it is necessary that they be values in the DBMS. Of the four sorts of constraint given above, only assertions can readily be treated in this way, although in TAXIS [Mylopolous *et al*, 1980], for instance, constraints associated with computed values are also nameable and thus manageable.

4.2 Constraints in the CDMS Architecture

A database is widely used to maintain information about a particular part of the real world, as has been explained in previous chapters. A schema is the framework of one or more databases, while a model is a common language in which schemata are defined. Semantic integrity constraints are restrictions on data models, data schemata and databases, ensuring that they reflect the relevant miniworlds accurately.

In the CDMS, a constraint can be categorised by a number of attributes:

1) At which level the constraint acts. A constraint may limit meta-metadata, metadata or data values.

2) Where the constraint is specified. A constraint can be built into the global model or be specifiable when constructing a model or defining a schema, or populating a database.

3) What kind of constraint it is. Some constraints restrict the range of some CDMS value. Others restrict combinations of values. Section 4.3 describes the varieties of constraint which the CDMS can manage. As a constraint is essentially a predicate, the form of a predicate decides the kind of constraint.

Following a subsection containing introductory examples, Subsection 4.2.2 describes the first and second of these attributes and how they interact. Clearly constraints can only act on levels lower than the one in which they are described. Subsection 4.2.3 gives some more extended examples.

4.2.1 Some Introductory Examples

Going back to the library database, here are a few constraints which limit interaction at the various levels:

1) Suppose one wished to put a photograph of the author of the books in the database. This requires an image base type but the CDMS does not support this. This is an example of a constraint acting at the data level which is embedded in the global model - a global-data constraint.

2) Suppose two people wished to borrow a book jointly. However, the ER model requires that a relationship instance relates exactly one instance from each participating entity class, so this is forbidden. This is an example of a constraint acting at the data level which is embedded in the model - a model-data constraint.

3) Next, there might be multiple authors of a book, but the schema specified one author to a book. This is an example of a constraint acting at the database level which is embedded in the schema - a schema-data constraint.

There are three other types of example:

4) It is impossible to define a schema in which a base node inherits from a complex node. This is forbidden by the global model - a global metadata constraint.

5) Similarly, it might be best to model the authors, staff members and library users as sub-classes of a person class, but the ER model does not have inheritance. Thus a constraint embedded in the model restricts the schema - a model-metadata constraint.

6) Finally, it is impossible to build a model in which there is a list construct, because the global model does not offer lists. This is a global meta-metadata constraint.

Thus all constraints in the CDMS arise in the global model either as inherent constraints (examples (1), (4) and (6)) or as identifiable constructs (examples (2), (3) and (5)). The constructs can be used either to embed constraints into a model (examples (2) and (5)) or become constructs in the model (example (3)). In the latter case the construct can be used to embed constraints into a schema.

4.2.2 Constraint Architecture in the CDMS

This subsection provides some definitions and categorisations of constraints in the CDMS.

The most important distinction is between **inherent** constraints and **implicit** constraints. The former are fixed constraints which limit values at a lower level, while the latter require further specification. Thus an inherent constraint, given a set of values, can return the value true or false according to whether it is violated or not. An implicit constraint, on the other hand, has one or more parameters left unspecified. It is provided as a construct for further specification of lower levels. Global inherent constraints are fixed in the CDMS system, while inherent constraints at lower levels of the architecture are either inherited from inherent constraints at higher levels or are produced by instantiating implicit constraints.

The next categorisation concerns the CDMS level which the constraint restricts. There are three alternatives:

- A **data constraint** limits the values of the data which may exist in a database - either by limiting individual data values or by limiting the connections between data values. Data constraints arise as limitations inherent to the schema of which the database is an instance, the data model in which the schema was defined, or the global model.

- A **metadata constraint** limits the values of the metadata which may exist in a data schema - i.e. the structures which may be represented in the schema. Metadata constraints arise either as constraints inherent to the data model being used to define the schema, or as inherent limitations of the global model.

- A **meta-metadata constraint** limits the values of the meta-metadata which may exist in a data model - i.e. the constructs which may be specified in the model. These are always constraints which are inherent to the global model.

An alternative categorisation is at which level the constraint exists:

- **Global constraints** occur as part of the CDMS system, either as inherent constraints limiting all use of a CDMS component, or as implicit constraints, which are used for defining data models and data schemata;
- **Model constraints** occur as part of a data modelling component, either as inherent constraints, limiting all use of schemata and databases built with the model, or as implicit constraints which can be used to define schemata;

- **Schema constraints** are inherent constraints of a schema which limit a database.

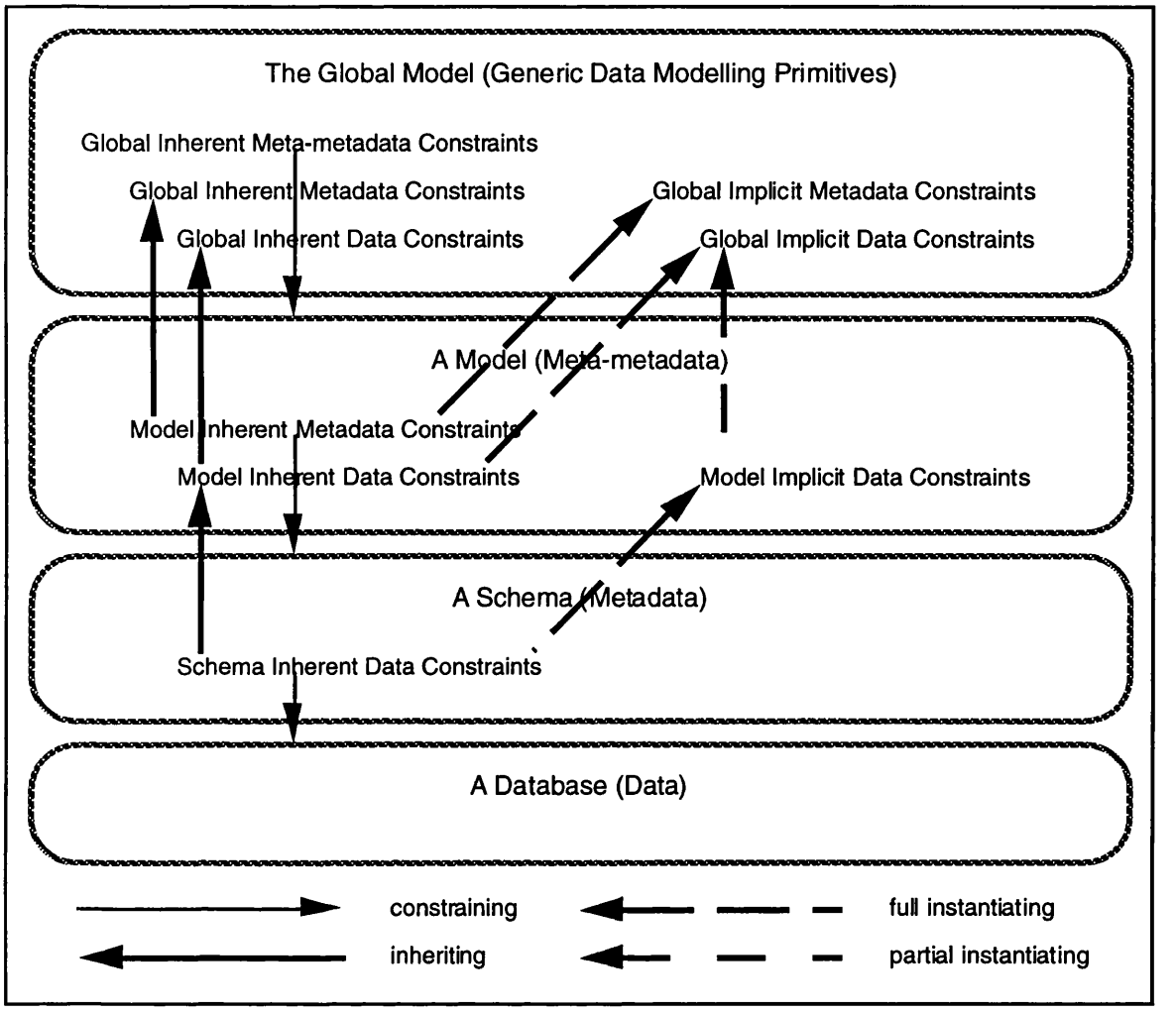

Figure 4.1 Constraint Structure in the CDMS

Figure 4.1 shows the whole set of constraints which the CDMS recognises. The figure shows four different relationships in the CDMS architecture in which constraints take part:

- **Constraining**. The end result of a constraint is that it becomes an inherent constraint which limits the use of the next level down.

- **Inheriting**. An inherent constraint at one level must also constrain all of the levels below.

- **Full instantiation**. If an implicit constraint has all of the relevant parameters fixed, then it yields an inherent constraint at the next level down.

- **Partial instantiation**. Alternatively, if an implicit constraint is taken down into a lower level with some or all of its constraints unspecified, then it becomes an implicit constraint at the lower level.

To take an example, there are the global implicit data constraints which limit the number of instances of a connection class, IC, which connect to a particular instance. As a parameterised predicate this looks like:

the connection class cardinality constraints
(IC, MINs, MAXs, MINt, MAXt):=
(MINs≤#(IC from a particular source)≤MAXs)∧
(MINt≤#(IC to a particular target)≤MAXt)

These constraints are specialised and become embedded in the ER model:

1) As an inherent constraint ensuring that every instance of *relationship* connects through an instance of *relating* to exactly one instance of *entity*;

2) As the implicit constraint to limit the number in which every instance of *entity* connects through instances of *relating* to instances of *relationship*.

In order to do these, the first three parameters are specified: IC becomes *relating*, both MINs and MAXs become 1, however, the last two parameters MINt and MAXt are not yet fixed, but are left to the schema designer.

Figure 4.2 shows this in detail and adds the further inheritance of the former and instantiation of the latter. As a result, every loan relates to exactly one library member, while every library member can borrow between 0 and 6 books.

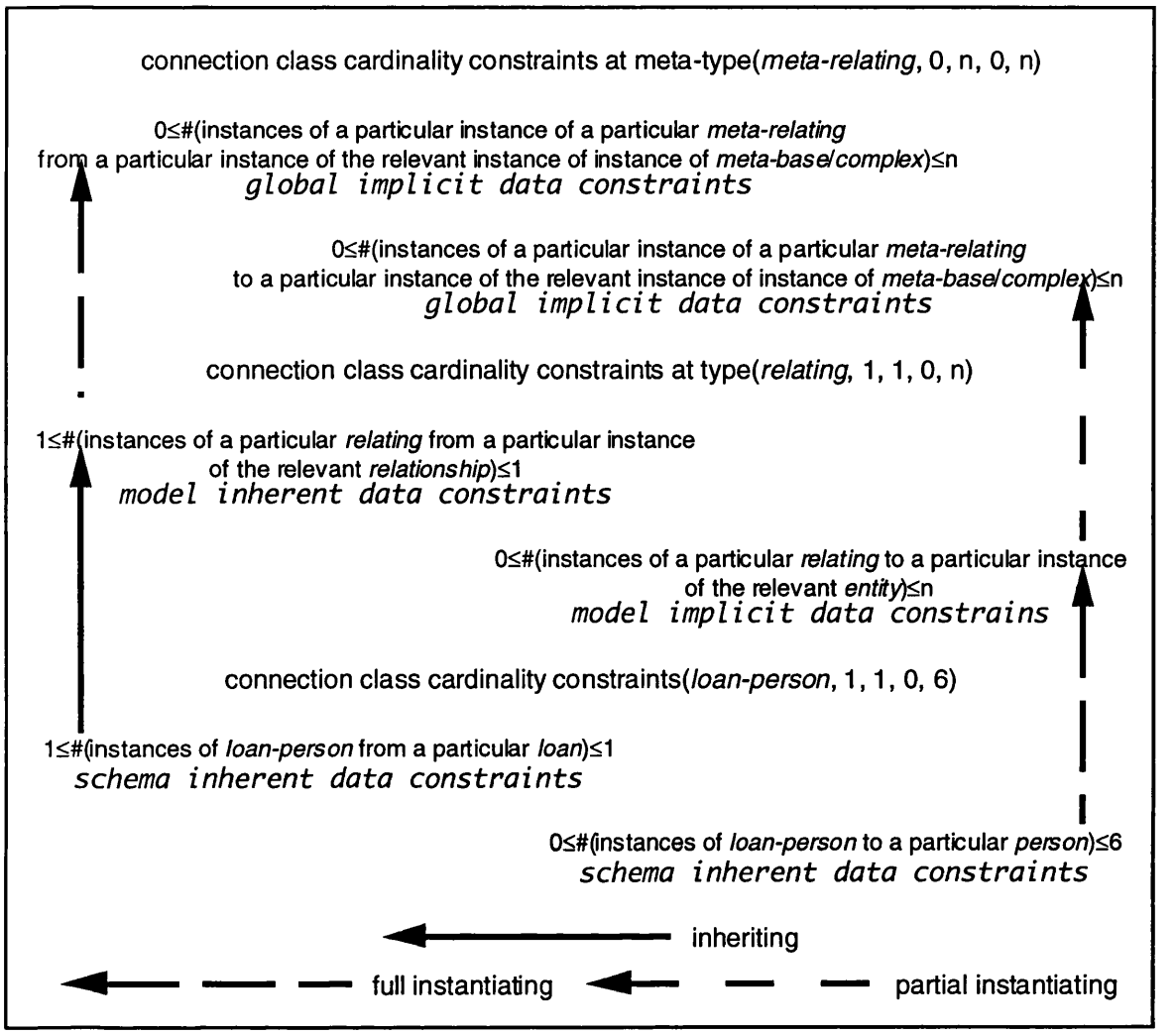

Figure 4.2 Constraint Specialisation

With this background, the next sub-section provides some fuller examples.

4.2.3 Further Examples

This subsection gives examples of the aforementioned constraints.

Examples of Schema Inherent Data Constraints

Figure 4.3 shows a number of schema inherent data constraints regarding the library system:

- a member person's age must be between 17 and 70;
- the number of memberships issued by the library must not exceed 1000;

- no member person may retain more than six books simultaneously;
- an address must consist of one house number, one street name and one city name;
- a loan instance must relate to a single person and a single book.

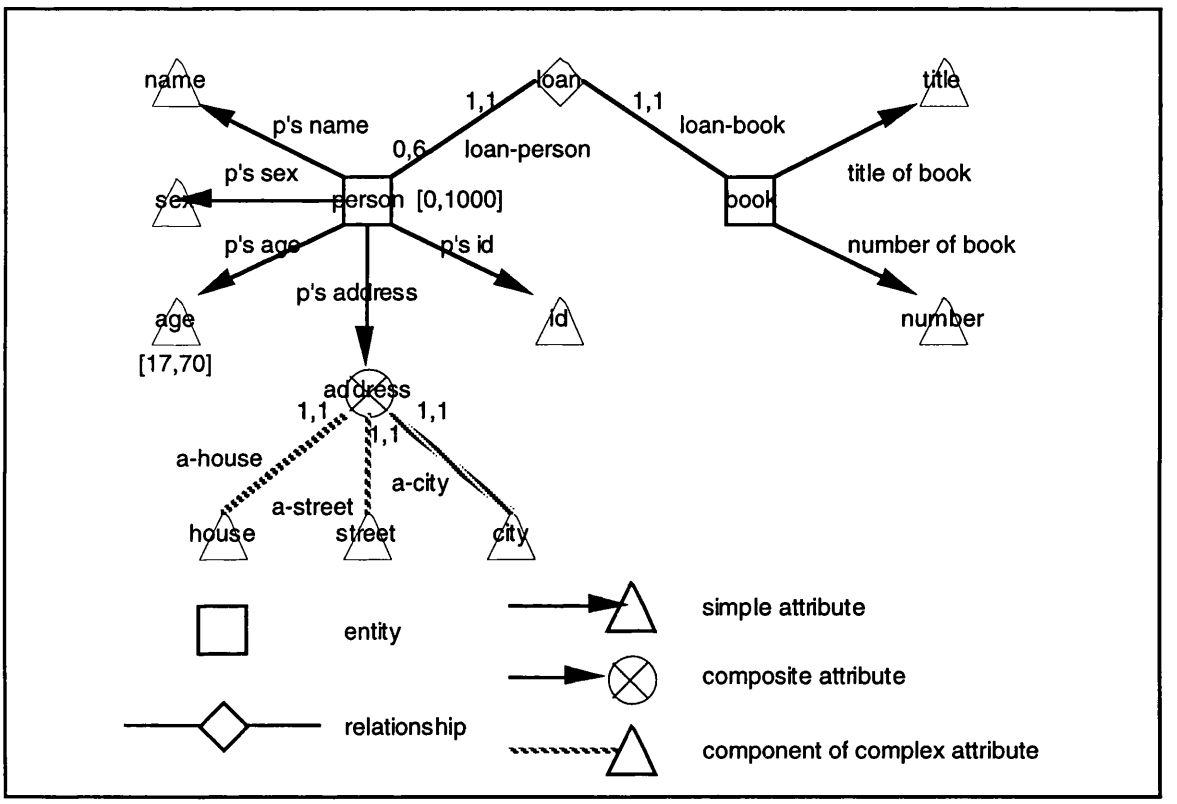

Figure 4.3 Schema Inherent Data Constraints

The first three constraints traditionally appear sometimes as fragments of the application programs which realise data manipulation and sometimes as assertions specified on the data, while in the CDMS they will be instantiated from model implicit data constraints. Thus the model has constructs for specifying such schema constraints - the constructs being partial instantiations of global implicit data constraints. The last two constraints are traditionally embedded in the DBMS software, while in the CDMS they are directly inherited from model inherent data constraints. Thus the model is itself restricted so that every *composite attribute* has one connection with each of their components, while every *relationship* has one connection with each of the entities they relate. These model inherent data constraints are themselves full instantiations of global implicit data constraints.

Examples of Model Implicit Data Constraints

Figure 4.4 shows a number of model implicit data constraints of the simplified ER model:

- a simple attribute class may have integer or string as its instances;
- the total number of instances of an entity class may be a non-negative integer;
- every instance of an entity class may be related by a non-negative number of instances of the relationship class which relates the entity class.

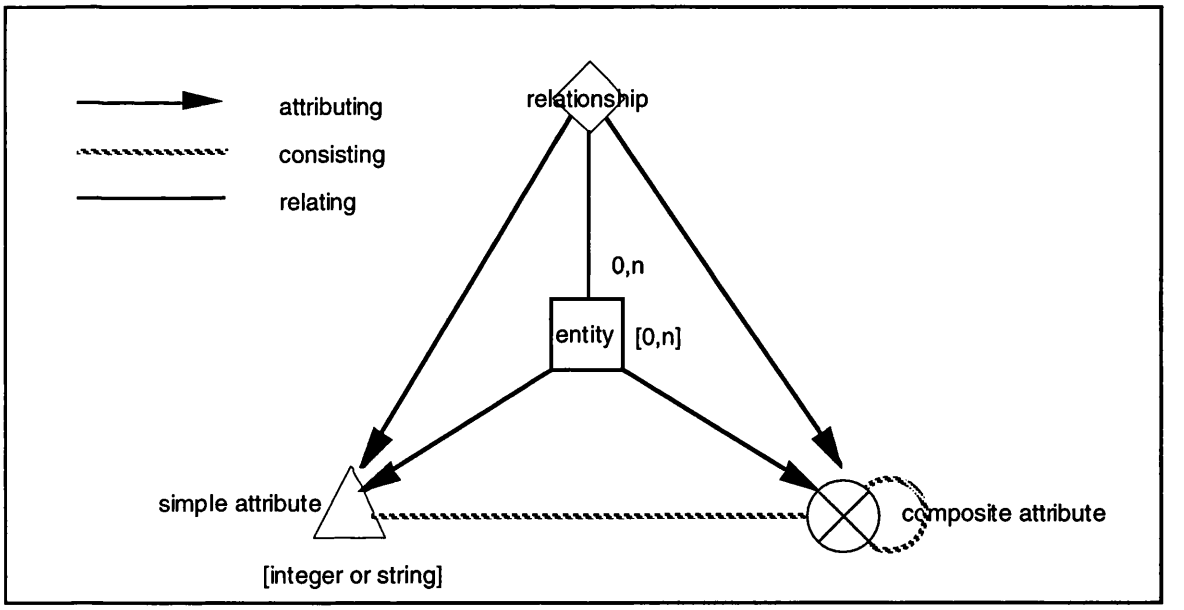

Figure 4.4 Model Implicit Data Constraints

These constraints are partially instantiated from global implicit data constraints. That is, they take a global implicit constraint and fix some of the parameters but not all. The first constraint requires further fixing to determine whether it is an integer or string, and may have its range restricted. The second constraint can be further specialised by fixing limits as the number of instances. The third can be further specialised by limiting the number of connections.

Referring to Figure 4.3, the first three constraints are particular instantiations of these three model implicit data constraints respectively.

Examples of Model inherent Data Constraints

Figure 4.5 shows a number of model inherent data constraints of the simplified ER model:

- every composite attribute instance of a composite attribute class C consists of exactly one simple attribute instance from each simple attribute class participating in C;

- every composite attribute instance of a composite attribute class C consists of exactly one composite attribute instance from each composite attribute class participating in C;

- every relationship instance of a relationship class R relates exactly one entity instance from each entity class participating in R in a specific role.

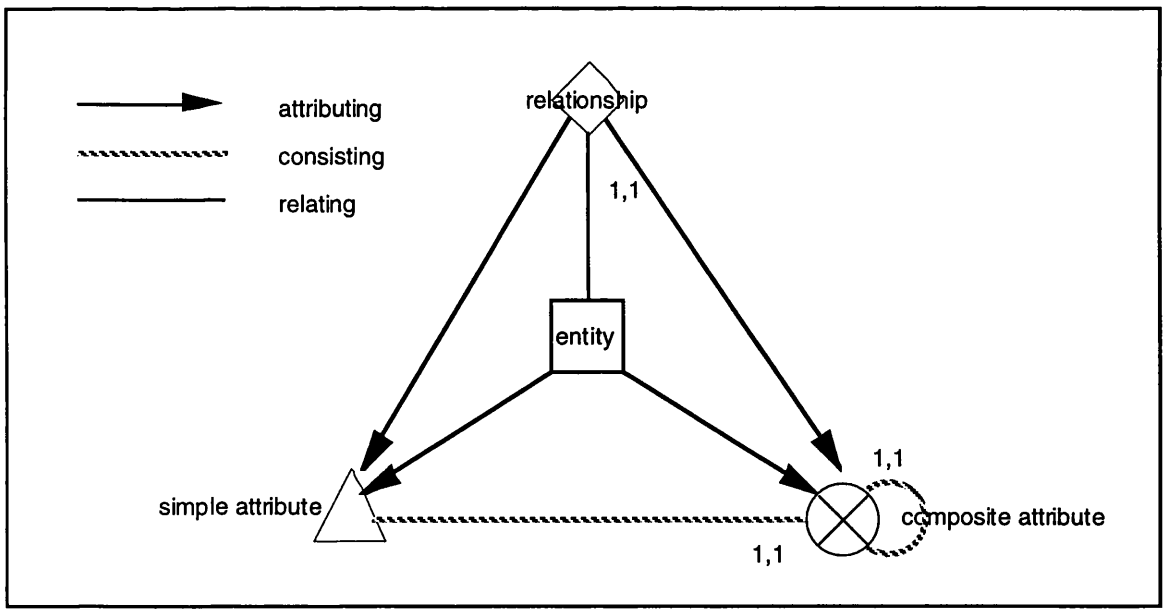

Figure 4.5 Model Inherent Data Constraints

These constraints are also instantiated from global implicit data constraints. This time, however, they have been completely fixed, so that they provide an absolute restriction of the schemata.

Referring to Figure 4.3, the last two constraints are directly inherited from the first and third model inherent data constraints above, respectively.

Examples of Model Inherent Metadata Constraints

Figure 4.6 shows a number of model inherent metadata constraints of the simplified ER model:

- a class specialised from *composite attribute* must consist of one or more classes specialised from either *simple attribute* or *composite attribute*;

- a class specialised from *relationship* must relate at least two classes specialised from *entity.*

Traditionally these constraints are embedded in the DBMS software. Here they are specialised from global implicit constraints.

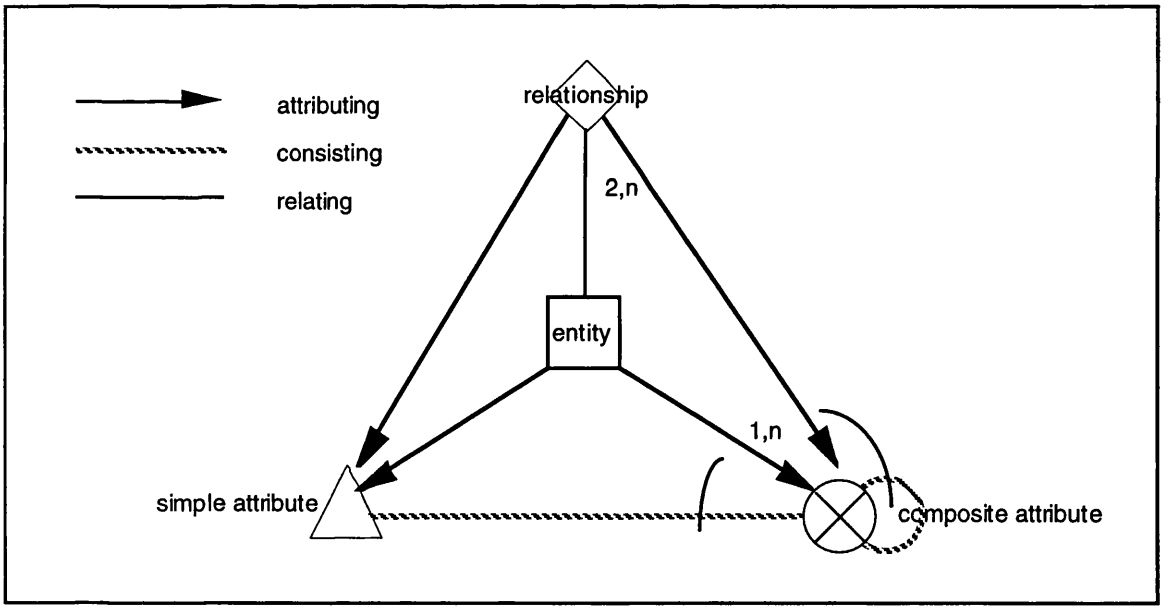

Figure 4.6 Model Inherent Metadata Constraints

Examples of Global Inherent Meta-metadata Constraints

Figure 4.7 shows an example of global meta-metadata constraints of the global data model:

- a type specialised from *meta-complex* must connect at least one other type in an appropriate way.

There is no traditional mechanism which treats this kind of constraint.

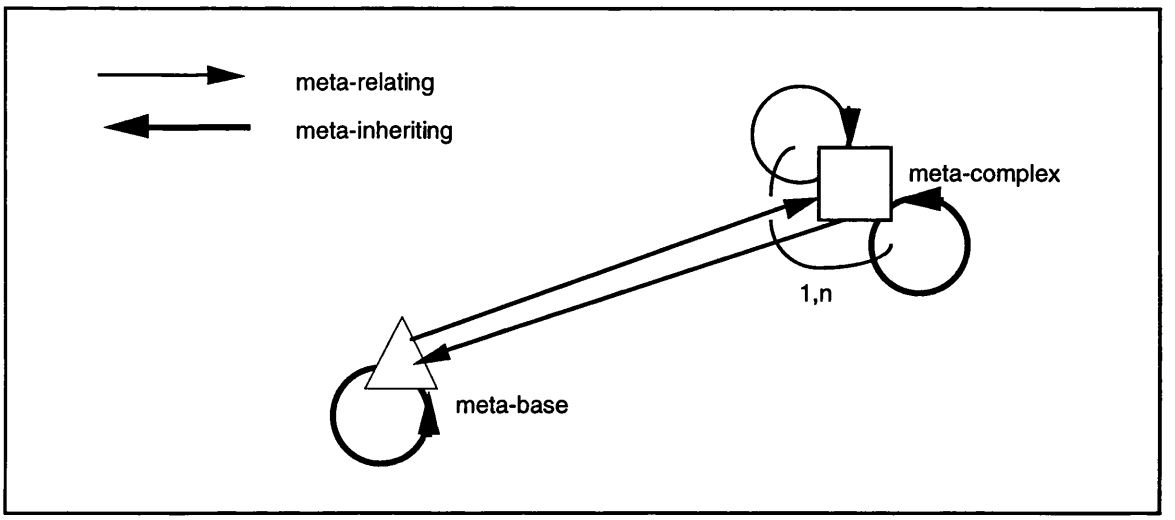

Figure 4.7 Global Inherent Meta-metadata Constraints

4.3 Varieties of Integrity Constraints

A **constraint** represents the semantics of the application. The relevant constraints must therefore be maintained so as to ensure the consistency of the database system.

As has been indicated in Subsection 4.1.2, a significant feature of a constraint is its predicate nature. Based on this, an orthogonal characterisation can be proposed. The CDMS reduces the great variety of constraints to a small set of predicate forms:

- **uniqueness constraint**, which requires that no duplicate value is allowed in a particular collection;

- **range constraint**, which requires that a value in a particular collection should belong to a subset of the potential values. This subsumes non-null constraints;

- **cardinality constraints**, which requires that the total number of a particular collection should fall within a given range;

- **graph constraint**, which limits the overall structure of a graph of values, for instance acyclicity;

- **general constraint**, which represents any constraint not covered above; an example of such is the one which requires that at least one instance of staff must have the position 'secretary'.

The rest of this section describes the constraints at each level in detail.

4.3.1 Schema-Data Constraints

As has been indicated before, data constraints restrict the values which data in a database may take. Generally speaking, each data schema should impose a consistent set of data constraints, or more precisely, schema inherent data constraints.

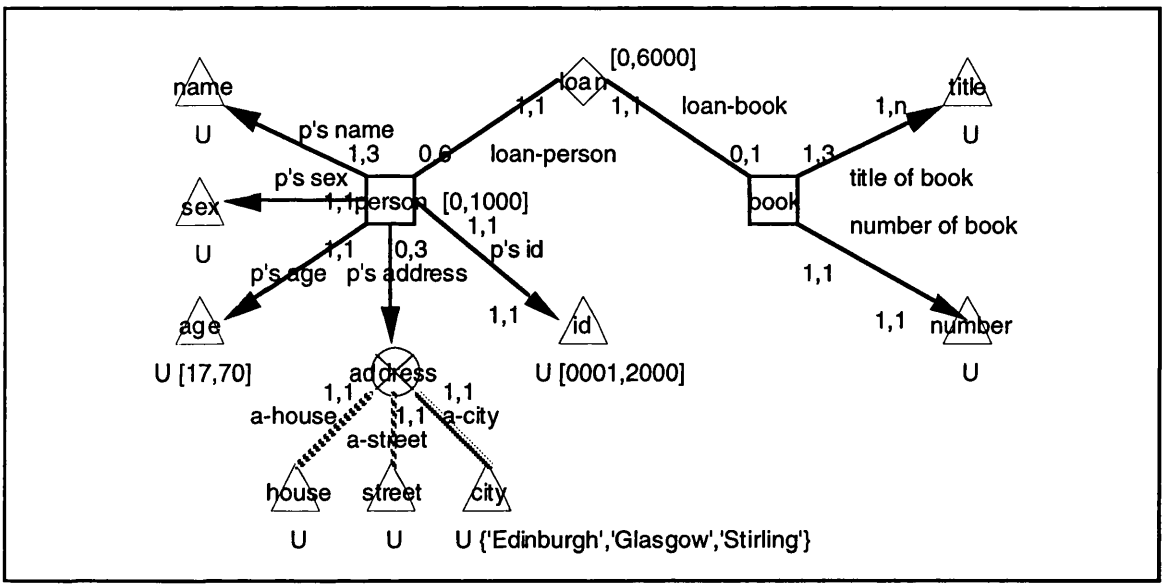

Figure 4.8 Data Constraints in a Simple ER Schema

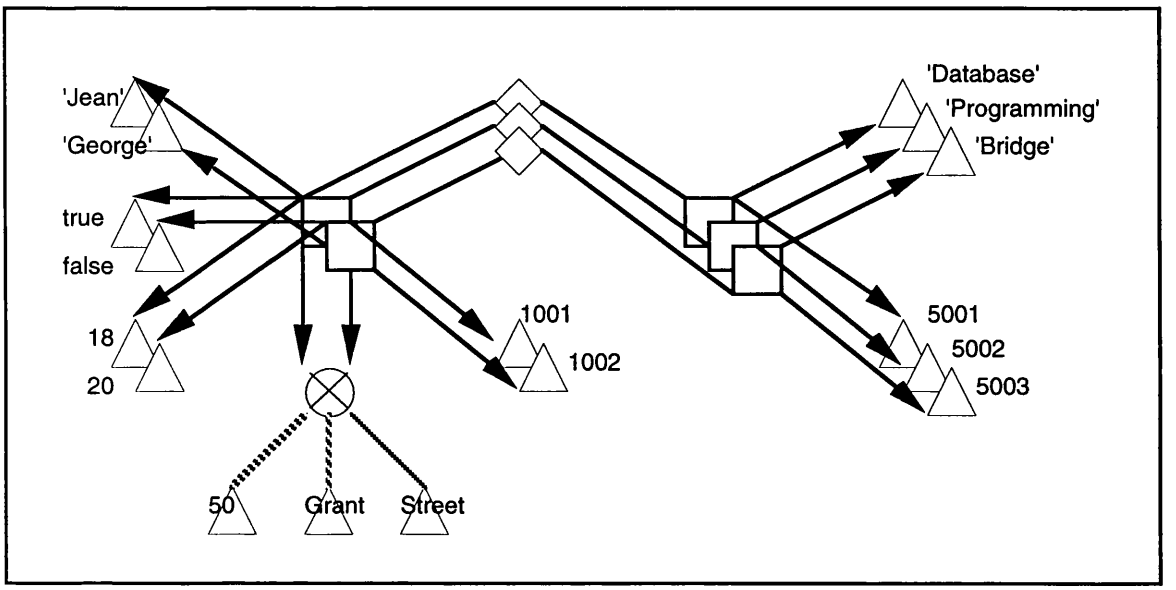

Figure 4.9 A Database Using the Simple ER Schema

In the context of the CDMS, schema-data constraints can be categorised as follows.

Base uniqueness holds on a base class and requires that each instance of the designated class should take a distinct value.

In the schema of Figure 4.8, such constraints hold on base classes *name*, *id*, *sex*, *age*, *house*, *street*, *city*, *title* and *number*, respectively. Figure 4.9 shows a database where there are no duplicate instances affiliated to any of these classes.

Base range holds on a base class and requires that each instance of the designated class should take its value from a given range.

Referring to Figure 4.8, for example, the following constraints hold:

- each instance of *age* should be between *17* and *70*;
- a valid instantiation of *city* could only be *'Edinburgh'*, *'Glasgow'* or *'Stirling'*.

Figure 4.9 shows that *18* and *20* as instances of *age* are both between *17* and *70*, and *'Glasgow'* as an instance of *city* does fall within the set of strings *'Edinburgh'*, *'Glasgow'* and *'Stirling'*.

It should be noted that the given range in such constraints might be expressed in various forms. In the example of Figure 4.8, the range of *age* is represented by an integer domain, while the range of *city* is represented by an enumerative set of strings. Actually, the concrete form depends on some higher level constraints. This issue will be further explored later. It should also be noted that **non-null** is just a special case of base range constraints, since null is *de facto* a special value.

Base/complex class cardinality holds on a base/complex class and requires that the total number of instances of the designated class should fall within a given range.

Referring to Figure 4.8, the following constraints hold:

- the total number of instances of *person* should be between 0 and 1000;
- the total number of instances of *loan* should be between 0 and 6000.

These constraints imply that no more than 1000 persons should be issued membership of the library; and no more than 6000 loans should be outstanding simultaneously.

Figure 4.9 shows that *person* has two instances and *loan* has three instances; that is, the relevant constraints are complied with.

Connection class cardinality holds on a connection class and requires that the total number of the instances of the designated connection class that are directed from or to a particular instance of the relevant class should fall within a given range.

Referring to Figure 4.8, for instance, the following constraints hold:

- the total number of the instances of *person's name* that are directed from a particular instance of *person* should be between one and three;
- the total number of the instances of *person's id* that are directed from a particular instance of *person* should be exactly one;
- the total number of the instances of *person's id* that are directed to an instance of *id* should be exactly one;
- the total number of the instances of *loan-person* that are directed from a particular instance of *loan* should be exactly one;
- the total number of the instances of *loan-person* that are directed to an instance of *person* must be between zero and six.

These constraints imply that each person may have up to three names, but must have a single id; each id must belong to a single person; each loan must relate to exactly one person; and no person may borrow more than six books at a time.

Now referring to Figure 4.9, in the relevant database each of the two persons has one name (*'Jean'* and *'George'* respectively) and one id (*1001* and *1002* respectively). Each of the two id's belongs to a single person (Jean and George respectively). Each of the three loan instances relates to one person instance (Jean, Jean and George,

respectively). Each person borrows no more than six books (two and one respectively). All these comply with the relevant connection class cardinality constraints.

It should be noted that, without violating the schema described by Figure 4.8, more than one person may have the same name, the same sex, the same age or share the same address, more than one address may have the same house number, the same street name or the same city name, and so on, because there is no corresponding constraint held.

Connection classes cardinality holds on a group of connection classes which connect a common class, and requires that the total number of the instances of the designated connection classes that connect, appropriately, a particular instance of the relevant class should fall within a given range.

Particular examples of connection classes cardinality constraints include covering and disjointness in relation to *specialisation* in IFO.

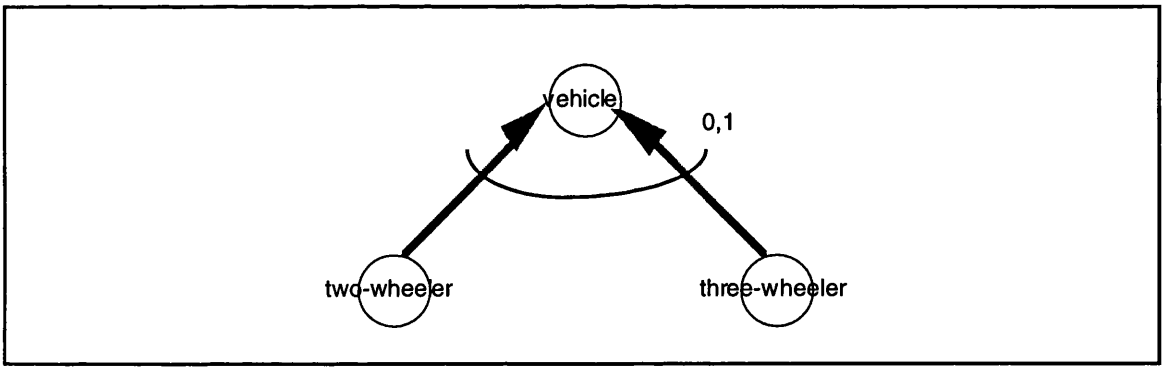

Figure 4.10 An Example of Disjoint Specialisation

Refer to Figure 4.10, which shows a part of an IFO schema. A connection classes cardinality constraint holds on the group which consists of *specialisation* (from *vehicle* to *two-wheeler*) and *specialisation* (from *vehicle* to *three-wheeler*), requiring that the total number of the instances of either connection class in the group that are directed from a particular instance of *vehicle* should be at most one. This implies that each vehicle must not be both a two-wheeler and a three-wheeler, that is, *two-wheeler* and *three-wheeler* are disjoint.

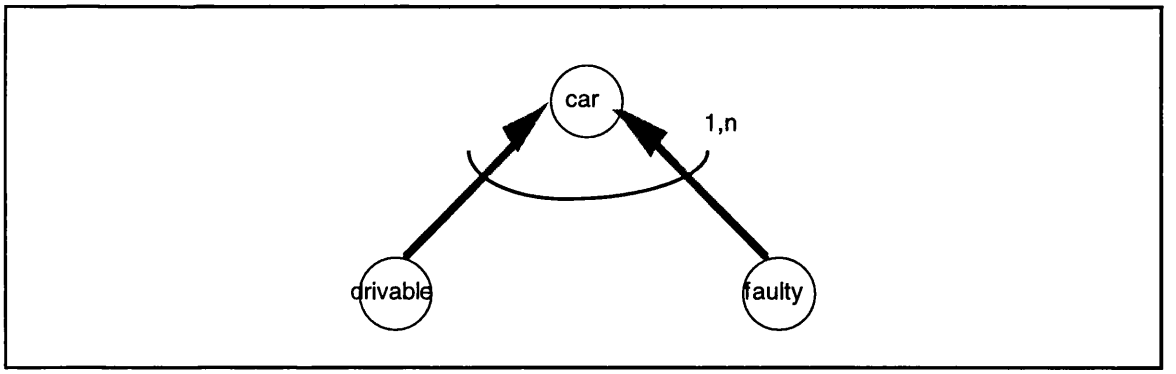

Figure 4.11 An Example of Covering Specialisation

Refer to Figure 4.11, which shows a part of another IFO schema. A connection classes cardinality constraint holds on the group which consists of *specialisation* (from *car* to *drivable car*) and *specialisation* (from *car* to *faulty car*), requiring that the total number of the instances of either connection class in the group that are directed from a particular instance of *car* should be at least one. This implies that each car must be a drivable car or a faulty car, that is, *car* is covered by *drivable car* and *faulty car*.

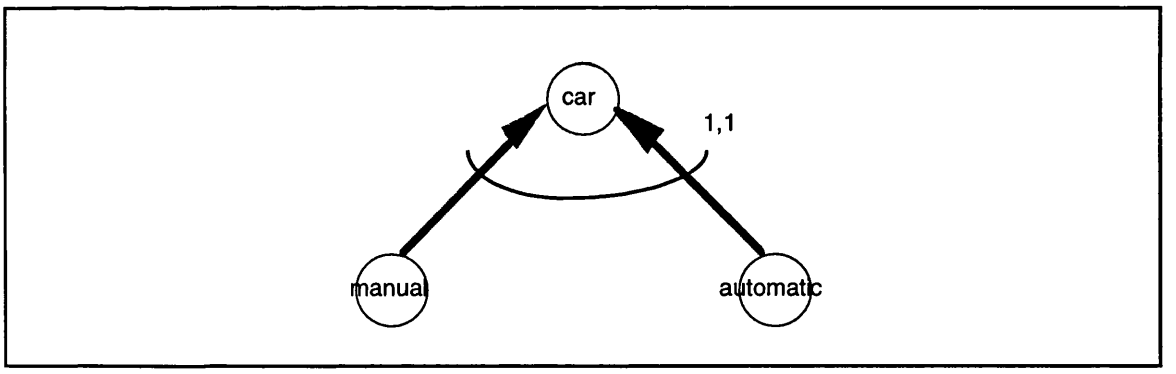

Figure 4.12 An Example of Disjoint Covering Specialisation

Now refer to Figure 4.12, which shows, again, a part of an IFO schema. A connection classes cardinality constraint holds on the group which consists of *specialisation* (from *car* to *manual car*) and *specialisation* (from *car* to *automatic car*), requiring that the total number of the instances of either connection class in the group that are directed from a particular instance of *car* should be exactly one. This implies that each car must be either a manual car or an automatic car. In other words, *car* is covered by *manual car* and *automatic car*, which are disjoint.

A connection classes cardinality constraint will reduce to a connection class cardinality constraint if only a single connection class is involved in the relevant group.

Connection classes combination cardinality holds on a combination of connection classes which connect a common class, and requires that the total number of the instances of the combination that connect some instance of the common class and a particular combination of instances of other relevant classes should fall within a given range.

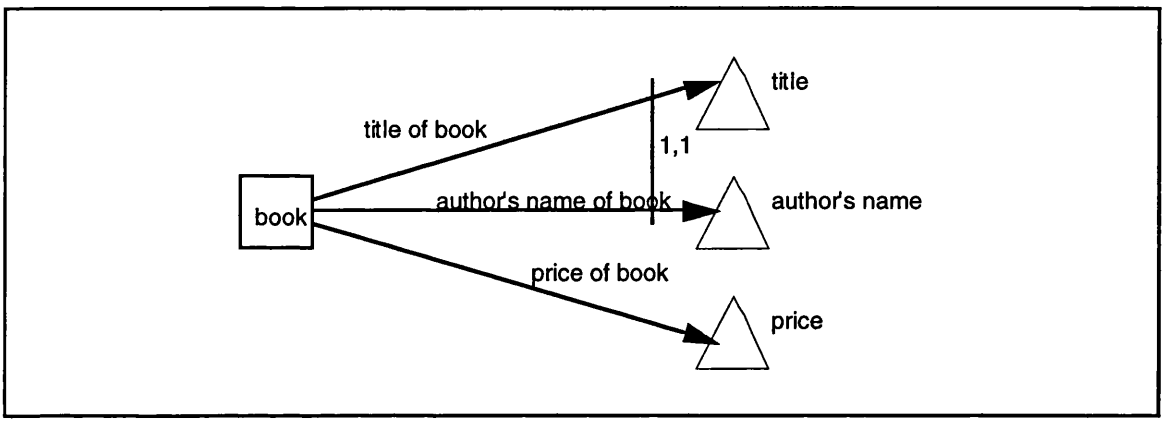

Figure 4.13 An Example of Connection Classes Combination Cardinality

Referring to Figure 4.13, which illustrates a part of an ER schema, a connection classes combination cardinality constraint holds on the combination of *title of book* and *author's name of book*, which are directed both from *book* but to *title* and *author's name* separately, requiring that the total number of the instances of the combination that connect some instance of *book* and a particular combination of instances of *title* and *author's name* should be at most one. This implies that a particular pair of title and author's name is not permitted to belong to more than one book. In this case, *title* and *author's name* act as the key attributes of *book*.

Regarding a connection classes combination cardinality constraint, if the combination involves only one connection class, then the constraint will deteriorate to a connection class cardinality constraint. In fact, if a key involves only one attribute, then the corresponding constraint can be denoted by a suitable connection class cardinality constraint straightforwardly. An example can be found in Figure 4.8, where a particular instance of *id* is never connected with more than one instance of *person's id*; that is, *id* is the key attribute of *person*.

It should be emphasised that a schema-data constraint is essentially a metadata value, and a metadata value must itself abide by the relevant metadata constraints. As has been indicated in Subsection 4.2.2, a schema inherent data constraint is either inherited from a model inherent data constraint or instantiated from a model implicit data constraint.

4.3.2 Model-Data Constraints

Model-data constraints, including model inherent data constraints and model implicit data constraints can, accordingly, be categorised as follows.

Base uniqueness at type holds on a base type and generally imposes a base uniqueness constraint on every class specialised from the designated type, requiring that each instance of the class should take a distinct value.

Base range at type holds on a base type and generally imposes a base range constraint on every class specialised from the designated type, requiring that each instance of the class should take its value from a given range.

Base/complex class cardinality at type holds on a base/complex type and generally imposes a base/complex class cardinality constraint on every class specialised from the designated type, requiring that the total number of instances of the class should fall within a given range.

Connection class cardinality at type holds on a (sub-)connection type and generally imposes a connection class cardinality constraint on every class specialised from the designated (sub-)connection type, requiring that the total number of the instances of the connection class that are directed from or to a particular instance of the relevant class should fall within a given range.

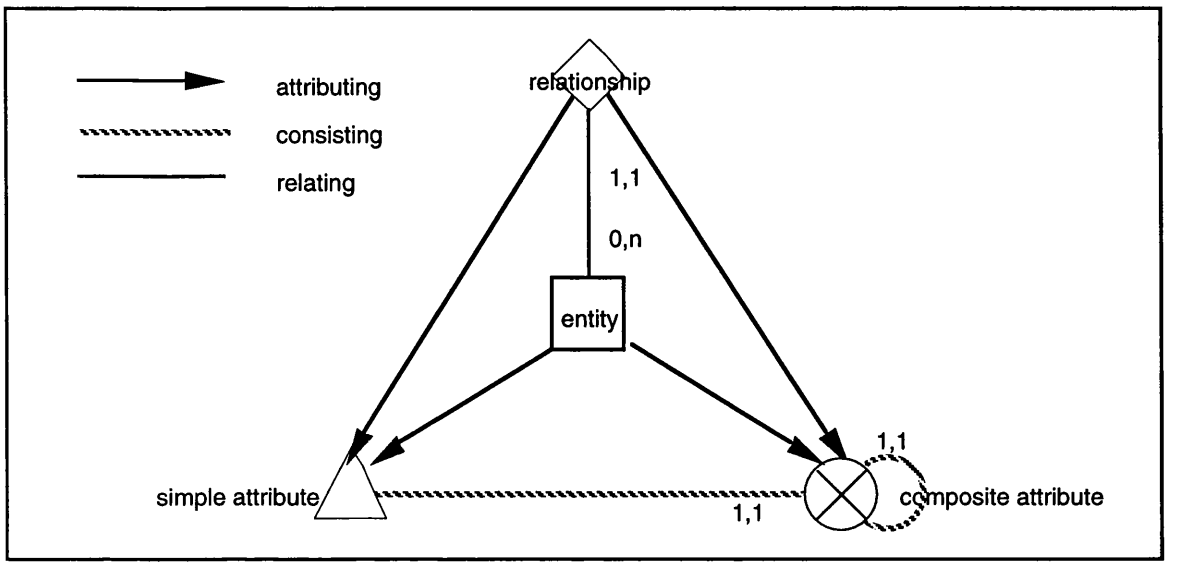

Figure 4.14 Connection Class Cardinality at Type

Referring to Figure 4.14, the following connection class cardinality constraints hold:

- the total number of the instances of each connection class specialised from *consisting* (from *composite attribute* to *single attribute*) that are directed from a particular instance of the relevant instance of *composite attribute* must be exactly one;

- the total number of the instances of each connection class specialised from *consisting* (from *composite attribute* to *composite attribute*) that are directed from a particular instance of the relevant instance of *composite attribute* must be exactly one.

- the total number of the instances of each connection class specialised from *relating* (from *relationship* to *entity*) that are directed from a particular instance of the relevant instance of *relationship* must be exactly one;

- the total number of the instances of each connection class specialised from *relating* (from *relationship* to *entity*) that are directed to a particular instance of the relevant instance of *entity* must be a non-negative integer.

These constraints are indicated in the figure by integer pairs [1,1] and [0,n], respectively.

Referring to Subsection 4.3.1, where a number of connection class cardinality constraints are listed, some connection class cardinality constraints are directly inherited from connection class cardinality constraints at type; some connection class cardinality constraints are instantiated from connection class cardinality constraints at type. An example of the former is that 'the total number of the instances of *loan-person* that are directed from a particular instance of *loan* should be exactly one' is inherited from 'the total number of the instances of each connection class specialised from *relating* (from *relationship* to *entity*) that are directed from a particular instance of the relevant instance of *relationship* must be exactly one'. An example of the latter is that 'the total number of the instances of *loan-person* that are directed to an instance of person must be between zero and six' is an instantiation of 'the total number of the instances of each connection class specialised from *relating* (from *relationship* to *entity*) that are directed to a particular instance of the relevant instance of *entity* must be a non-negative integer'.

Connection classes cardinality at type on a group of (sub-)connection types which connect a common type. This kind of constraint generally imposes a connection classes cardinality constraint on every group of connection classes. Each of these connection classes is an instance of one of the designated (sub-)connection types and connects to a common class. The constraint requires that the total number of the instances of the connection classes that connect a particular instance of the relevant class should fall within a given range.

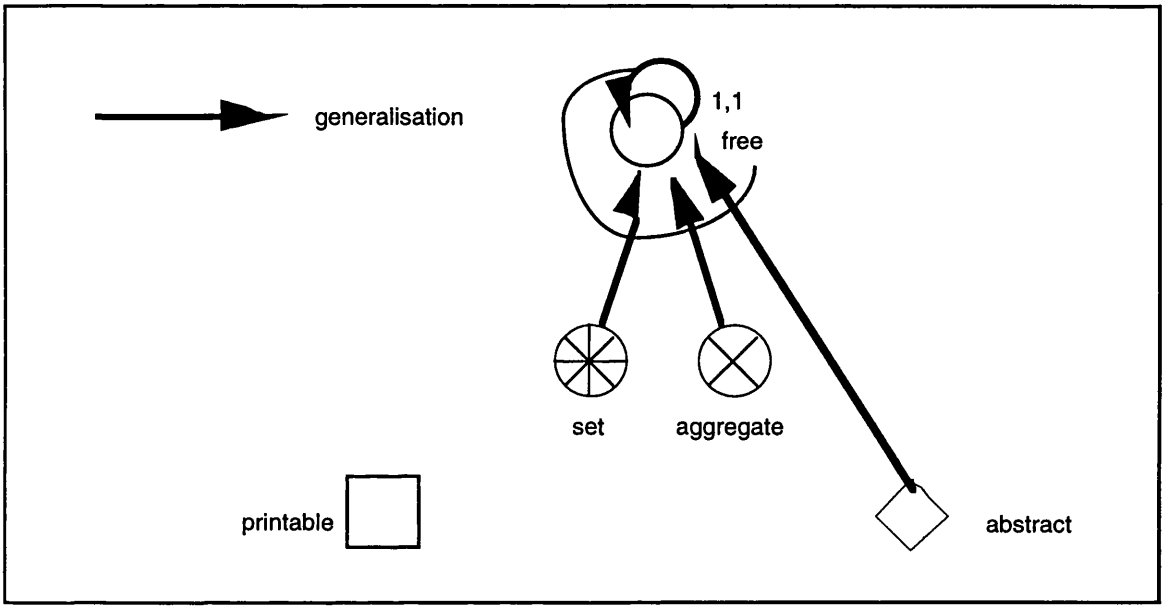

Figure 4.15 Generalisation of the IFO Model

Referring to Figure 4.15, which shows a part of the IFO model, a connection classes cardinality constraint at type holds on the group which consists of *generalising* (from *abstract* to *free*), *generalising* (from *set* to *free*), *generalising* (from *aggregate* to *free*) and *generalising* (from *free* to *free*), requiring that the total number of the instances of the instances of (sub-)connection types in the group that are directed to a particular instance of the relevant instance of *free* should be exactly one. This constraint is indicated in the figure by an integer pair [1,1]. This means that generalisation always comes with a covering constraint.

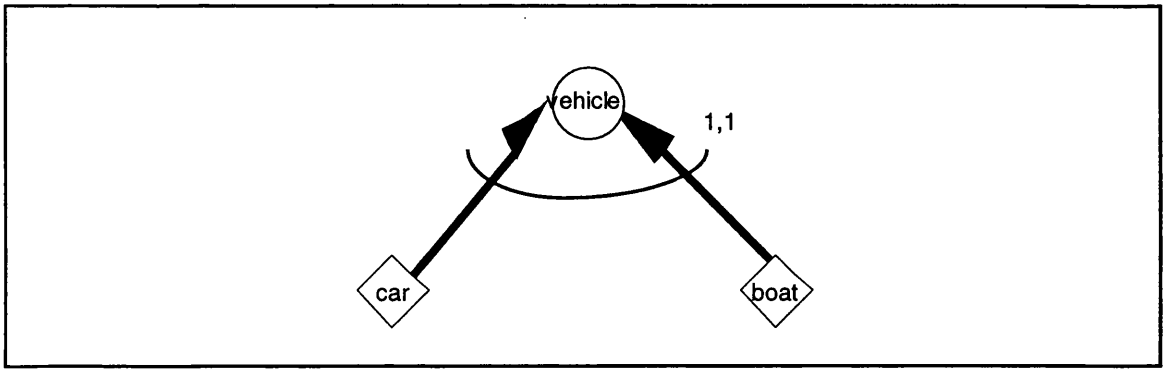

Figure 4.16 Generalisation in an IFO Schema

Now referring to Figure 4.16, which shows a part of an IFO schema, a connection classes cardinality constraint is actually imposed on the group which consists of *generalising* (from *car* to *vehicle*) and *generalising* (from *boat* to *vehicle*), requiring that the total number of such instances of either *generalising* (from *car* to *vehicle*) or *generalisation* (from *boat* to *vehicle*) directed to a particular instance of *vehicle* must be exactly one. Thus *car* and *boat* partition *vehicle*.

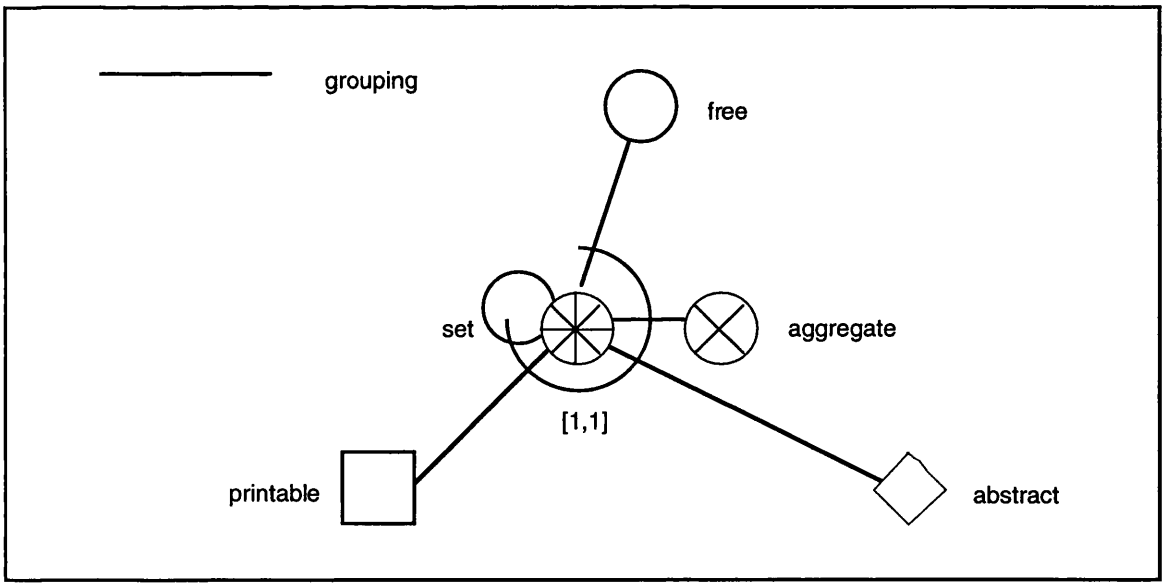

Figure 4.17 Grouping of the IFO Model

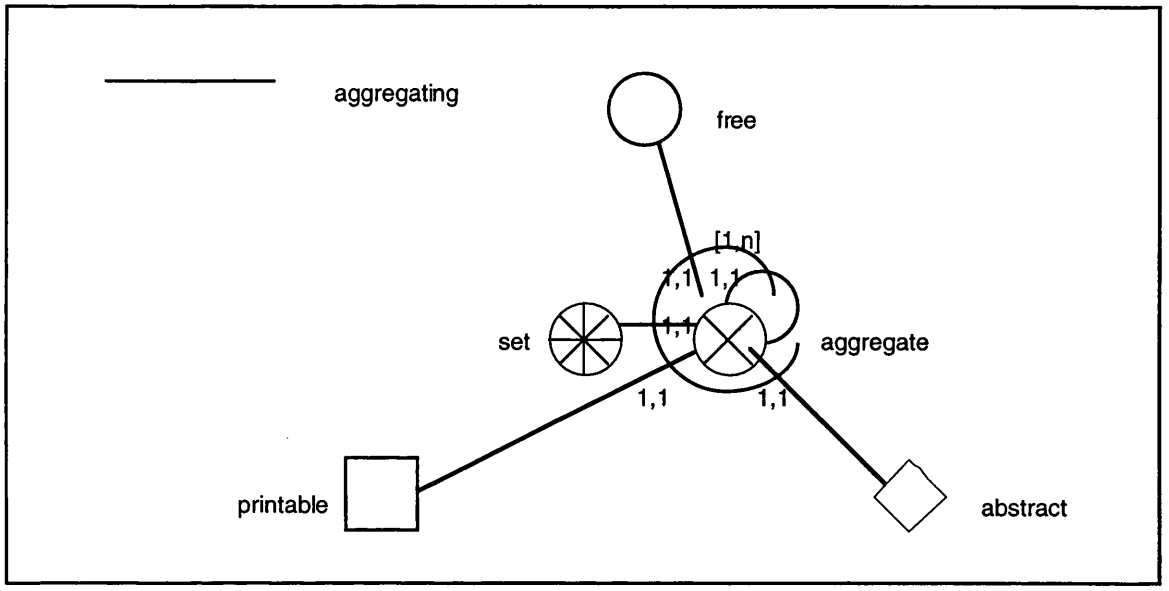

Figure 4.18 Aggregating of the IFO Model

From Figure 4.17 and Figure 4.18, which illustrate *grouping* and *aggregating* of the IFO model respectively, one can appreciate the unifying approach of the CDMS owing to its consistent constraint handling capacity. Although both *set* and *aggregate* are specialised from *meta-complex*, and both *grouping* and *aggregating* are specialised from *meta-relating*, they have been attached with different constraints and hence represent distinct semantics. In practice, each instance of *set* must be connected by one instance of *grouping* (due to a connection types cardinality constraint*); while each instance of *aggregate* must be connected by one or more instances of *aggregating* (also, due to a connection types cardinality constraint). Moreover, the number of such instances of an instance of *aggregating* that are directed from a particular instance of the relevant instance of *aggregate* must be one exactly (due to a connection class cardinality constraint at type), but no similar constraint exists in relation to *grouping*.

4.3.3 Global Data Constraints

Two global data constraints are as follows.

Connection class cardinality at meta-type holds on a (sub-)meta-connection type and generally imposes a connection class cardinality constraint on every class specialised from an instance of the designated (sub-)meta-connection type, requiring that the total number of the instances of the connection class that are directed from or to a particular instance of the relevant class should fall within a given range.

* Connection types cardinality constraint will be described in Subsection 4.3.4.

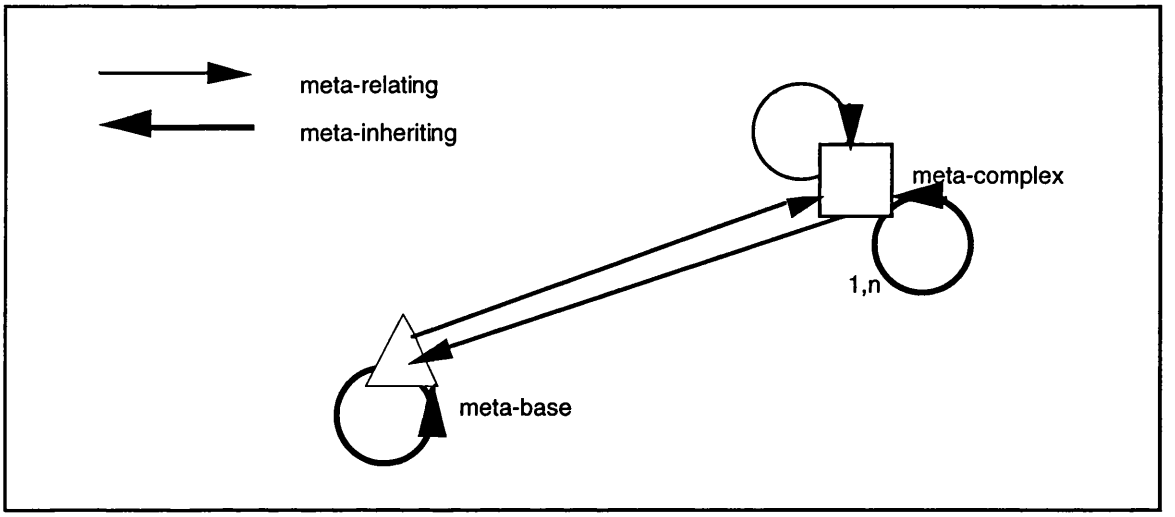

Figure 4.19 A Global Data Constraint

Referring to Figure 4.19, which represents the global data model, the following constraint holds:

- the total number of the instances of a particular instance of a particular instance of *meta-inheriting* (from *meta-complex* to *meta-complex*) that are directed to a particular instance of the relevant instance of the relevant instance of *meta-complex* should be at least one. This is indicated in the figure by the integer pair [1,n].

Connection classes cardinality at meta-type holds on a (sub-)meta-connection type and generally imposes a connection classes cardinality constraint on every group of connection classes each of which is specialised from an instance of the designated (sub-)meta-connection type and each of which connects, appropriately, a common base/complex class, requiring that the total number of the instances of the connection classes that connect, appropriately, a particular instance of the relevant class should fall within a given range.

4.3.4 Metadata Constraints

Metadata constraints restrict the values which metadata in a data schema may take. In the context of the CDMS, each data model should impose a consistent set of metadata constraints, or more precisely, model inherent metadata constraints.

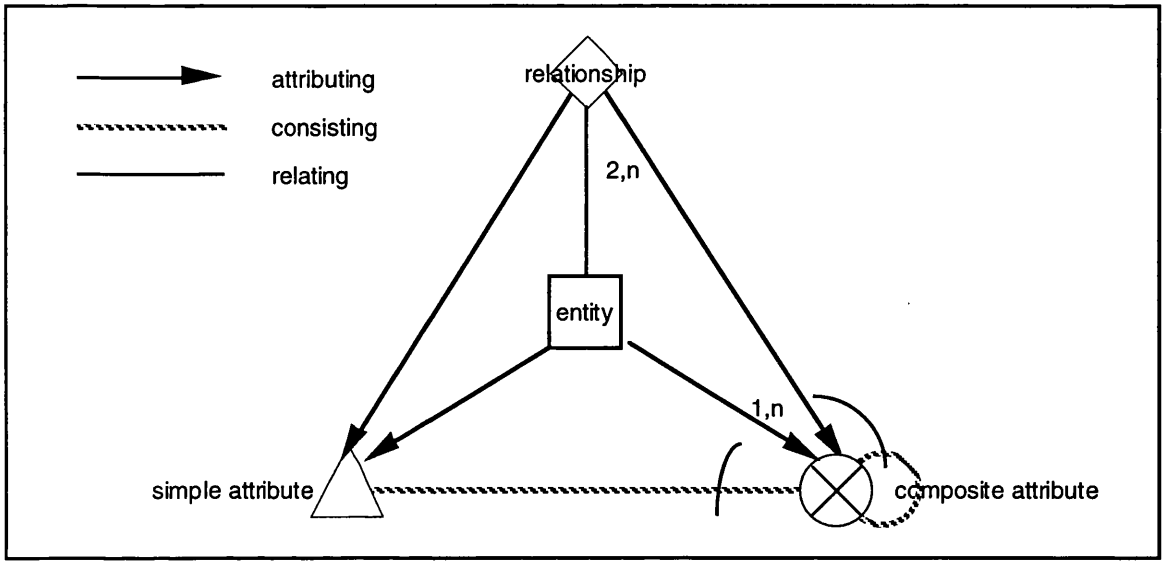

Figure 4.20 Model-Metadata Constraints

A number of model-metadata constraints can be found in Figure 4.20. In the simplified ER model, an instance of *composite attribute* must be directed by instances of *consisting* to at least one instance of *single attribute* or *composite attribute*, and an instance of *relationship* must be directed by instances of *relating* to at least two instances of *entity*. These constraints are indicated in the figure by integer pairs [1,n] and [2,n].

In the context of the CDMS, model-metadata constraints can be categorised as follows.

Connection type cardinality holds on a (sub-)connection type and requires that the total number of the instances of the designated (sub-)connection type that are directed from or to a particular instance of the relevant type should fall within a given range.

Referring to Figure 4.20, which represents a simplified ER model, the following constraint holds:

- the total number of the instances of *relating* (from *relationship* to *entity*) that are directed from a particular instance of *relationship* should be at least two.

This constraint implies that each relationship must relate to at least two entities.

Referring to Figure 4.8, where *loan-person* and *loan-book* are directed from *loan*, the total number of the instances of *relating* (from *relationship* to *entity*) that are

directed from a particular instance of *relationship* is two in the ER schema. The relevant connection type cardinality constraint is thus well maintained.

Connection types cardinality holds on a group of (sub-)connection types which connect a common type, and requires that the total number of the instances of the designated (sub-)connection types that connect a particular instance of the relevant type should fall within a given range.

Referring to Figure 4.20 again, the following constraint holds:

- the total number of the instances of either *consisting* (from *composite attribute* to *simple attribute*) or *consisting* (from *composite attribute* to *composite attribute*) that are directed from a particular instance of *composite attribute* should be a positive integer.

This constraint implies that each composite attribute must consist of at least one simple or composite attribute.

Referring to Figure 4.9, where *address-house*, *address-street* and *address-city* are directed from *address*, the total number of the instances of either *consisting* (from *composite attribute* to *simple attribute*) or *consisting* (from *composite attribute* to *composite attribute*) that are directed from a particular instance of *composite attribute* is really a positive integer (three plus zero, exactly). The relevant connection types cardinality constraint is thus complied with in this case.

Connection acyclicity holds on a connection type and prohibits any cycle consisting of instances of the designated connection type.

Referring to Figure 4.21, any group of class instances of *specialisation* type must not produce a cycle in IFO data schema, otherwise all involved classes would have to be identical, which would not be useful.

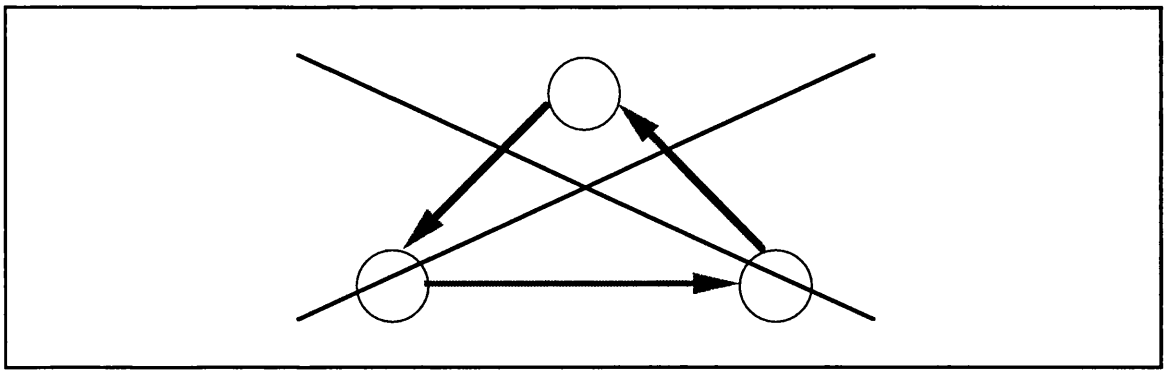

Figure 4.21 An Invalid Pattern in IFO Schema

It should be emphasised that a model-metadata constraint is essentially a meta-metadata value, and a meta-metadata value must itself abide by the relevant meta-metadata constraints. As has been indicated in Subsection 4.2.2, a model inherent metadata constraint is either inherited from a global inherent metadata constraint or instantiated from a global implicit metadata constraint.

Global metadata constraints can, accordingly, be categorised as follows.

Connection type cardinality at meta-type holds on a (sub-)meta-connection type and generally imposes a connection type cardinality constraint on every (sub-)connection type specialised from the designated (sub-)meta-connection type, requiring that the total number of the instances of the (sub-)connection type that are directed from or to a particular instance of the relevant type should fall within a given range.

Connection types cardinality at meta-type holds on a group of (sub-) meta-connection types which connect a common meta-type. This kind of constraint generally imposes a connection types cardinality constraint on every group of (sub-) connection types. Each of these (sub-)connection types is an instance of one of the designated (sub-)meta-connection types and connects to a common type. The constraint requires that the total number of the instances of the (sub-)connection types that connect a particular instance of the relevant type should fall within a given range.

4.3.5 Meta-metadata Constraints

Meta-metadata constraints restrict the values which meta-metadata in a data model may take. In the CDMS, the global data model imposes a consistent set of global inherent meta-metadata constraints.

Meta-metadata constraints can be categorised as follows.

Meta-connection type cardinality holds on a meta-connection type and requires that the total number of the instances of the designated meta-connection type that are directed from or to a particular instance of the relevant meta-type should fall within a given range.

Meta-connection types cardinality holds on a group of (sub-)meta-connection types which connect a common meta-type, and requires that the total number of the instances of the designated (sub-)meta-connection types that connect, appropriately, a particular instance of the relevant meta-type should fall within a given range.

Referring to Figure 4.22, the following constraint holds:

- the total number of the instances of *meta-relating* (from *meta-complex* to *meta-base*) and *meta-relating* (from *meta-complex* to *meta-complex*) that are directed from and the instances of *meta-inheriting* (from *meta-complex* to *meta-complex*) that are directed to a particular instance of *meta-complex* should be at least one.

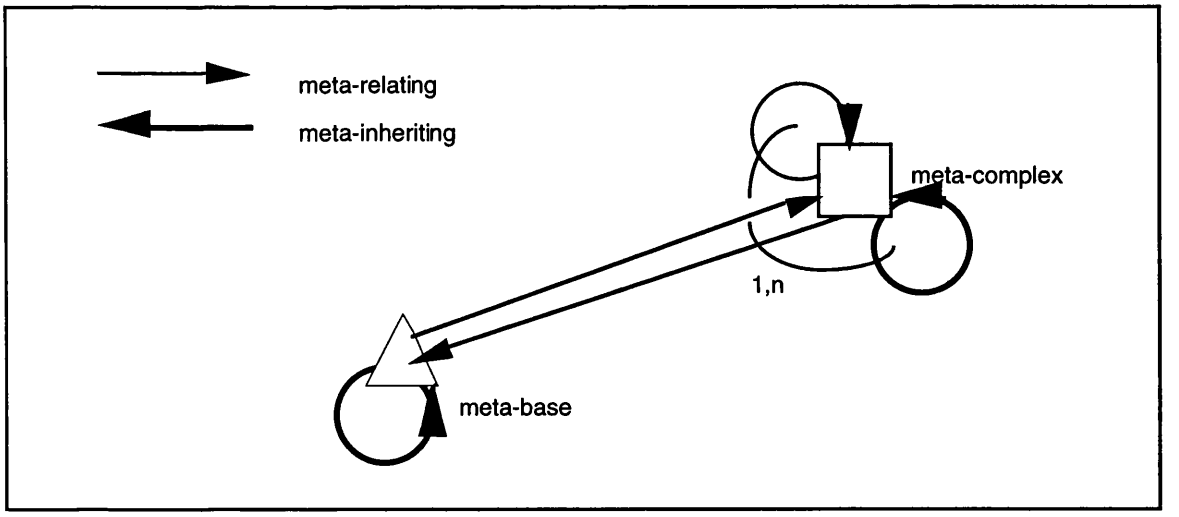

Figure 4.22 A Global Meta-Metadata Constraint

This constraint implies that each instance of *meta-complex* must at least either relate to an instance of either *meta-base* or *meta-complex* or be inherited from an instance of *meta-complex*.

4.4 Constraint Configuration in the CDMS

A distinct advantage of semantic data models is that they offer effective methods for describing semantic constraints. A constraint is a logical condition holding on a schema so as to be satisfied by all databases framed by the schema. Constraints thus provide a general means for expressing restrictions in the data model.

In the context of the CDMS, distinguishing levels skilfully and placing individual constraints at suitable places are important factors which help the system meet various users' needs, as well as being an advantage for the system's further development. If a constraint were put at an unsuitably high and general level, the system flexibility and usability would be reduced, because the low level designers would have no sufficient choices in this case. Otherwise if a constraint were not put at a level which is high and general enough, the system would become difficult to use, because the low level designers would have too many things to specify. The system consistency would be undesirably compromised in this case, too.

The four level architecture of the CDMS provides a firm basis for constraint configuration and constraint specification. Using this framework, how and where constraints arise can be distinguished systematically. At each suitable level there are constraints which are fixed at that level and facilities for imposing further constraints at the next lower level. For instance, the ER model forces simple attributes to be base values and offers the facility for imposing cardinality constraints on schema.

In this section, Subsection 4.4.1 introduces constraint configuration, while Subsection 4.4.2 describes constraint specification.

4.4.1 Constraint Configuration

Constraints exist in the CDMS in the same manner as other generic modelling primitives, hence constraint configuration is essentially a series of constraint specialisations.

Figure 4.23 shows examples of the configuration of data constraints in a specific model - a simplified ER model.

global-data constraints (primitives)	**model-data constraints** (meta-metadata)		**schema-data constraints** (metadata)	
	base uniqueness			
meta-base: {bool}	simple attribute: true		name:	*true*
			id:	*true*
			sex:	*true*
			age:	*true*
			house:	*true*
			street:	*true*
			city:	*true*
			title:	*true*
			number:	*true*
	base range			
meta-base: {bool, int, real, string}	simple attribute:	{bool}	sex:	{*bool*}
		{int}	id:	[1001, 2000]
			age:	[17, 70]
			house:	[1, n]
			number:	[0001, 9999]
		{real}		
		{string}	name:	{string}\{''}
			street:	{string}\{''}
			city:	{'Edinburgh', 'Glasgow', 'Stirling'}
			title:	{string}\{''}
	base/complex class cardinality			
meta-base: [0, n]	simple attribute:	[0, n]	name:	[0, n]
			id:	[0, 1000]
			sex:	[0, n]
			age:	[0, n]
			house:	[0, n]
			street:	[0, n]
			city:	[0, n]
			title:	[0, n]
			number:	[0, 9999]
meta-complex: [0,n]	entity:	[0, n]	person:	[0, 1000]
			book:	[0, n]
	complex attribute:	[0, n]	address:	[0, n]
	relationship:	[0, n]	loan:	[0, 6000]
	connection class cardinality			
meta-relating: [0, n], [0, n]	attributing:	[0, n], [0, n]	person's name:	[1, 3], [0, n]
			person's id:	[1, 1], [1, 1]
			person's sex:	[1, 1], [0, n]
			person's age:	[1, 1], [0, n]
			title of book:	[1, 3], [1, n]
			number of book:	[1, 1], [1, 1]
			person's address:	[0, 3], [0, n]
	consisting:	[1, 1], [0, n]	address-house:	[*1, 1*], [1, n]
			address-street:	[*1, 1*], [1, n]
			address-city:	[*1, 1*], [1, n]
	relating:	[1, 1], [0, n]	loan-person:	[*1, 1*], [0, 6]
			loan-book:	[*1, 1*], [0, 1]

Figure 4.23 Configuration of Data Constraints

- **base uniqueness**: Base uniqueness is a global implicit data constraint, that is, base uniqueness is open for model definition. As the model is created, *meta-base* is specialised to form *simple attribute*, on which the base uniqueness constraint is imposed (*true* is a specialisation of {*bool*}). That is, in all the schemata to be described by the model, all simple attribute classes will have no duplicated value. A model inherent data constraint is thus configured in the model. For the particular schema, this is shown in the figure since all nine attribute classes have this constraint set to true.

- **base range**: Base range is another global implicit data constraint. *Meta-base* is actually specialised to form four sorts of *simple attribute*, imposed with different base range constraints. As *meta-base* generally requires its associated data must take values from the set of all booleans, integers, reals and strings, four sorts of *simple attribute* require their associated data must take values from the set of booleans, the set of integers, the set of reals and the set of strings, respectively. In such a way, four model implicit data constraints are configured in the model by specialising the equivalent global implicit data constraint. These have been instantiated in the schema for each of the attributes.

- **base/complex class cardinality**: This global implicit constraint permits the restriction of the number of members of a base or complex class. At the global level and in the particular model the only restriction is that the number of members is non-negative. In the schema, for instance, the numbers of members of *person* and *loan* are further restricted to not more than 1000 and not more than 6000 respectively.

- **connection class cardinality**: This is another global implicit data constraint, this time limiting the number of instances at each end of a connection. In general, *meta-relating* is unrestricted and remains so for *attributing* - i.e. there is no restriction to the number of values of an attribute, nor the number of times any value can be an attribute. In the schema, however, this is restricted for each attributing connection. For instance, everyone has between 1 and 3 names, but the same name can be shared by any number of people. *Consisting* connects composite attribute values with their components and every composite attribute value is defined by exactly one such connection for each component, while the same attribute value can be components in any number. At the schema level, this is refined so that every

address must have a house number, street and city name, while any street name in the database must be part of at least one address. *Relating* connects *relationships* to *entities*. Every relationship instance is connected to one instance of the participating entities, but any entity can take part in any number of relationships.

At the schema level these constraints have different effects. That a relationship value is connected to exactly one instance of an entity is an inherent constraint of the model and so is inherited at the schema level. Conversely, the ability to specify how many relationships an entity is involved in is an implicit constraint in the model. In this case it has been instantiated twice - once to assert that people can borrow up to 6 books and once to assert that each book can either be borrowed or not.

global-metadata constraints (primitives)	**model-metadata constraints** (meta-metadata)
	connection type cardinality
meta-relating: [0, n], [0, n]	relating: relationship [2, n] - entity [0, n]
	connection types cardinality
meta-relating: [0, n]	consisting: composite attribute [1, n] - {simple attribute, composite attribute}
	connection acyclicity
meta-relating: {bool}	attributing: true consisting: true relating: true

Figure 4.24 Configuration of Metadata Constraints

Figure 4.24 shows examples of the configuration of metadata constraints in a specific model - again, a simplified ER model.

- **connection type cardinality**: This is a global implicit metadata constraint, which restricts the number of connections allowed to any classes. The only restriction specified in the model is that every *relationship* must be in at least two *relating* connections.

- **connection types cardinality**: This is also a global implicit metadata constraint, which is specialised to hold on the *consisting* connection type so

that it connects to at least one component, which may be simple or composite attribute.

- **connection acyclicity** is another global implicit metadata constraint, which is specialised to hold on *attributing* (since attributes cannot have attributes), *consisting* (since a composite attribute cannot be its own component) and *relating* (since the connection is from a relationship to an entity), respectively.

4.4.2 Constraint Specification

In all cases, constraints operate by taking a relatively abstract construct and making it more concrete. In the CDMS, the creation of schema components and of model constructs operates in exactly the same way; that is, supplying a name and constraining the values it may take. Since model constructs, schema components and data are all values in the CDMS store, it becomes possible to unify the treatment of the various kinds of constraint no matter where they reside.

As has been explained before, all metadata constraints and data constraints exist in some form at the global level. That is, there are global inherent metadata constraints, global implicit metadata constraints, global inherent data constraints and global implicit data constraints in the global model.

When a particular data model is defined, global inherent constraints are automatically inherited to become model inherent constraints; while global implicit constraints need to be instantiated to become model inherent constraints or model implicit constraints. Therefore, during a model definition process, firstly the appropriate base types and complex types will be defined, secondly the appropriate relationship types will be defined, thirdly some metadata constraints and some data constraints will be inherited from global inherent constraints automatically, and finally some other metadata constraints and some other data constraints will need to be specified using facilities for the instantiation of the global implicit constraints. As a model is modified, a similar process will be carried out.

In order to specify a metadata constraint, the model designer selects an entry from the metadata constraint menu* , which displays the names of all the global implicit

* Note, once again the interface is designed to make explicit the architecture of CDMS and not to provide usability.

metadata constraints, and then the system will provide a dialogue to allow the designer to input the necessary information. In parallel, in order to specify a data constraint, the model designer selects an entry from the data constraint menu, which displays the names of all the global implicit data constraints, and then the system will provide a dialogue to allow the designer to input the necessary information. When all these are complete, metadata constraints and data constraints will appear along with other model constructs which constitute the model; that is, base types, complex types and relationship types.

Figure 4.25 shows a window in which the specification of constraints in a data model is conducted.

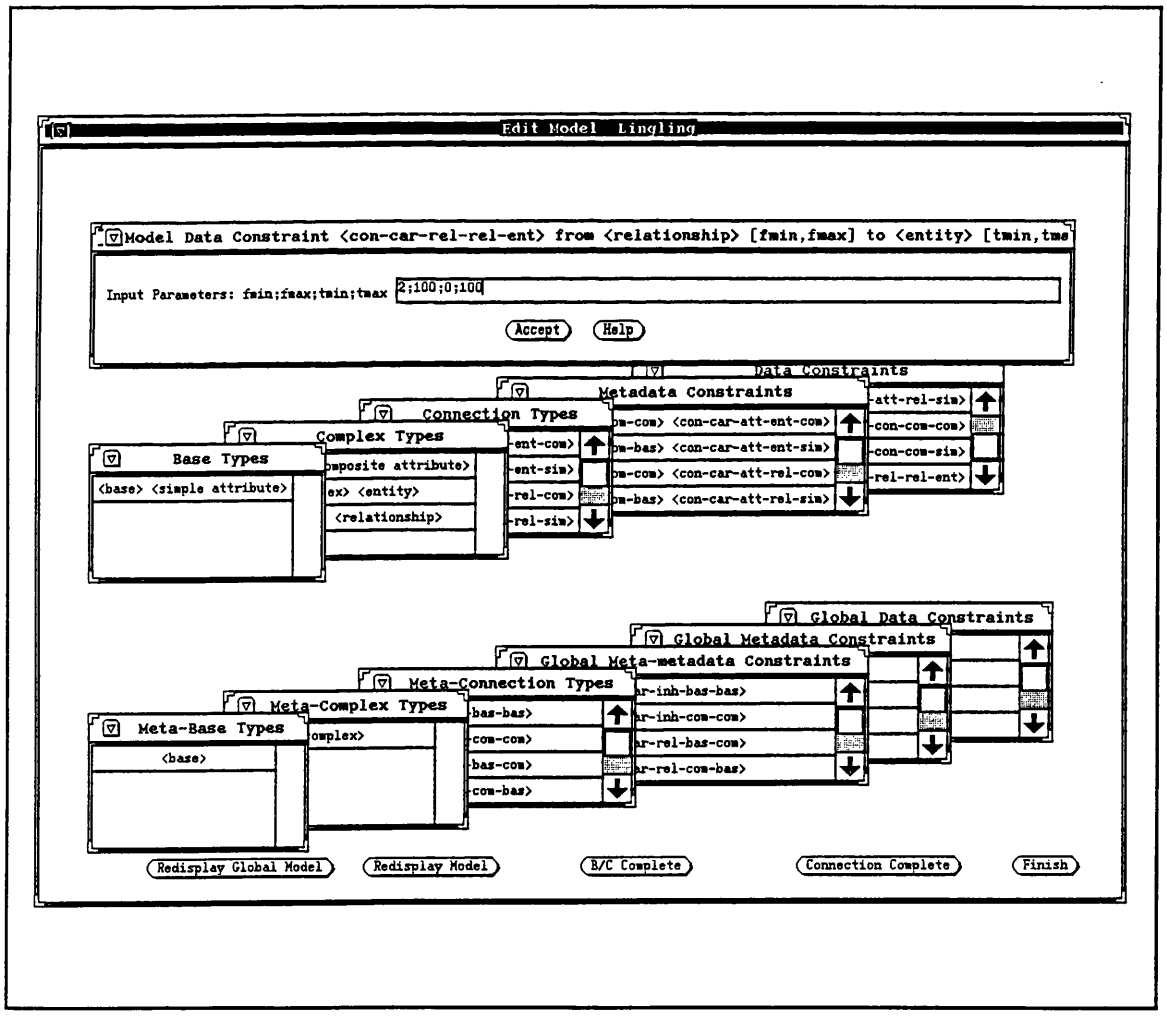

Figure 4.25 Specification of Constraints in a Model

Similarly, when a particular data schema is defined, model inherent data constraints are inherited to become schema-data constraints automatically, while model implicit data constraints need to be instantiated to become schema-data constraints. Therefore during a schema definition process, firstly the appropriate base classes and

complex classes will be defined, secondly the appropriate relationship classes will be defined, thirdly some data constraints will be inherited from model inherent data constraints automatically, and finally some other data constraints will need to be specified using facilities for the instantiation of the model implicit data constraints. As a schema is modified, a similar process will be carried out.

In order to specify a data constraint, the schema designer selects an entry from the data constraint menu, which displays the names of all the model implicit data constraints, and then the system will provide a dialogue to allow the designer to input the necessary information. When all these are complete, data constraints will appear along with other schema structures which constitute the schema; that is, base classes, complex classes and relationship classes.

Figure 4.26 shows a window in which the specification of constraints in a data schema is conducted.

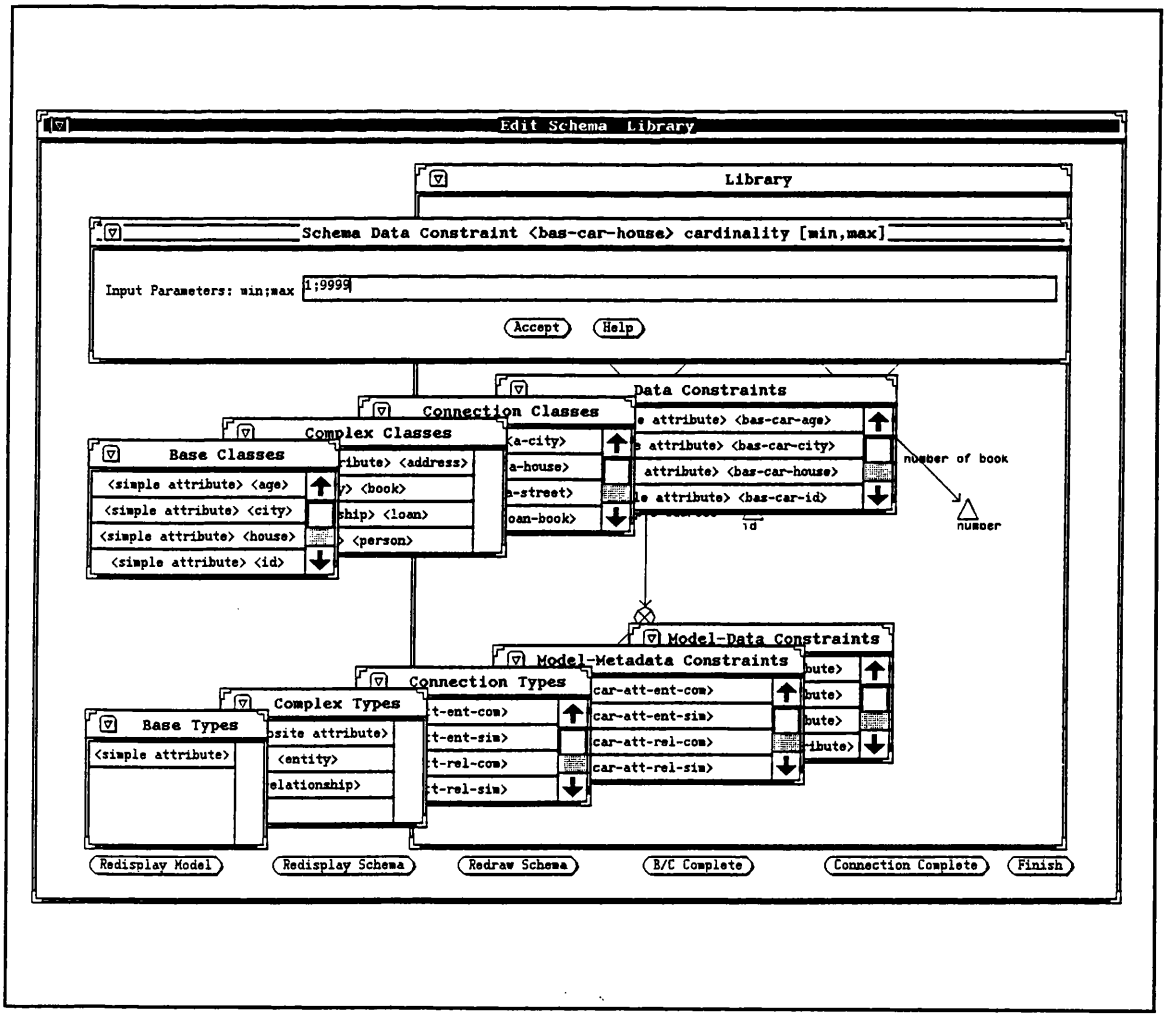

Figure 4.26 Specification of Constraints in a Schema

After metadata constraints and data constraints are stored, they can be used to verify metadata and data respectively. The common treatment is to scan the relevant tables in the context of the relevant environments to detect any conflicts between constraint definition and the real data, including metadata.

4.5 Constraint Management

The management along with the specification of constraints helps to ensure the integrity of the database. Now that constraint representation has been separated from system software and application programs, and has been made visible and manipulable, it becomes feasible to enable the user to decide the point of time when the constraints should be enforced. Moreover, it seems reasonable, even imperative, to allow a certain degree of constraint violation temporarily. For instance, when a person has just been inserted into a database and the process of inserting the relevant attributes is to be completed, the constraints that a person must have an age, a sex and so on should be allowed to be violated momentarily.

The admission of the values which are inconsistent with the constraints that are supposed to hold must, of course, be controlled in some way. It is therefore essential to decide what to do when constraints are violated. In other words, at some stage either the value or the constraint must be altered. An effective mechanism for restricting the time during which inconsistency is allowed is the transaction.

In this section, Subsection 4.5.1 introduces some basic concepts on constraint management; Subsection 4.5.2 describes simple constraint management methods; while Subsection 4.5.3 proposes sophisticated constraint management using transactions.

4.5.1 Some Concepts of Constraint Management

This subsection will clarify some basic concepts regarding the management of constraints within the framework of CDMS.

Access Mechanism and Constraints

In an ordinary DBMS, appropriate **access mechanisms** are provided to permit **data definition** and **data manipulation** operations, which must not conflict with any relevant constraints. The target acted on by data definition is the metadata values that constitute a data schema, while the target acted on by data manipulation is the data values that are contained in a database. Data definition operations and data manipulation operations must therefore abide by the requirements on metadata values of the data schema and data values of the database, respectively. In the CDMS, extra access mechanisms are supplied to permit **model definition** operations. The target acted on by model definition is the meta-metadata values that constitute a data model. Model definition operations must therefore abide by the requirements on meta-metadata values of the data model.

Constraint Verification

Constraint verification, or constraint enforcement is essential in safeguarding the integrity of a database system. Traditionally, some constraints are ensured by the relevant DBMS. For example, a particular attribute of an entity may be declared unique so that verification will be made by the system on the inserted data. Some constraints are ensured by programs which update the relevant schema or data. Thus, the metadata constraints and data constraints will be verified, respectively, when the schema is defined and when data is inserted, removed or updated.

Verification of constraints is vital in terms of integrity constraint management. It must be applied where possible violations may affect the integrity of the database.

Recovery from Constraint Violations

The recovery from constraint violation is as important as constraint verification in terms of integrity constraint management. The DBMS or the relevant programs that handle the constraints should be able to take the necessary actions if any violation is detected.

Recovery actions may take the following forms:

- **forbidding**. The update which causes the violation is rejected right away.

- **cascading**. Further update actions which repair the violation are performed to make the update acceptable. To illustrate this, take the example of deleting the *person* Jean. Because Jean is also a *student*, *student* Jean should be removed at the same time as *person* Jean is removed.

- **suspension** and **re-imposition**. Sometimes, constraint enforcement will be suspended for a limited period. A questionable update is thus allowed to go through. An example of this is the temporary removing of the restriction on the number of books a library member is permitted to borrow. At some stage, however, if the definition of the database is to retain its meaning, the inconsistency between the data value and the constraint must be resolved. A means of identifying the updates that by-passed constraint checks is required to correct these updates whenever it becomes necessary.

4.5.2 Simple Constraint Management

Some time ago, two projects [Tan, 1991; Tai, 1991] extended the IFO data model to allow the specification and management of five kinds of integrity constraint.

An interface was implemented to accept implicit and explicit constraints [Tan, 1991]. The five integrity constraints that were handled were uniqueness, range, non-null, cardinality and general constraints. A general constraint consists of predicates linking constraints and nodes which are related to each other - the linking being performed by the usual boolean operators and connectors. For example: a manager's staff number must be less than 1000.

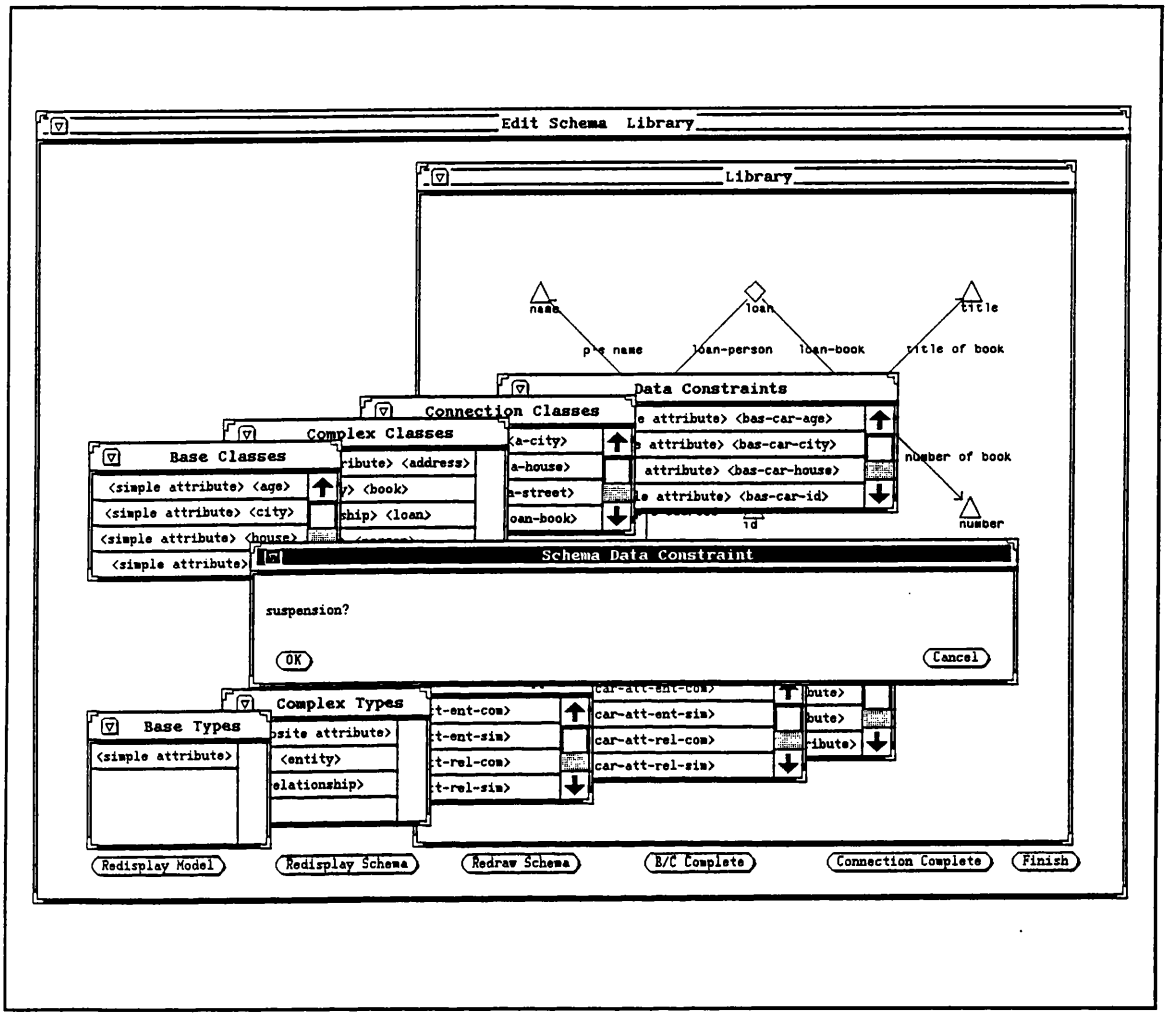

Figure 4.27 Constraint Suspension and Re-Imposition

In the original version of the IFO program [Cooper and Qin, 1990], the two constraint checking components, after alerting the user to a violation, give the option of either to forbid or permit the data. The database is thus able to get into a state which invalidates the constraints. With this version as a platform, the notion of fine control on suspending and re-imposing constraints as the user requires were implemented [Tai, 1991]. Figure 4.27 shows the case.

4.5.3 Constraint Management using Transactions

A more sophisticated method of constraint management is related to the concept of **transaction** [Lee, 1992].

Transactions are traditionally used to group changes together into composite changes which are considered to be atomic. It is therefore feasible to use transactions as a

device to control periods of inconsistency. That is, constraint violations will possibly be permitted until the end of the current transaction at which point they will be reviewed again.

Using transactions makes the constraint management facilities more powerful. Considering that a database system should normally allow transient inconsistent states as a process is running, it is crucial that the process can tell the system whether the integrity constraints should be verified at each particular point of time. The constraint management using transactions aims to provide control on the suspension and re-imposition of constraints as the user desires. For this purpose, the following facilities are essential:

Begin transaction creates:

- a log of changes to be made within the transaction;
- a log of constraint violations which might occur within the transaction;
- a log of constraints to be suspended within the transaction.

Suspend constraints has two forms. One suspends verification of constraints fully; the other suspends verification of selected constraints.

Re-impose constraints also has two forms. One re-imposes all constraints; the other re-imposes selected constraints.

Nested transactions. Transactions are allowed to be nested arbitrarily.

End transaction, which will be explained later.

When data definition or data manipulation is in progress, the user may require a start of transaction, then when any violation is detected, the system will indicate it and ask the user how to deal with it; if the user wishes to undo it, the system will undo it, while if the user wishes to pass it, the system will pass it until the end of the transaction. At the end of a transaction, the system will summarise all violations that occurred within the relevant transaction and ask the user how to deal with them. The choices include:

- **abort whole transaction**. Whatever updates have been made during the transaction will be discarded, with the next outer transaction becoming the current one;

- **allow to next transaction** means that the user wishes to let the violation go through. In this case, the violation will be left until the end of the next outer transaction, where it will be reviewed again, if it is still not rectified.

- **repair violations**. This is where compensating actions are sought by the user. Depending on the constraint that was initially violated, the program will go on to do the necessary repair work. However, in repairing the violation, the same constraint may be violated again or other constraints may be violated if more than one constraint is in effect.

Thus, other than aborting the whole transaction, where the logs disappear right away, all the violated instances in the current transaction will either get repaired or passed on. If the current transaction is the outmost transaction, the contents of the logs will be merged with those of the store. Otherwise, the contents of logs will be merged with those of the next outer transaction.

Only the data which does not violate any constraints should eventually be put into the persistent store. A full treatment of transactions can only be dealt with in the context of application behaviour.

4.6 Summary

Semantic integrity constraints make up an important part of accurate representation of the real world.

A main strength and distinguishing characteristic of CDMS is that it offers powerful means for expressing semantic integrity constraints. There are various types of integrity constraints that may hold. The important question is, however, how the constraints should be categorised and how they should be enforced. Existing systems are typically weak in this area; that is, most integrity verification is still done by user-written procedural code, which is not flexible and imposes an extra burden on the programmer, who must know all the constraints that a transaction may violate and must include checks to ensure that none of the constraints will be violated. Any misunderstanding, omission, or error by the programmer may leave the database in an inconsistent state. In addition, it is difficult to manipulate constraints implemented in this way. There is therefore a need for a mechanism to specify what constraints the meta-metadata, metadata and data concerning the

corresponding data model, data schema, and database must satisfy. It is also beneficial for the relevant constraint issues to be handled centrally by the system rather than by individual application programs. In this way, enforcement will be consistent and difficult to get by. Constraint management using transactions is introduced in this chapter as a potential sophisticated constraint management method.

The constraints existing at the global level are caused by limitations of the CDMS as well as by the intrinsic nature of the global data model itself. That is, while most constraints are intended to reflect the semantics of the global components, some may reflect unwanted limitations in design and implementation of the system.

Once characterised in this universal pattern, firm and consistent control of how the constraints are managed becomes possible. Some possibilities have been outlined for this. One final point needs to be discussed, however, is the issue of efficiency.

The great benefit of programming of constraints as *ad hoc* code fragments embedded in the operations is the obvious advantage that the checking can be localised and made more efficient. However, as the CDMS is implemented in an effective environment (see Chapter 7) this advantage can be carried over. The description in this chapter has emphasised the separation of constraints as separate denotable values. This is the logical view. One of the options which are given at the implementation level is to embed constraint checks wherever needed, possibly guarded so that they can be suspended. Now the constraints appear in the code consistently, but identifiably. This implies that application behaviour is also given to manipulation and it is to this that attention is turned.

5 Behavioural Issues

This chapter is devoted to another important concept in the CDMS, the management of the **behaviour** of values in a database.

Throughout this chapter, the behavioural aspects of a database are described by use of the term **active object**, which represents any kind of value that includes a component which is code to manipulate the database* .

The need to encompass behaviour in the CDMS actually arises from a number of considerations:

1) It has long been a goal of semantic data modelling to encompass the description of behaviour as well as structure.

2) It seems a natural extension to the ability to configure the structure of data to add the ability to configure how the data is managed.

3) The treatment of constraints has brought up the possibility of configuring the constraint management process. This means configuring code.

4) Since many kinds of transaction management have been proposed, it seems sensible to allow transactions to be configurable. Transactions are active objects.

5) Process modelling is carried out in a way which is similar to data modelling, it is therefore sensible to unify the two approaches. Again, processes are active objects.

* Active object used here carries much more meaning than the term usually does.

6) Object-oriented database systems include a behavioural aspect. To permit object-oriented models to be configurable in the CDMS, active objects must be involved in the system. Methods are active objects.

7) Active databases include events and actions, both being active objects.

Essentially, any value including a piece of executable code can be thought of as an active object class, whereas every execution of the code can be thought of as an active object instance. A piece of program can thus be dealt with as a manipulable 'value' in the same way as an ordinary entity. A remarkable feature of an active object instance is that it does not exist statically in the database, but exists dynamically in the system as a running process. However, the code of which this is an instance can be held in the database.

In this chapter, Section 5.1 reviews active objects in the context of semantic data modelling; Section 5.2 analyses transactions as active objects; Section 5.3 gives further examples of active objects by concentrating on a process support system; Section 5.4 describes active objects in database systems; Section 5.5 introduces active database systems, followed by Section 5.6, which summarises the active object survey in the previous five sections; finally, Section 5.7 presents the behavioural aspect of CDMS, followed by Section 5.8, which concludes the chapter.

5.1 Active Objects in Semantic Data Modelling

Originally semantic data models could refer only to **passive objects**, but later models, such as **TAXIS** [Mylopoulous *et al*, 1980], **requirements modelling language** (RML) [Greenspan, 1984] and the **event model** [King and McLeod, 1984], extended the notion of the object to cover **active objects**, such as **activities**, **processes**, and so on. This requirement occurred primarily in the field of office automation, in which the need to refer to certain activities as denotable entities arose. Thus, assessment of an insurance proposal is a process with a certain structure, including constituent activities and associated static objects (information, paperwork etc). It seems likely that working environments without the facility to describe processes will lose favour in the development of complex systems. On the other hand, process modelling, in which processes are described at a high level of abstraction, will gradually become the basis of the design methodology for a more sophisticated system.

TAXIS [Mylopoulous *et al*, 1980] is designed for the creation of interactive information systems. Its main modelling construct is the class, which is used to model both passive and active objects. The latter include expressions, constraints, transactions, exceptions and the like. There is an inheritance hierarchy, in which all classes participate, and an instantiation hierarchy of three levels: tokens, classes and meta-classes, with each token being an instance of a class and each class being an instance of a meta-class. Properties are defined on tokens, classes and meta-classes. These represent single facts, functions from one class to another, and functions from a collection of classes to a single class, respectively.

A special meta-class is called TRANSACTION_CLASS, which contains class objects which are not sets of tokens, but are program objects. Such a class consists of a full specification of the input parameters, local variables, a pre-condition for the transaction to execute smoothly and a list of sub-actions which constitute the transaction's behaviour. Here is an example:

TRANSACTION_CLASS RESERVE_SEAT **with**
parameter_list: *reserve_seat*: (*p*,*f*);
locals *p*: *PERSON*;
f: *FLIGHT*;
i: *INTEGER*;
prereqs
seats_left: *f.seats_left*>0
actions
make_reservation:
insert_object_in *RESERVATION* with
person←*p*, *flight*←*f*;
decrement_seats: *f.seats_left*←*f.seat_left*-1
assign_aux_vars: *i*←*f.seats_left*
returns
rtrn: *i*
end

The transaction's specification consists of the input parameters and local variables, a precondition and the actions which constitute the transaction's behaviour.

Transactions are used to model any active component of the system, including the procedures which define test-defined classes, exception triggers and exception

handlers. TAXIS is a consistent system which uses two mechanisms, inheritance and instantiation, to model the real world. It has some limitations, and some aspects, like exceptions, feel somewhat unnatural to use. However, it is very clear as a specification language.

Requirements modelling language (RML) [Greenspan, 1984] is a by-product of the TAXIS project. It allows the specification of the entities, activities and assertions that may be defined for an application in a syntax that is essentially the same as that of TAXIS.

RML is designed to allow the description of the 'problem situation' rather than of the 'solution system'. Definitions in RML are designed to be statements of what must be included in any program which models the application domain.

The **event model** (EM) [King and McLeod, 1984] is another model which involves active object descriptions. The EM intends to describe a database by making statements on the database which are always true. It therefore needs some notion of active objects to make statements such as 'Event E modified object O at time T'. There are two types of passive object, which are descriptor objects and abstract objects. The former are strings and hold identifiers and printable values, whereas the latter are complex objects consisting of attributes, of which one (the primary attribute) defines the object uniquely. Attributes are modelled by functions and can be specified to be unique, single-valued, non-null, exhaustive or the inverse of another attribute. Objects may be sub-typed, by using restricting predicates or adding more attributes. Events are divided into application events and perusal events. These model transactions and queries respectively. They may be parameterised and require the specification of objects to be used in the event and the sub-actions involved.

Since the CDMS has evolved from a semantic data modelling perspective, these kinds of 'active' data model show the way forward to extending the CDMS into the active dimension. The sorts of construct provided by RML and the EM must be instances of the generic model which underpins the CDMS.

5.2 Transactions as Active Objects

5.2.1 Database Transactions

An **atomic database transaction** is a logical unit of database processing, or an execution of a piece of code which performs database access or database update operations, retrieving or changing the contents of the relevant database. Transactions are typical of active objects found in traditional database applications. Because a transaction groups a number of changes to the database into a single atomic unit, it enables a DBMS to supply facilities which handle changes of varying levels of complexity in the same way. For instance, transactions can be used in such a manner that concurrency control is properly organised so that the competitive changes may vary in extent and be combined without interfering with each other.

5.2.2 Transactions and Concurrency Control

The transactions submitted by the various users may execute concurrently and may intend to access and update the same database elements. If this concurrent execution is uncontrolled, it may lead to problems such as an inconsistent database [Papadimitriou, 1986].

In order to avoid this, transactions have four important properties; that is, atomicity, consistency, isolation, and durability. These properties are also referred to as ACID [Harder and Reuter, 1983].

Atomicity means that either all of the constituent updates get executed or none of them do.

Consistency means that transactions preserve the integrity of the database; that is, the overall effect of a transaction must not result in a database state which violates integrity constraints. A transaction always transforms a consistent state of a database into another consistent state of the database, but it does not necessarily preserve consistency at all intermediate points.

Isolation means that transactions do not interfere with each other. Even though there will generally be many transactions running concurrently, any particular

transaction's updates are hidden from the others, until that transaction commits. In other words, for any two distinct transactions T1 and T2, T1 might be able to see T2's updates after T2 has committed or T2 might be able to see T1's updates after T1 has committed, but certainly not both.

Durability means that once a transaction commits, its updates survive, even if there is a subsequent system crash.

The main mechanism which ensures that when multiple transactions are submitted, they do not interfere with each other so as to produce incorrect results is **concurrency control**. Concurrency control ensures that a transaction is either performed in its entirety or is not performed at all (atomicity), a transaction should not make its updates visible to other transactions until it is committed (isolation), once a transaction changes the database and the changes are committed, these changes must never be lost due to subsequent failure (durability), and an execution of the relevant transactions takes the database from one consistent state to another (consistency). Another mechanism, the **recovery system** is essential for handling transaction failures. This will ensure that for every transaction that has been started, either all of its updates will be committed or they will be completely undone with the relevant transaction being rolled back.

5.2.3 Transactions in Relation to Constraints

As has been presented in the previous chapter, another vital use which a DBMS makes of transactions is the management of **constraint violation**. Given that it is sometimes necessary or convenient to violate enforced constraints temporarily, the transaction is used to limit the scope of the violated condition. A sophisticated control of violation is made possible by offering the user a mixture of options, including to abort violating updates, to cascade the effect of an update, and to allow the violation to remain until the end of transaction [Lee, 1992]. Transactions can be nested. In this case, violation may be permitted until the end of the outmost transaction.

Alternatively, constraints can be embedded in transactions in the form of pre-conditions, triggers or post-conditions [Mylopoulous *et al*, 1980]. Accordingly, updates are made only in the context of a particular state, or in order to cause repairing actions, or after their effects are seen to be acceptable. In any case, the effect of updates has been limited by the existence of constraints. Thus a transaction in this kind of system is

composed of a mixture of updates and constraints. This issue will be further discussed in Section 5.5.

5.2.4 Conclusion

From the above discussion, it is clear that transactions constitute a significant category of the constructs which a database system uses. They are used for several critical facilities and are expected to make the coherent use of a DBMS more straightforward. They also came in a variety of flavours, each of which is appropriate to certain circumstances.

Therefore, in configuring a database application, it would be preferable to configure also the transaction model. In order to achieve this uniformly, a generic model which encompasses transactions will be required.

5.3 Process Support System

An application in which active objects play an important role is **process modelling**.

The modelling of processes within an enterprise is becoming a more prevalent activity. By creating computer models of its activities, an organisation can expect to find shortcomings in the current ways of doing things and also create cost-cutting regularity across the organisation. In some cases, this modelling can be used as the basis for automating all or part of the processes.

Process modelling is carried out in a way which is very similar to that of data modelling. Simple graphical or textual languages are used to describe the processes [ProcessWise, 1993]. Typically, there may then be some way of enacting the processes with simulated data to watch how the elements of the schema behave. Given that the same organisation is likely to have a database holding the real information which the processes are involved with, there is clearly some benefit in tying the process schema and the database together, but this is rarely done and even if it is done, it usually involves some *ad hoc* means of communicating between the two sub-systems. One of the promises of the CDMS is the ability to create database systems and process modelling systems in the same

coherently organised environment. This should improve the quality of process modelling significantly.

This section presents a **process support system** (PSS) developed at ICL [Bruynooghe *et al*, 1991], which illustrates the main ideas of process handling. Both industrial and business systems can be simulated in PSS.

From the process support point of view, process treatment can be divided into four stages; that is, process capture, process modelling, process enactment and process control.

- **Process capture** produces a process description, or a static representation of a certain process. The significance of process capture is that this enables initial assessment of the process and hence makes it possible to improve the effectiveness of the relevant treatment.

- **Process modelling** means formalisation of the process, which enables further analysis to be carried out. Necessary changes can thereafter be designed and implemented, resulting in the process being optimised. Process modelling is an essential precursor to process automation.

- **Process enactment** means the automatic execution of a process application which guides users through the process and provides users with access to their required tools and data through a consistent, intuitive user interface. At this stage, the computer takes charge of simulating the whole process and also allows alteration of the process as needed.

- **Process control** is the ultimate stage of process support. As a result of thorough experimentation in the stage of process enactment, an optimised process control system can be designed and implemented to run effectively and efficiently.

The PSS developed at ICL is a process enactment system* . Its kernel is a **process control engine** (PCE), which provides the appropriate working environment for process simulation at each point of time. The PCE is programmed in **process management language** (PML), which is used to describe the objects upon which the

* PSS was implemented in the same language, PS-algol, which was used in the early stages of CDMS development.

user might carry out some operations, the type of interaction the user has with the system, the tool to help the user realise those operations, and the means of communicating changes to the working environments of the user.

The basic PSS concepts include process, role, activity and interaction. A **process** consists of a set of roles, each of which serve particular goals in an executing process. A **role** may be subdivided into a number of activity threads which may proceed concurrently. A **thread** of **activity** is actually a sequence of activities, and an **interaction** between roles is represented by a connection between an activity in one role and an activity in another.

As the central factor of process support, process modelling can be realised by two approaches, or more precisely, two stages; that is, abstract modelling in **role-activity diagram** (RAD) and detailed modelling in PML. A process schema can thus be embodied in a diagrammatic form or in a program form. The PSS supports process description in PML and process enactment by a sophisticated user involved animation interface.

5.3.1 Role-Activity Diagram

This subsection is devoted to exploring the approach of process modelling using a RAD. This approach is used to help understanding and reasoning about processes.

Some features of a process in the real world can be described graphically in a **control flowchart** (CF) or in a **data flow diagram** (DFD). The former can express features such as the order and decision points of the activities, while the latter, without this ability, is able to express features such as the movement of data, its external sources and sinks, and the files or databases which hold the stored data.

An ideal process modelling tool should, however, be able to model the essential properties of a process intuitively and with sound underlying semantics, and its primitives should correspond to real-world actions. An ideal process modelling tool should also be able to model the goals, to model the business rules, to model how people achieve the goals, and to model how people interact to get the job done collaboratively in their different roles.

Unfortunately, neither a CF nor a DFD is able to model the goals, that is, what the organisation is trying to achieve; neither CF nor DFD is able to model the business

rules, that is, the limits the business places on people; neither CF nor DFD is able to model how people interact with each other.

In process modelling, what people do is represented by roles and activities. Goals are represented by post-conditions of activities. Business rules are represented by the logic between activities and roles. How people interact is represented by interactions between roles.

Drawing up a process schema as a diagram helps in the design of a new process, because the weaknesses in an existing process can thus be identified and possibly eliminated. Expressing the schema diagrammatically encourages shared understanding of the existing process, which then helps to ensure proper organisational operation, coherence and completeness in a set of process.

In a RAD, the process is drawn up as a set of roles. The activities within each role, which operate on data entities, are shown with their ordering and logic. Interactions between roles are shown at the appropriate place in the logic.

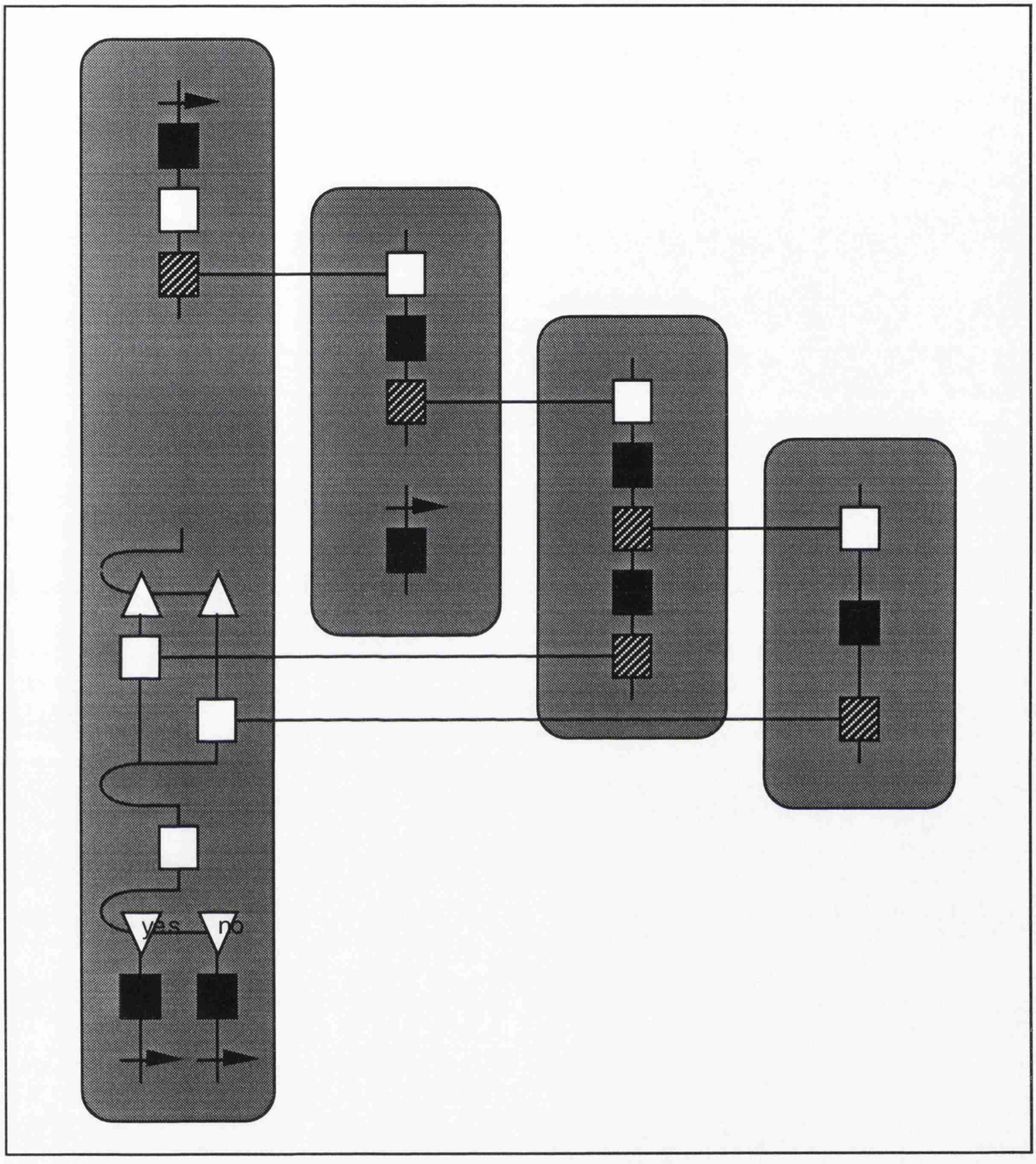

Figure 5.1 A Sample RAD

As an example, Figure 5.1 shows an unlabelled RAD for a process involving four roles. A role is a grey box in the diagram in which threads of activity are shown as vertical chains. Each chain shows a series of states changed by the activities. The first role has two threads of activity which initiate and complete the process. The initiating thread has three activities, the last of which starts a thread of activity in the second role. This in turn starts threads of activity in the other two roles which eventually feedback to the completion thread of activity of the first role. In the diagram there are parallel sub-threads branching from the main thread which mean concurrent processing for the activities which reside on each of the parallel sub-threads, and there are also alternative sub-threads

whereby which sub-thread follows the main thread depends on the branching condition. The lines joining two activities belonging to two different roles indicate the interactions between roles.

A RAD distinguishes various kinds of activity which may be included in a role. The main kinds are shown in Figure 5.2. The basic activities distinguished are black boxes which require user involvement and white boxes which do not. By composing these with conditional and parallel sub-threads, as well as interaction links, processes of arbitrary complexity can be modelled.

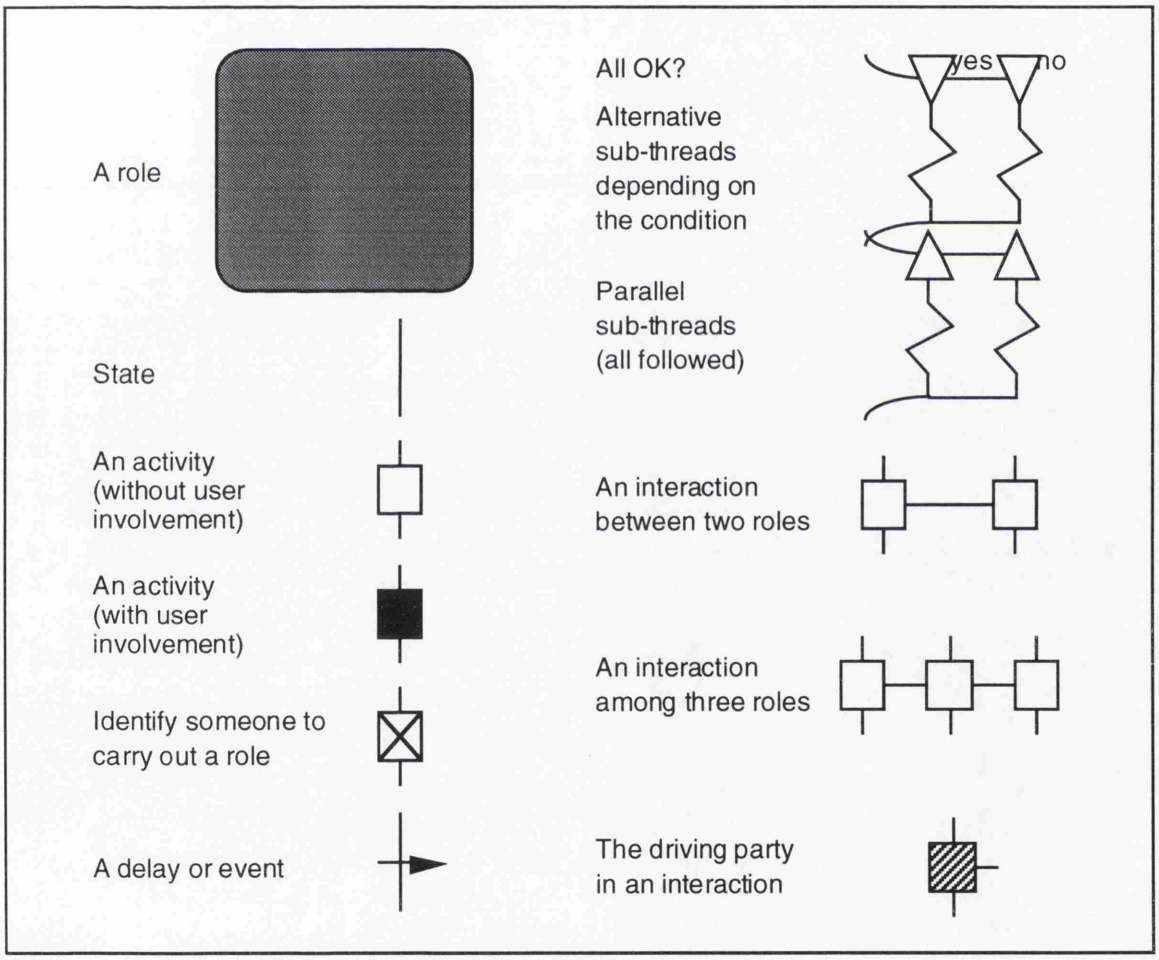

Figure 5.2 RAD Notations

5.3.2 Process Management Language

The Process Management Language (PML) is used to formally describe or program a process in the context of the PSS, PML is thus central to the PSS.

Six aspects of PML can be identified which help its users to encode processes:

- **Concurrent threads of execution**: Concurrent threads provide a convenient abstraction that simplifies the process design task, by enabling each thread to concentrate on a particular sub-task.

- **Dynamic thread creation**: The dynamic creation of new threads captures the unpredictability with which new activities can start within a process.

- **Subtyping**: The support for subtyping allows operations to be defined that can be applied to data objects of different but related types.

- **Persistence**: The persistence provided by the PSS environment relieves the programmer of any concern as to whether program data is held in primary or secondary storage.

- **User interface**: PML allows the user to define a simple relationship between the state of a process thread and the display its owner sees.

- **Tool interface**: The ability to start and then transfer data to and from tools that are running outside the PSS allows the process program to coordinate their operation with the state of the process.

PML is a strongly typed language, with a class inheritance structure. The class hierarchy supports three kinds of classes: entities, actions and roles. **Entity** class definitions create record types, **action** class definitions introduce procedures, and **role** class definitions act as schemata for subsequent execution thread creation.

A role class definition defines both the data values and the behaviour of its instances. The data properties are defined in the resources category. Although PML is not an object oriented language in the usual sense, the role is an object encapsulating its composing actions and responding to external stimuli or messages, namely, the **interactions** it has with other roles. One role can only communicate with another by sending it a message. Using roles as data encapsulators both simplifies the programmer's task and makes it less error-prone.

A sample process coded in PML is shown in Figure 5.3. This process involves two role instances 'role1' and 'role2', both of which belong to the same role class

'SampleRole'. Each role instance can accept an integer value from the screen or from the other role instance, and show an integer value on the screen or send it to the other role instance when there is one in the relevant role instance.

```
classes
    SampleRole isa Role with
    resources
        value: String
        gp: giveport String
        tp: takeport String
    actions
        getValue:
        {GetNew(agendaLabel='supply value', object=value)}
        viewValue:
        {ViewResource(agendaLabel='show value', object=value)}
        when nonnil value
        sendValue:
        {UserAction(agendaLabel='send value'); Give(interaction=gp, gram=value)}
        when nonnil value
        receiveValue:
        {Take(interaction=tp, gram=value)}
    end with

resources
    role1: Role
    role2: Role
    gp1: giveport String
    tp1: takeport String
    gp2: giveport String
    tp2: takeport String

actions
    startSampleRoles:
    {UserAction(agendaLabel='Start Sample Roles');
    NewInteraction(giver=gp1, taker=tp1);
    NewInteraction(giver=gp2, taker=tp2);
    StartRole(roleClass=SampleRole, agendaLabel='Sample Role 1', roleInst=role1, gp=gp1, tp=tp2);
    StartRole(roleClass=SampleRole, agendaLabel='Sample Role 2', roleInst=role2, gp=gp2, tp=tp1)}
```

Figure 5.3 A Sample PML Program

5.3.3 Process Support Environment

Using common software tools from outside as needed, PSS does all computerised work in process handling, and guides its user to do the rest. The roles of a process can be divided into on-line roles, which do not need any resources other than PSS itself, and off-line roles, which involve PSS users and the software tools existing outside PSS. Without a PSS, a user would only be able to accomplish particular functions using separate pieces of software. These functions roughly correspond to the roles, and the user would have to accomplish most interactive work among these functions. In PSS, owing to the existence of the tool agent which, as a special role of the corresponding PCE, deals

with the utilisation of the outside software tools, all available software tools can be utilised in a more convenient and more consistent way. The user agent is another special role of PCE and it deals with all matters which need users to intervene when the process is running. Interaction may occur either between two on-line roles, or between one on-line role and one off-line role.

The PSS has a **distributed architecture**. The roles of a particular process are all executed by a single PCE, while each workstation executes an instance of a UI server, providing a window management capability that reflects at least one instance of a tool server that runs in the environment in which the tools executes. The **user agent** and the **tool agent** provide appropriate interfaces to the servers in the outside world.

As mentioned above, the user agent enables the user to access the PCE from a user server. Each user agent therefore identifies an actual user by name and password. Those roles that share the same user agent are said to be owned by the user that the agent identifies. The user agent sends these roles out to the UI server and enables the user to participate in the process program.

Parallel to the effect of the user agent, the tool agent offers access to the environment in which its associated tool server is executing. It allows a tool to be started and data to be given to and taken from these tools.

A PCE runs in the **process support environment** (PSE), which is a persistent working environment and is able to support processes existing for any length of time. The time within which a process exists equals to the length of the lifetime of what is enacted by the process. It may take several years or even longer, so a persistent environment is needed.

In practise, the command 'runpss' starts a PCE, while the command 'xpssui' starts a user interface server. Once these commands have been issued, a role agenda window will be displayed, and action agenda windows will be displayed for each active role. Figure 5.4 shows an action agenda window. The user responsible for a particular role should accomplish actions by clicking on the relevant action agenda window, then appropriate dialogue widows will be automatically provided to allow information exchange between the user and the system. Figure 5.5 shows such a window.

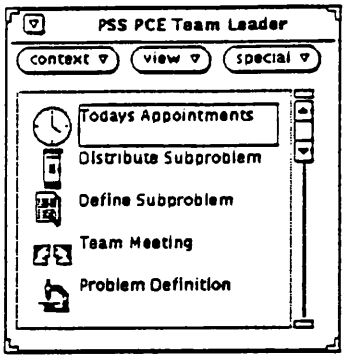

Figure 5.4 An Action Agenda Window

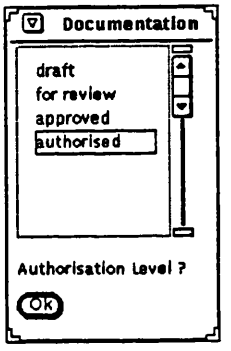

Figure 5.5 A Dialogue Window

5.3.4 Implementing a Process Model

To be sure that the ideas discussed here coincided with the general philosophy behind the CDMS, it was decided to embark on an implementation of a process model using similar techniques to those used for the IFO implementation [Qin, 1993].

The implementation was started by an MSc student [Lum, 1992], but this was enhanced to enable a user to design a process model using a RAD then carrying out their transformation into outlines of program in PML.

The success of this implementation created confidence that Process Modelling and Data Modelling could be combined since many of the underlying techniques are, in practice, the same.

5.3.5 Conclusions

Process modelling, being an increasingly valuable technique for managing complex organisations, is similar to data modelling in many respects. PSS makes process modelling a database application by adding persistence and data models which are primarily about changes to information. One weakness of current process modelling software is that it exists in isolation from the actual data being managed by the organisation, for instance, payroll, orders, etc. It would make process modelling more powerful if the descriptions of processes and the descriptions of the company's data could be unified.

Again this implies that a uniform generic model covering various kinds of process model and the static data that the company uses would be valuable.

5.4 Computation in Database Systems

In order to permit the computation which underlies a database application to be configurable, there is a need to describe the code at some high level of abstraction.

When a user is interacting with a database application, the following code fragments will probably be needed:

- code implementing the DBMS itself, which includes storage methods, concurrency control and the like;
- code transforming user actions into database storage and retrieval, for instance, a query processor;
- queries and updates framed in a query language;
- canned queries or application programs written or embedded in a high-level language;
- code embodied in 'active objects' if the data model is powerful enough.

Considering the action of requesting a balance at an autoteller. The function is achieved by:

1) Canned query code which creates a query using code written in the high-level language;

2) The query is processed and this calls various of the underlying mechanisms.

Ultimately, all of these computations map down into operations which create, destroy, update and display data or metadata. These operations come in different flavours - different access methods and different user interactions for instance - but they are all intended to achieve actions from this small set. So it is useful to create a characterisation of these operations.

A data definition operation should be taken as an active object class at the data model level, each instance of which will affect the relevant data schema in the way that the operation defines. For instance, *create a composite attribute* should be defined as an active object class in the ER data model, while each instance of this, namely, an execution of the relevant procedure, yields a concrete composite class in the relevant data schema, for example, *loan* in an ER schema.

A **data manipulation operation** should be taken as an active object class at the data schema level, each instance of which will in turn affect the relevant database in the way that the operation defines. For instance, *create a person's name* should exist as an active object class in the ER schema shown by Figure 5.6, and each instance of this, namely, an execution of the relevant procedure, yields a concrete attributing instance in the relevant database, for example, an attributing instance which is directed from the appropriate person instance to name instance *'Jean'* in Figure 5.7.

When using a data model and a schema, these operations are implicitly available. They are installed as specialisations when a data model is implemented or a schema is designed. The form they take is not manipulable. Thus, a designer can determine that a *book* has a *title* and a *number*, but cannot control what form *create a book* takes. One of the gains of CDMS is to allow the operations to be manipulable.

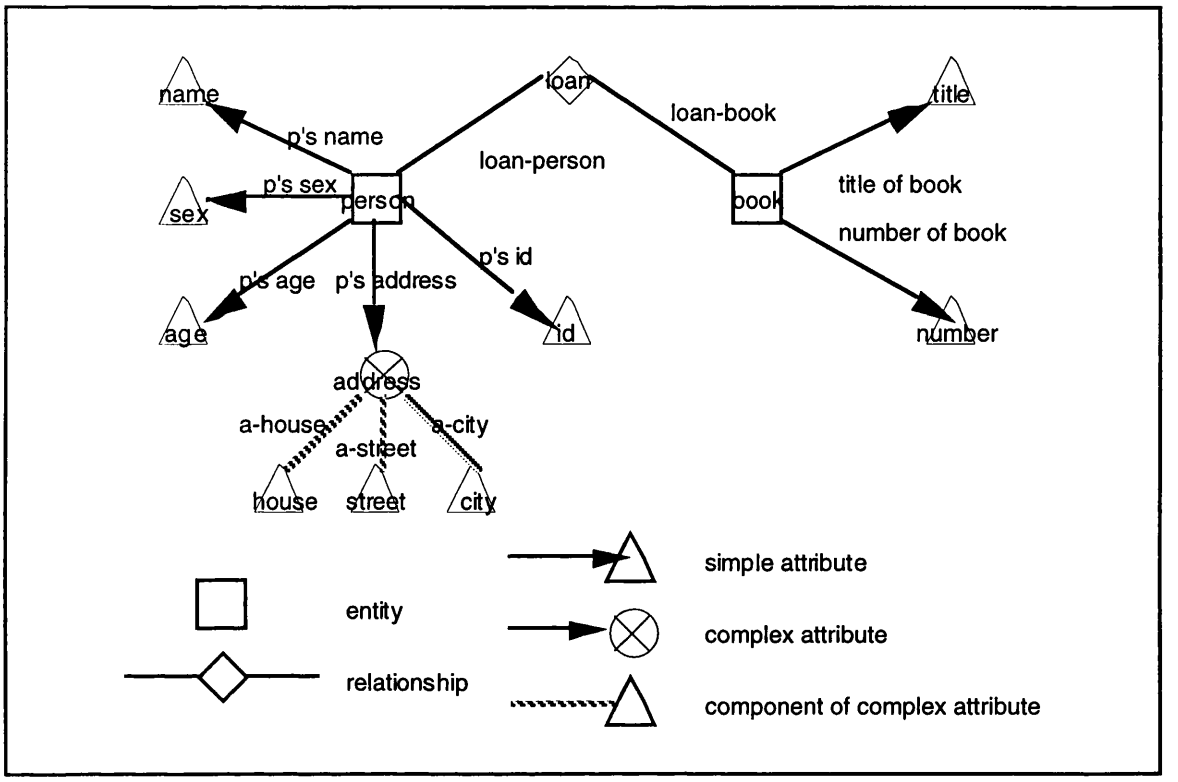

Figure 5.6 A Library Schema

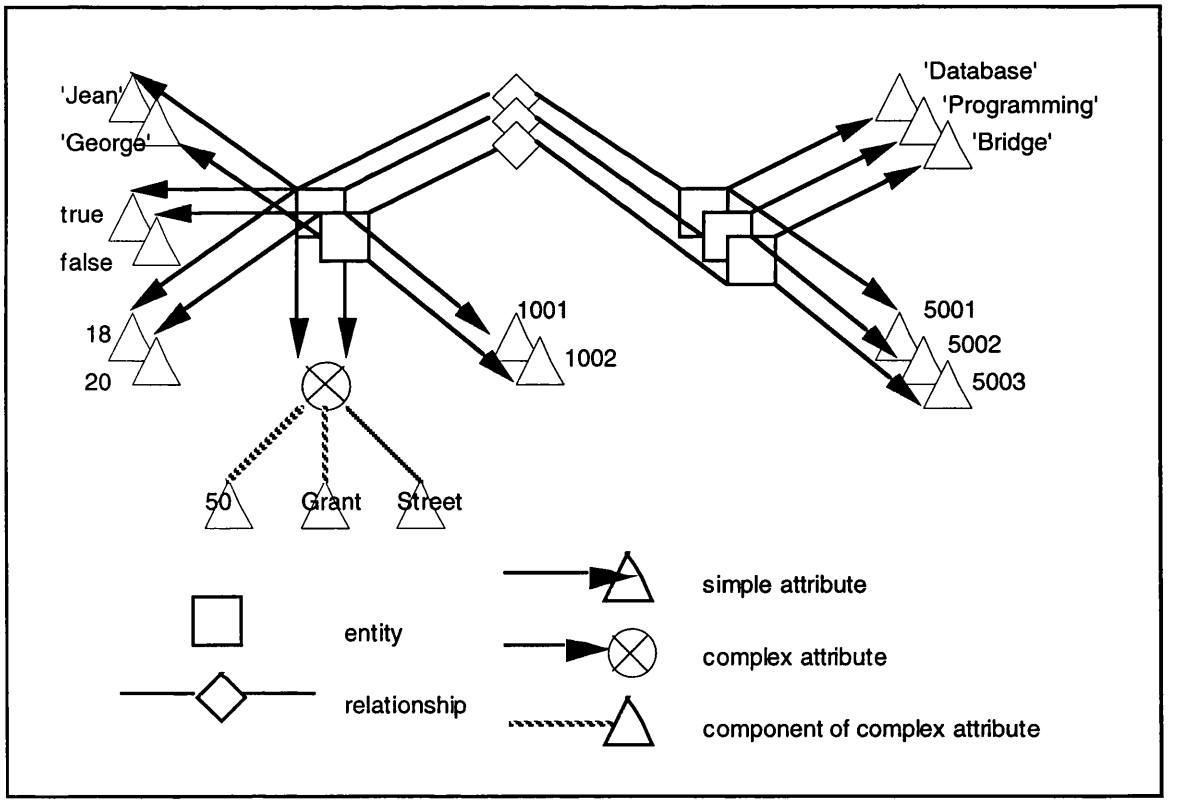

Figure 5.7 A Library Database

In order to allow the configuration of the operations in a particular end-user application on a database, it will be necessary to include the operations above in the generic

model. In configuring an interface, an important step will be to select which operations are exported to the user. The CDMS should be able to do this in a uniform manner to the rest of the configuration task.

5.5 Active Database Systems

A database system was conventionally thought of as a store room that keeps the information which is relevant to an application and which is accessed by user programs or through interactive interfaces. Now database systems are extensively used in the areas which require highly stringent performance as well as the ability of both substantial data storage and complicated information processing, and the traditional multi-component environment has proved to be insufficient. This has led to database research in the direction of more functionality being supported within the database system itself, and has resulted in database systems with more comprehensive facilities for modelling both the structural and the behavioural aspects of an application.

An **active database system** [Paton *et al*, 1994] supports mechanisms which can respond automatically to the events occurring either within or outside the system. Different proposals for active database systems have been made and various applications for such systems have been suggested.

A conventional DBMS is passive in that system commands, including query, update and delete and so on, are executed only upon being requested by the user or application program. This pattern is, however, unable to model some situations appropriately. Consider a flight database which is accessed by different terminals. It is desired to add extra flights when the number of spare seats available a fortnight in advance falls below some threshold value. In order to deal with such situations in a passive database environment, one approach is to add this function to all booking programs, while another uses a polling mechanism which periodically examines the number of seats available. The former, however, causes the functions to be distributed, replicated and hidden among different application programs; while the latter incurs the difficulty of selecting an appropriate polling frequency. An inappropriate frequency may in turn cause either a slow reaction, or an expensive cost, both being unacceptable.

On the other hand, an active database supports such an application by moving the reactive behaviour from the application or polling mechanism into the DBMS. There it is able to monitor and react to specific circumstances. In this way, the reactive semantics

become both centralised and handled in a timely manner. For this purpose, an active database system offers a description mechanism.

A common approach for the description mechanism uses **rules** which may have the following components:

- **event**, which describes an occurrence to which the rule may be able to respond;
- **condition**, which examines the context in which the event has taken place;
- **action**, which describes the task to be carried out by the rule if the relevant event has taken place and the condition has evaluated to true.

A rule with all three of the components is referred to as an **event-condition-action rule** (ECA-rule). Most active database systems support ECA-rules.

In some proposals the event or the condition may be either missing or implicit. A rule without an event is called a **condition-action rule**, or **production rule**; while a rule without a condition is called an **event-action** rule, or **trigger**.

Clearly rules add significantly to the power and clarity of a DBMS. Behavioural aspects of the applications can thus be specified in a straightforward manner. The CDMS is therefore intended to manage rules. It should be able to achieve this given that there are constraint and behavioural constructs which can be configured to create the application semantics.

5.6 Summary of Behavioural Issues

The previous five sections have examined five different sorts of database perspective in which a behavioural component plays an important role.

Semantic data models aim for a more abstract description of activity. **Transactions** are composite active objects in which data definition and data manipulation operations are clustered and which represent complex changes to the data. **Processes** in process models are simulations of real-world dynamic systems in which primitive actions are organised to mimic the activities involved. Both transactions and processes can thus be

seen as structural concepts equivalent to 'set' or 'aggregate' in the structural modelling component of the CDMS. **Database operations** are the fundamental behavioural elements of the application. **Active database systems** support mechanisms to respond to events which take place inside or outside the database system.

The rest of this chapter presents a proposal for extending the CDMS with behavioural values. This extension is uniform with the preceding chapters and promises a clear method of incorporating functionality with structure and constraints into the configuration process.

5.7 The Behavioural Aspect of CDMS

In order that CDMS concepts function properly, there needs to be some way of bringing these unrelated elements into a common structure. In the CDMS, it is aimed to unify the handling of the code with the management of data structure.

The CDMS global model includes a number of basic actions to manipulate instances of the global model structures and to interact with the user. The global model further allows constructors with which to put these primitive actions together, both with other actions to form complex actions and with data values to form active values. Finally, there are frameworks in which the actions can be embedded to produce interface templates for user interaction.

In the context of the CDMS design and implementation, the main issues relating to active objects are how active objects should be categorised and how configurability should be performed within the system.

5.7.1 Categorisation of Active Objects

Active objects in the CDMS can be categorised into base actions and complex actions.

Base actions are atomic actions in the system and can be subdivided into user involved actions and non-user involved actions. **User involved actions** are logical actions which transmit information to or from the user. They consist of actions such as *input an integer*, *output an integer*, *input a string*, *output a string*, *select an integer from a*

set of integers, *select a string from a set of strings*, and so on. Each user involved logical action may have multiple concrete actions instantiating it. This instantiation activity is the role of the user interface component of the CDMS. **Non-user involved actions** include such actions as *modify an integer value*, *retrieve meta-relationship type directory*, *construct a relationship type*, *store a relationship type*, *initiate a process*, and so on.

A **complex action** represents a combination of a number of actions, which may include other complex actions as well as base actions. There are four combination constructors, which are the sequential constructor, the parallel constructor, the conditional constructor, and the repetitive constructor. The **sequential** constructor takes a number of actions and creates one which is the sequence of these actions. The **parallel** constructor takes a number of actions and creates one which executes them in parallel. The **conditional** constructor takes a number of actions and creates an action in which one of the component actions will be executed according to the matching test result. The **repetitive** constructor takes one action and creates an action which executes it a number of times.

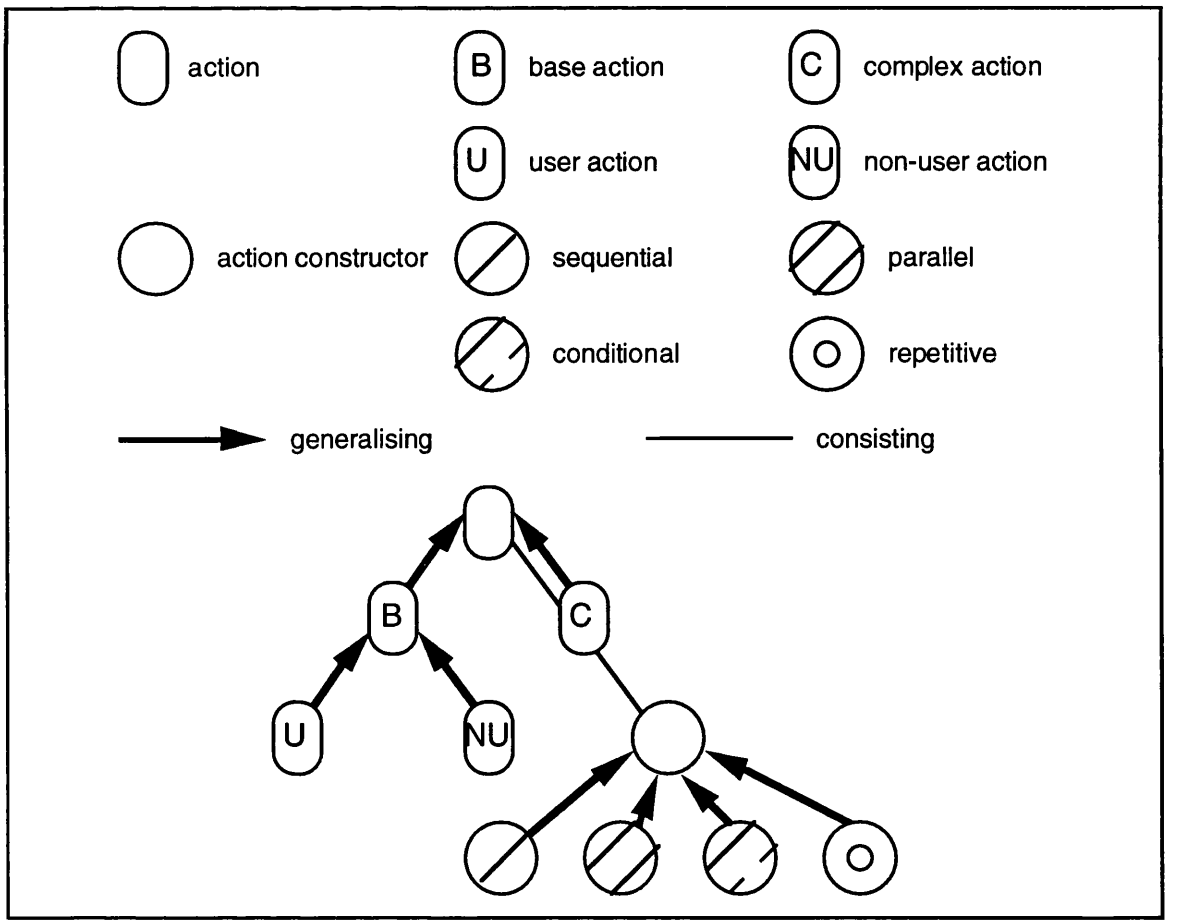

Figure 5.8 Active Objects in the CDMS

Figure 5.8 shows the global view of the active objects in the CDMS. In the diagram all of the action types described above are shown in a generalisation hierarchy. At the top of this hierarchy is an empty action symbol which denotes any unspecified kind of action. The diagram indicates that an action is either a base action or a complex action and that a complex action is made up of a constructor and some actions.

As a model is defined, a base action can be specialised from a meta-base action, and a non-base action can be specialised from a meta-non-base action.

5.7.2 Examples of Configuration of Active Objects

Example 1

In order to manage constraints in a data model, the concept of transaction can be configured and included in the relevant model:

transaction:=sequence of (action, if constraintsOK then commit else abort)

This process can also be displayed graphically, as in Figure 5.9.

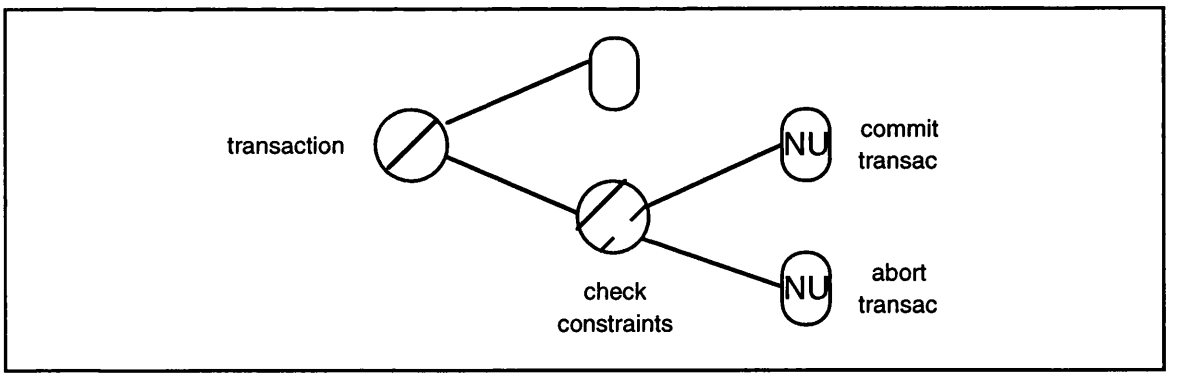

Figure 5.9 Transaction

Here a transaction is abstracted as some action (probably a complex action) followed by a test resulting in either commit or abort. The process of creating a transaction is similar to that of creating a record out of its components.

Because a transaction is itself an action, the case of nested transactions is covered by the above definition.

Having designed the transaction model, it becomes possible to add operations for handling transactions or enhance previously defined operations to manage them.

Example 2

In order to support business processes, such action types that represent thread of activity, role and process can be configured accordingly:

process:=set of (role)
role:=set of (thread)
thread:=sequence of (action)
Figure 5.10 shows the graphical equivalents of the above.

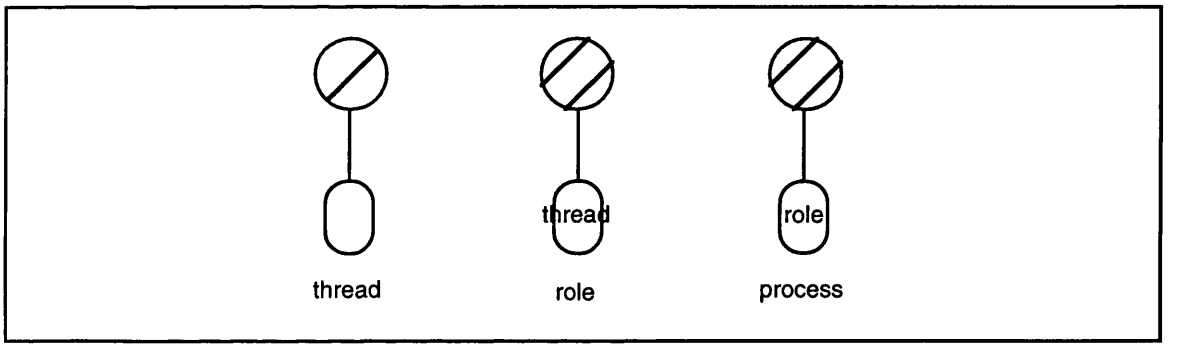

Figure 5.10 Thread, Role and Process

That is, a process is a set of roles, each being a set of threads, each in turn being a sequence of activities. It is obvious that all these are special actions in the context of the CDMS which supports active objects.

Example 3

An example of active schema is shown in Figure 5.11, which represents model manipulation of the CDMS itself, without including the constraint processing part for simplicity. Model manipulation is shown as the conditional execution of one of create, edit and remove operations on models. *Create*, *edit* and *remove* leads to the repetition of operations to create, edit and remove CDMS values respectively.

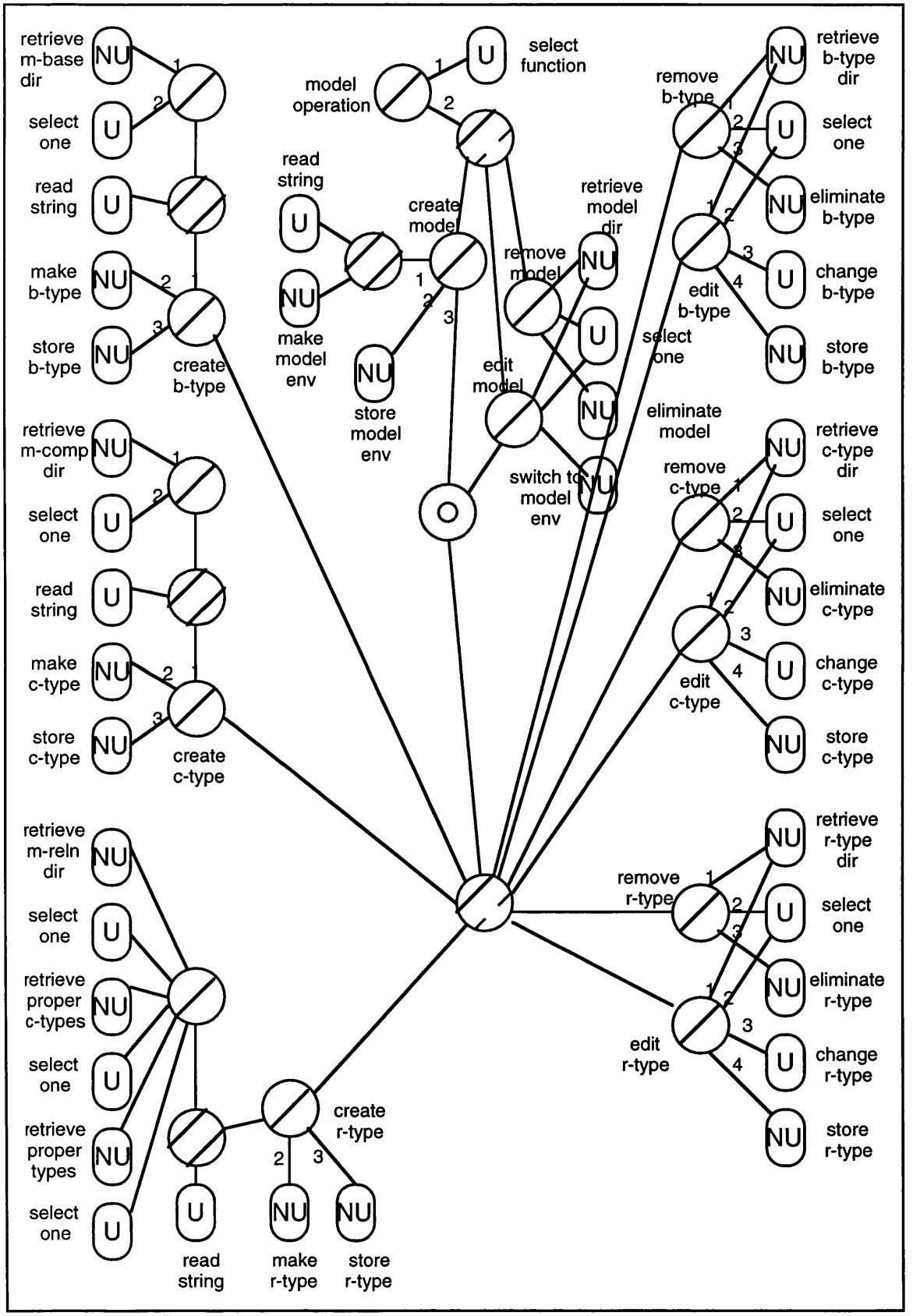

Figure 5.11 Model Manipulation Schema in the CDMS

5.7.3 Using the Configured Functionality

In order to make the operations available to the user, some frameworks need to be offered in which to construct access to the relevant actions. Although these could themselves be built out of the basic actions, a considerable amount of repetitive work will be saved if they are provided as completed components, which may include:

- menu-based framework - the operations constructed as complex actions are slotted into a hierarchical menu framework - the user interface component of the CDMS then turns this into a textual or graphical menu interface;

- graphical framework - the operations appear as manipulating the screen representation of an underlying value graph;

- form-based framework - the operations are tailored to the use of forms for data input and output and also for schema design (in the manner of Freeform [King and Novak, 1987]);

- textual language framework - the framework consists of a template for an interpreter for a textual query language and slots for placing the complex actions - the user interface component of the CDMS then creates a syntax for the language [Cooper, 1994].

When delivered to the user, an interaction facility will comprise a set of available operations built into one of the frameworks outlined above. The approach is influenced by FaceKit [King and Novak, 1989].

The process for constructing an interaction facility starts by choosing the framework and then selecting from the set of modelling primitives which are appropriate for the structures selected to be in the data model. Thus if the base type *simple attribute* instantiated from meta-base is included in a data model, then the global *createBaseValue* will be instantiated as *createSimpleAttribute* and included in the set of data manipulation operations of the model. The relevant details of how the operations are built, combined and embedded into a query language interpreter are give in [Cooper, 1994], while a brief description of the same can be found in [Cooper and Qin, 1994].

5.8 Conclusions

Thus it has proved possible to create a small set of descriptive elements of active objects equivalent to those used for structural features of data items. By putting these global active objects into the CDMS, the system can support the creation of a wide range of systems which include active objects. This will include both databases which contain active values and data managers which have different kinds of operations, for instance it should be possible to tailor a variety of transaction models. All of this can be configured using the menu-driven style of the rest of the CDMS.

Implementing models with active objects in a language which has first class procedures is greatly simplified. Launching an active object is straightforward to program when it becomes possible to retrieve procedures from structures and then execute them. Similarly, providing mechanisms for making sub-activity calls is easier.

6 The Platform for Implementing the CDMS

This chapter presents the platform on which the CDMS is implemented. In order to create an effective implementation of the CDMS, the choice of platform is critical. The CDMS must manipulate and store data, metadata, meta-metadata and application code. Therefore a direct representation of each of these is crucial for programming accuracy. The choice of using persistent programming was made based on the availability of orthogonal persistence and the ability to treat code as data in persistent systems.

Section 6.1 generally introduces persistent programming systems. Section 6.2 describes PS-algol. Sections 6.3 and 6.4 describe Napier88 and the Napier88 Library, respectively. Section 6.5, as a summary, discusses the advantages of the persistent approach.

6.1 Persistent Programming Systems

The main implementation tools used to build the CDMS are programming languages based on the concept of persistence, together with the persistent store which they operate against and the libraries which populate the store.

This section introduces the principal ideas which motivate the design of persistent languages. Two issues dominate the discussion. One is the provision of consistent support for data with a wide variety of lifetimes - persistence. The other is the extension of data management to include the management of code as well as data.

The **persistence** of a value represents the duration for which it is accessible by a program. This could be as short as the time it takes to execute the block in which a value is declared, or it could be longer than the process or even the computer system in which the

relevant value is created. Conventionally, programming languages manage short-term or transient data, while file managers and database systems deal with long-term or persistent data. Programming languages are designed to enable computation to be described efficiently, while file systems are designed to enable data to be accessed efficiently. It is therefore no surprise that the data handling facilities of each vary considerably. In creating applications which must manipulate short-term data and access long-term data, a great deal of code must usually be written which bridges the gap between the two. By creating a system in which long term and short term data are treated in the same way, much of this coding can be eliminated.

The manner in which persistent languages achieve this is to start with a programming language model of data, but then to maintain this same model for stored data. Thus any piece of data which is created as a typed value in a program retains its structure when it is stored. The two important ideas here are that any value can be stored and that the structure of the value is not lost when it is stored. Any programming language which satisfies these principles, which together are called the principle of **orthogonal persistence**, is referred to as a **persistent programming language** (PPL).

The other critical property of persistent languages is that code fragments (structured into procedures) are first-class values. That is, a procedure can be manipulated in the same way as any other value. Given the orthogonally persistent nature of the language, this means that procedures can be stored as procedures rather than by use of some indirect representation. A persistent language therefore manages a database in which code and data can be freely mixed. Since a great deal of the CDMS functionality concerns the management of code, this is an extremely valuable aspect of the support that a persistent language brings to the implementation.

The next two sections introduce the principal persistent programming languages PS-algol and Napier88. PS-algol was used in the early stages of the implementation, but the work was later transferred to Napier88 in order to make use of the more powerful features of that language.

A formal definition of PS-algol may be found in [PS-algol, 1987] and tutorials in the use of the language in [Carrick *et al*, 1987], while an informal definition of Napier88 may be found in [Morrison *et al*, 1989]

6.2 PS-algol

6.2.1 Overview

PS-algol was the first of a series of persistent programming languages introduced by the Persistent Programming Research Group (PPRG). It was developed in Universities of Edinburgh, Glasgow and St Andrews from 1979 to 1987.

PS-algol is a block-structured language of the Algol family. It is simple, powerful and consistent in its treatment of data. The power of the language has been developed from the existing language features by extending their scope, so that there are no exceptions to syntactic rules and no arbitrary restrictions on the usage of the language components. The language possesses the following principal features:

- **orthogonal persistence**, which means that data of any type may have any lifetime;
- **type completeness**, which means that all data types enjoy equal status within the language, so that there are no restrictions on the construction of types or on the management of values of these types;
- **first class procedures**, which means that procedures can be manipulated in the same way as other values - this is implicit in the data type completeness of the language;
- **reflection** - a callable compiler is available in the language, which means that new code fragments can be added to a program at run-time;
- **graphical types** - both image handling and line drawing facilities are provided;
- **strong typing** - each value has a fixed unchanging type;
- a mixture of **dynamic** and **static binding** and **type checking**;
- an **integrated stable store** providing persistence by reachability;

- **associative structures** via a standard library.

6.2.2 The PS-algol Type System

Every value in PS-algol has a type which is fixed and statically determinable at compile time. Values may be divided into three kinds:

- **scalar values** which are atomic data values;
- **composite values** which are values composed of component values;
- **complex values** which have identity and updatable state and are similar to records in Pascal and objects in Object-Oriented languages.

The scalar types of PS-algol include **bool**, **int**, **real** and **string**. There are also scalar types for files, pixels and pictures. The type **file** enables PS-algol to access data outside of the persistent store. The type **pixel** represents a single pixel in a bitmap, while the type **pic** is the type of pictures constructed as line drawings in the Cartesian plane.

Composite types may be created by use of the type constructors:

- the type $c\tau$ denotes the type of constant values of any variable type τ;
- the type **#pixel** is the type of a rectangle of pixels, or an image;
- **proc**(τ_1, ..., $\tau_n \rightarrow \tau$) is the type of a procedure having n arguments of any types τ_1 ... τ_n and one result of any type τ or, omitting the '$\rightarrow\tau$', no result;
- the type $*\tau$ is the type of a vector (or array) of elements, all of the same type τ.

The third kind of value in PS-algol is the complex object, called a **structure** in PS-algol, all of which have the same type - **pntr**. A structure is similar to a record in Pascal and belongs to a class similar to a Pascal record type. Structures are given a very important role in PS-algol. They are used for data modelling in a similar way to tables in a relational database. They are used to organise the persistent store. Furthermore, they are

used to provide the limited degree of polymorphism permitted in PS-algol. The problems that arise because of this overburdening of a single construct will be discussed later.

6.2.3 Using PS-algol

Values can be introduced anywhere in a program by **let** declaration clauses. When a new object is introduced, its name is given, its constancy is determined, its initial value is designated by an expression, while its type is inferred from the expression. Some examples of declarations in PS-algol are given in Figure 6.1.

```
let I=1                                  ! introduces an integer constant
let S:="ABC"                             ! introduces a string variable
let I=image 10 by 10 of off              ! introduces an image constant
let Bvec=@1 of bool [true, true, false]  ! introduces a vector of booleans
let Add=proc(I, J: int→ int)             ! introduction of a procedure constant
...                                      ! body follows on subsequent lines as a
                                         ! single block of code
structure Person(name: string; age: int) ! introduction of a structure class
let J=Person("Jean",18)                  ! introduction of a structure value
```

Figure 6.1 Declarations in PS-algol

This figure shows how **let** is used to allow values of all of the different kinds to be introduced in the same way. It should also be noticed how the presence of a colon before the equal sign indicates that it is a variable that is being introduced - the absence of the colon means that it is a constant. **let** clauses can occur anywhere in a program - wherever a new value is found.

Programming the behaviour of the application uses a similar syntax to Pascal - with sequences of statements combined into blocks by use of **begin** ... **end** delimiters. There are also similar **if**, **while**, **for** and **case** statements to provide repeated and conditional execution. Ultimately code is made out of atomic statements - assignment to change the values of variables and read and write statements to allow user interaction via the standard input and output streams and to allow data to be stored and retrieved in files. User interaction can also occur via the graphical functions. Input comes from a standard function to read the mouse, while output goes via standard images which hold the current values of a screen window and the cursor.

A table in PS-algol is a structure which contains a set of one-to-one mappings from strings or integers to objects of type **pntr**. The table structure is of particular importance as it is used by the persistence mechanism of the language. In PS-algol, the mechanism for making data persist consists of inserting the data into a structure which is reachable, by following **pntr** chains, from some persistent root called a 'database', which will itself be an object of type **pntr**. Since any data object may be put into a structure, any data object may be made to persist and so the provision of persistence is orthogonal to data type.

To summarise, creating an application in PS-algol consists of writing programs which contain: the retrieval of values from the persistent store; some computation expressed procedurally; and the storage of values in the persistent store [Cooper, 1989]. Retrieval consists of opening the relevant database and following pointer chains from tables, structures or vectors to the required objects. Storage is only explicitly required if new values in the store are to be created. Otherwise if any variable value already in the store is updated the new value will be stored automatically. Creating a new value in the store consists of creating a reference from an already stored value. Since procedures are first-class value in the language, building an application is carried out in a modular fashion by writing programs which store the modules in the persistent store for later re-use.

6.2.4 Discussion

This subsection provides a review of the unusual features of PS-algol, their value and concludes with some problems of using the language.

The advanced features of PS-algol which do not normally exist in other Algol-like languages include data type completeness, the graphics facilities, the mechanisms which provide persistence, first-class procedures, the availability of the compiler as a function at run time, and the extensible union type **pntr** for modelling complex objects and providing a degree of polymorphism.

Data type completeness is a critical feature of PS-algol and has the following consequences. There can be variables of any type. Any value can be made the component of an object, can be stored or can be either the argument or result of a procedure. This gives the application programmer a great deal of freedom in creating and storing data structures which combine numerical, textual, pictorial and procedural values.

The graphical types bring two primary benefits - an inbuilt model for pictorial values and the ability to program the user interface. For the purposes of the CDMS, it is this latter benefit which dominates. Since the CDMS is intended to manage the user interface, it is of particular value to be able to represent user interface components and, indeed, the whole interface as values in the language.

The persistence facilities eliminate the need for elaborate coding to ensure that the complex data structures which embody all of the information managed by CDMS are stored properly. It is a hard enough task to construct a model, which includes data, metadata, meta-metadata and the user interface, without having to construct a mapping from this to the file store as well.

The notion of first-class procedures requires that procedures can be manipulated in the same way as any other value. In PS-algol, procedures can be the values of variables, and the argument or result of other procedures.

The principal usage of first-class procedures includes:

- the representation of actions as procedural objects with no restriction on what can be done with them;

- the production of abstract data type representations of data, since a package of procedures can be returned from data creation procedures - a table is one example of this;

- the storage of procedures, which greatly enhances modular and incremental application development.

Owing to the existence of first-class procedures, it has become possible to create a function which calls the compiler. A string which represents a procedure can therefore be passed to the compiling process, which returns the compiled procedure, and this may then be used in the same way as any of the statically written procedures of the program. Having the compiler available at run-time is another vital aspect of the support which PS-algol brings to the implementation of the CDMS. The CDMS works by using the DBE's choices to guide the generation of user interfaces to the global model. Each of these interfaces is built as a source string, compiled and added to the set of available interfaces. Although it would have been possible to create a generic model of the CDMS

environment by use of some form of indirection, this would have been inefficient in practice. The availability of the compiler makes it possible to create the same program that a human programmer might have programmed given the DBE's choices - but by automatic means.

The **pntr** type, however, causes problems since the type space in PS-algol is effectively partitioned. There are scalars, vectors, images and procedures on the one side, and the PS-algol structure classes on the other side. The types of the former are checked at compile time, while the check of classes is deferred until instances are used. Deferring the type checking of programs is essential for the incremental development of complex applications and for allowing applications to be re-bound to new databases [Atkinson *et al*, 1988]. Thus there are two conflicting uses of **pntr** - as a data modelling construct for complex object structures and as a language feature to provide dynamic typing. This conflicting use of **pntr** causes two problems:

1) The data modelling is not very precise. For instance, if the *Person* structure of Figure 6.1 is to be enhanced with an indication of the organisation that the person worked for and there existed:

   structure Organisation (name: **string**; address: **string**)

   then *Person* could be extended to

   structure Person (name: **string**; age: **int**; worksFor: **pntr**)

   i.e. the *worksFor* field refers to a structure, but one of completely unspecified kind. It would be possible to have the *worksFor* field point to a value of any class at all, although it should, of course, point to an *organisation* value.

2) The polymorphism is partial. It is possible to write a procedure which is polymorphic over all structure types since all have type **pntr**. However it is not possible to make a procedure be polymorphic over any type - **int**, ***bool**, **proc** (**int**→**string**), etc without creating indirection. This is a serious deficiency which impedes progress on complex programming tasks.

Napier88 fixes these problems by providing two better facilities in place of the single overburdened **pntr** type.

PS-algol has proved suitable for various sorts of programming task. Firstly, it suits writing data-intensive applications. Orthogonal persistence eliminates the burden of low level data storage from the application programmer's shoulder, while graphical types enable the creation of user-interface straightforwardly. PS-algol also suits writing database systems which support semantic data models or object-oriented database systems [Cooper, 1987; Kulkarni and Atkinson, 1987; Cooper and Qin, 1990]. Construction of such systems uses first-class procedures, as well as persistence and the graphical types.

6.3 Napier88

6.3.1 Overview

Napier88 has passed through a series of versions which are of improved power and efficiency - version 2.0 appearing in 1994 and version 2.2, the latest, appearing in the Summer of 1995 [Napier88, 1995]. The work to be described here, however, was carried out using version 1 of Napier88.

Napier88 inherits much of its style and facilities from PS-algol. The syntax is very similar. There are first class procedures and the same graphical types. The tightly integrated stable store is much the same, albeit using a different bulk type as its main organising feature. The language is data type complete. The principal additions include:

- a much richer type system with a null type, polymorphic types, recursive types, abstract data types, variant types and environments;
- explicitly declared types;
- multi-entry processes, distribution and concurrency control - not used in CDMS implementation and not further discussed here.

The Napier88 system consists of the language and its persistent store. The system is supported by a layered architecture which contains, among other things, the **persistent abstract machine** layer and the **stable persistent storage** layer. The persistent abstract machine provides the ability to execute Napier88 programs, and also monitors interaction with UNIX on which Napier88 resides. The persistent store is

populated when the system is installed and is used by the persistent abstract machine itself for error and event procedures.

6.3.2 The Napier88 Type System

Much of the type system is inherited from PS-algol: the same base types; the same graphical types; the same vector types. Procedure types are enhanced with type parameters. Structures become an integrated part of the type system.

One crucial change which has occurred, though, is that types can be explicitly named by the programmer, and newly named types can appear anywhere that the system types can. Figure 6.2 shows some type declarations in Napier88. Such declarations can occur anywhere in a Napier88 program, but are best gathered together in a separate file to create what is called a declaration set.

The first two declarations in Figure 6.2 are aliases for system types, which can be used to help document the program. The third declaration shows how procedure types can be named. The fourth declaration shows how structure types are declared. The second field is of some user-defined type *Address* and this is explicit. The type *Organisation* is then used in the definition of *Person1* and *Person2*, but unlike PS-algol, the *worksFor* field can only be an *Organisation* record.

```
type StaffNum is int
type Date is string
type AFunction is proc(int, int→ int)
type Organisation is structure(name: string; address: Address)
rec type Person1 is structure(name: string;age: int;
                              worksFor: Organisation; spouse: Person1)
type Number is variant(I: int; R: real)
type Optional[T] is variant(present: T; absent: null)
rec type Person2 is structure(name: string; age: int; worksFor: Organisation;
                              spouse: Optional[Person2]; extraInfo: any)
```

Figure 6.2 Declarations in Napier88

The definition of the two forms of Person illustrates another improvement in Napier88; that is, the ability to define **recursive types**. Names in Napier88, like most

Algol-like languages, must be declared before they are used. If a type declaration makes use of itself, this is flagged by the programmer to the compiler by the key word **rec**. Mutually recursive types and recursive procedures can similarly be defined.

One extension to the type system is the availability of **variant** or union types. The definition of the type *Number* in Figure 6.2 specifies that a *Number* is either an integer or a real - the options are tagged.

Another important facility added in Napier88 is that of **parametric** types; that is, types with a component whose type is a parameter. *Optional* is a variant type which models situations in which a value may or may not be present. The second alternative uses the system base type **null** which is the type of values with no data associated with them. Null values model missing information, but are also vital to ground recursive values. The definition of *Person1* discussed above, although legal, is quite literally valueless. It is impossible to create an instance of type *Person1*, since the first instance created requires the existence of a previous created value for the spouse field* ; a definition like that given as *Person2* is therefore needed so that the first instance created can have a null spouse.

There are two other system types which are heavily used in the system - **any** and **env**. The type **any** can be used wherever it is useful to leave the type of a variable, parameter or structure field undefined until run-time. Again, *Person2* has a field for schema expansion. The *extraInfo* field can hold values of any type and so different applications could extend the *Person2* type in different ways. Use of an **any** value requires an operation to project out its underlying type.

The type **env** has instances which are environments. An environment is a container for a set of name-value bindings. Operations an environment allows include the addition and removal of bindings and bringing bindings into scope of the current program. Type **env** is particularly valuable since it is used to organise the persistent store - playing the same role as the table structure does in PS-algol.

Finally, Napier88 provides an Abstract Data Type mechanism. An **abstype** may be used where the data object displays some abstract behaviour independent of representation type.

* Napier88 does not allow recursive value declaration, one of the main weaknesses of the language.

6.3.3 Use of Napier88

Napier88 is used in a similar way to PS-algol. The syntax is similar with the addition of a **project** clause to get values out of variants and **any**. Once more, the principal mode of operation is to write programs which access stored values, perform some computation and then place values into the store. Applications are built out of small programs which put modules into the store for later re-use. Procedures are first class and reflection is available in a similar way.

Some differences from PS-algol however occur:

- the richer type system permits more detailed and precise data modelling;
- organisation via the **env** type permits a greater range of binding styles to be used;
- the existence of the type **any** means that deferring the type of an object and the use of structure types is clearly separated;
- the availability of parameterised types and procedures means that truly polymorphic programs can be written;
- there is a larger library of system functions - see Section 6.4.

6.3.4 Discussion

The work was switched to Napier88 primarily because of the first point above; that is, a more precise description could be given of the complex data which CDMS must manage. In addition, the polymorphism and improved binding mechanism were attractive.

The convenience of using Napier88 Library including WIN facilities is also an important consideration of making the switch.

The CDMS in this research could be considered, in one respect, an experiment in the use of Napier88.

6.4 Napier88 Library

The previous section described the language features of Napier88, but one of the benefits of a persistent language is the ability to extend the power of the language with a coherently organised library of software components. The library, which is held in the store, contains all of the usual functions for arithmetic, string manipulation and so on, but also contains system level functions. Moreover, the library is extensible by any user and so the Napier88 system evolves largely through the development of new library components. Adding to a library is achieved by running a program which inserts a new component in the persistent store; that is, no other reloading or relinking is required.

Among the most important sections of the library for this project are those dealing with: the compiler and its components; the management of bulk values; and the window management system, WIN.

6.4.1 The Napier88 Compiler

The Napier88 compiler is provided as a function which can be used at run-time in a similar way to PS-algol (refer to Subsection 6.2.1). As the language has first class procedures, this appears as a set of composable procedures for syntax analysis and so on. Although these were not used separately, the underlying philosophy provided support for the implementation of CDMS.

6.4.2 The Bulk Values

Bulk values are those which are composed of potentially large numbers of similarly typed component values - lists, sets, vectors and so on. A large library has been developed for managing such values [Atkinson *et al*, 1993b], but the work described here has made particular use of one such bulk type - the map [Atkinson *et al*, 1993a]. A map is made up of a set of key, value pairs and is used to support keyed search. To take an example from the current work, the set of X is implemented as a map with string keys and X values; that is, it has type Map[string, X].

The maps library has a great many procedures, but the most important are ones to: create an empty map; insert a new key, value pair; delete a pair; create a new map by filtering the contents of an old one; and applying a procedure to every pair in the map.

Maps are used in CDMS to model most of the large structures. There are maps for many of the collections in the CDMS database.

6.4.3 WIN

Developed on top of the graphical facilities of the Napier88, WIN [Cutts *et al*, 1989] is a window management system which provides a mechanism for the distribution of input events to a group of concurrently active Napier88 programs. Each program is a procedure that takes an input event as a parameter and performs some action to process the event. WIN offers the programmer fine control over the distribution of events and indeed the event distribution system may be reconfigured while it is running.

In the WIN system, an interactive program is built from a number of sub-programs, each of which responds with an action as a particular input event occurs. Thus, a simple window-based text editor could be split into programs which:

- insert text;
- adjust the currently selected text;
- select the current view of the text;
- execute particular functions associated with light-buttons, such as cut, copy, paste and clear.

Each of these operations is activated by a particular event or sequence of events. The relevant types of events for these programs could be keyboard events, mouse events over the text area, a scroll-bar area or a light-button area.

These programs are called **applications**, which are implemented as procedures taking an input event as a parameter. A typical application is designed to process a limited class of input events. The programmer provides an associated procedure which tests events to determine whether the application should accept them. The procedure takes an event as a parameter and returns a boolean value, **true** if the application should accept the event and **false** if not. The application and its associated procedure are known as a **notification**, while the routing of input events to the appropriate applications is performed by a **notifier**. An application is connected to a notifier's event distribution system by

passing a notification to the procedure *addNotification*. This causes the notifier to insert the notification into an internal list. There is a parameter which specifies the insertion position in terms of an offset from the front or back of the list. The result returned by the procedure is another procedure which will, when called, remove the notification from the list.

Events are routed through the notifier's distribution system by passing them to the procedure *distributeEvent*. When this procedure is called, the notifier scans its internal list, passing the new event to the *examineEvent* procedure of each notification in turn, until one of those procedures returns **true** or the end of the list is reached. If one of them does return **true** the notifier calls the associated application, passing it the new event, and the scan terminates. The event is discarded if none of the *examineEvent* procedures return **true**.

The position of the notifications in the list is significant. If there are several notifications that would potentially accept an event the application of the notification highest in the list will be the only one to receive it. If the notification that accepts a particular event is different from the notification that accepted the previous event, the notifier sends a deselect event to the application of that previous notification and a select event to the current application before sending the new event to it.

The way in which a notifier is used will depend on the application. There are single-level notifiers and multi-level notifiers. A simple text editor program could be programmed using a single notifier. It is also possible to construct hierarchical event routing structures as a notifier's *distributeEvent* procedure and a notification's *processEvent* procedure are both of type Application. One notifier's *distributeEvent* procedure can be registered with another notifier. In fact, the application which deals with mouse events over the scroll bar could itself be a notifier which routes them to smaller sub-applications.

A notifier may be reconfigured dynamically. In fact, any programs with access to the *addNotification* procedure of a notifier may add notifications to that notifier at any time, and similarly any programs with access to the procedures returned by *addNotification* may remove those notifications. In short, the event routing structure is changeable while the system is active.

Moreover, the use of notifiers gives an easy way to make applications persistent, that is, if a notifier is made to persist by arranging for it to be reachable from the root of persistence, all the notifications and associated applications within its closure are

also made to persist. The application implemented by the notifier can then be restarted by a program which binds to the notifier and sends fresh events to it.

A notifier receives input events and distributes them among the applications registered with it but it is not responsible for generating those events in the first place. This is done by an **event monitor** which continually polls the keyboard and mouse, generates appropriate events and passes them to a notifier for distribution. Normally there will be only one event monitor active at a given time.

Alongside this control mechanism, WIN provides a tool kit of user interaction objects including menus, scroll-bars, editors, radio buttons and so on. Using these a good looking interface can be built to a programming functionality - the CDMS is built on top of WIN for this reason.

6.5 The Advantages of the Persistent Approach

The features described in the previous subsections offer great advantages for writing applications of large scale. These include the following:

1) Low level data management is dealt with by the system instead of the programmer. In fact, the components of program concerning data input and output in a persistent system are redundant. Using PS-algol or Napier88, the organisation that is imposed on the data in order to handle these within the program is also the structure in which it is stored. For instance, to store a list of integers, all that is required is to enter the head of the list into the database. No recourse to external data handling programs, such as file management systems, is needed any longer. Actually this is merely an instance of a more general advantage of using a persistent language; that is, all programming jobs can be done in the same language.

2) Image and picture objects facilitate the modelling of graphical data. The existence of the image and picture types permits graphical data to be stored in exactly the same way as textual or numerical data. It relieves the programmer from having to invent a coding strategy to handle the graphical data.

3) The graphics facilities also offer tools to produce user interfaces. As a great deal of attention must be paid to the user interface, having efficient tools to

produce a good interface within the language is therefore of great benefit. This avoids the problems of forcing unnecessary constraints on programs and of restricting the kinds of interface that can be generated. These problems are often caused by using external packages.

4) Strict type checking gives early detection of data misuse. The detection actually occurs at program compilation time, data loading time or data reference time; that is, it always occurs before the relevant data is used.

5) First-class procedures facilitate the modelling of actions. As procedures can also be manipulated in the same way as textual, numerical and graphical data, it becomes possible to model and store activities. Assertions, conditions or triggers may also be modelled by procedures which take in values for the variables and return a boolean result. In addition, objects which are pairs of condition and action procedures can be composed. It is therefore easy to write a program fragment which loops, testing conditions and applying the paired action if the condition is satisfied.

6) First-class procedures support incremental compilation. The languages provide an ideal framework in which to program the modules. Each module as a procedure is stored in the persistent store, and can then be retrieved from the store and accessed by other modules.

7) First-class procedures also support ADT. The essence of the Abstract Data Type is the restriction of access to data to a limited number of operations which are pre-defined on it. Since procedures can be put into structures and structures can be returned as the results of procedures, ADT generating procedures can be constructed which return a structure containing a package of procedures.

8) Structure types model complex data. Since PS-algol and Napier88 are data-type complete, the fields of a data structure can be of any type. This means that complex data objects can be constructed which combine numbers, textual information, graphical data, activities and assertions. Furthermore, objects with a more complicated structure can be modelled by using pointer fields to sub-objects.

9) The **pntr** in PS-algol and **any** in Napier88 permit delayed binding of programs to objects. It is hence possible to write general purpose procedures which manipulate objects of various types by packaging the objects of different types into structures and passing around pointers to those structures. For instance, a polymorphic list processing package can easily be composed.

10) The run-time compiler supports polymorphism. The run-time compiler allows programs to be constructed which run against an unbounded set of data classes. Such programs are written so that they discover the class of data they are expected to deal with this time and, using string manipulation, merge the class information with the algorithm and compile the resulting procedure. Once compiled, the subsidiary program would be stored so that it need not be regenerated every time an object of that class is encountered.

The next chapter describes, based on the platform described in this chapter, the implementation of the CDMS, which is the main practical work in this research project.

7 The Design and Implementation

The CDMS is a set of programs which manage data models, interfaces, schemata and databases. The system is built around a central model - the global model - in which data structure, constraints and behaviour can be described coherently and consistently. The programs are designed to allow the user to:

1) create data models by instantiating the global model;

2) manipulate those models;

3) create data schemata by instantiating a model;

4) manipulate those schemata;

5) create databases as instances of a schema;

6) manipulate those databases;

7) determine the user interface to the last four of these facilities.

The creation of a CDMS thus requires the design and implementation of storage structures and manipulating operations of data, metadata, meta-metadata and user interfaces. This chapter describes such a design and implementation. Section 7.1 goes into more detail about the design philosophy. Section 7.2 describes the overall architecture of the system. Sections 7.3 to 7.5 describe the global model, the instantiation process and storage structure respectively, while Section 7.6 describes the user interface. Finally, the CDMS is not complete in every respect and so Section 7.7 describes the implementation status at the time of writing.

7.1 Implementation Overview

The principal task that the CDMS must support is user driven instantiation from one level of the architecture to the level below. The result of this instantiation is a tool for using the instantiated sub-system. The instantiation processing makes use of user choices to configure the relevant tool.

Thus the system is implemented so that:

- the functionality of the system is decomposed into fragments which can be recomposed easily;
- the fragments are parameterised so that user choices can guide the configuration;
- the user choices can be captured;
- for each level of the system, there is a framework, or template, into which the fragments can be fitted;
- the system as a whole is built on a structure which allows new tools to be added and used.

Furthermore, the CDMS has been implemented with a user interface designed to provide the following:

- all usage of the program is carried out in a unified way;
- at every level full details of the decisions being made are kept consistently visible.

These two design decisions are appropriate for the current prototype as they make explicit the underlying architecture and ensure that the behaviour of the system is continuously observed. It is recognised that these are not appropriate for a delivered system. It is not usual for the user to manipulate a schema in the same way that the

database is manipulated; it is not possible for the implementation detail of each value to be continually in view.

7.2 Program Architecture

The CDMS program is designed to capture user choices and use these to guide the assembly of program fragments into new components. Figure 7.1 shows the way in which this works.

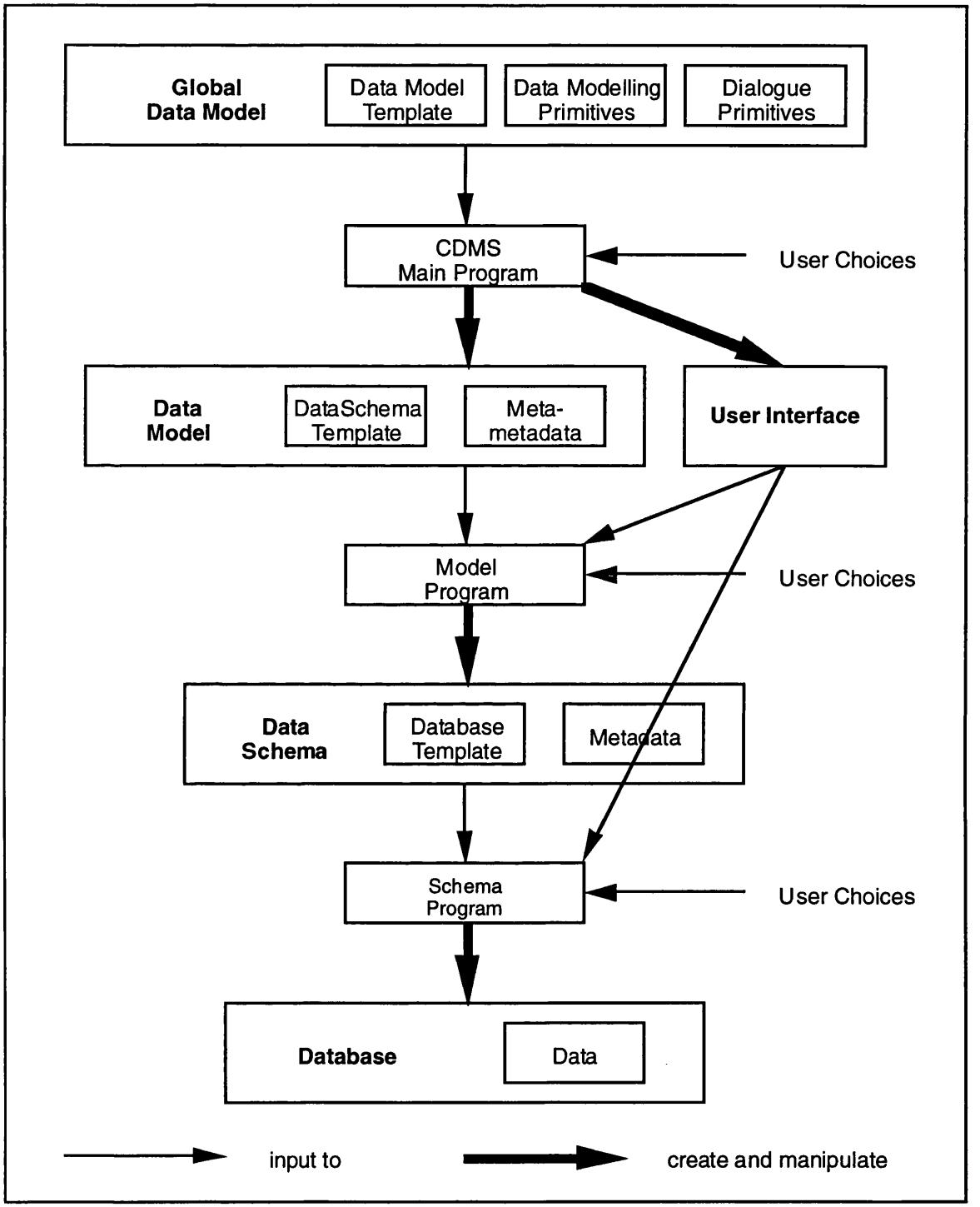

Figure 7.1 Program Architecture

The program makes use of four main inputs:

- a **data model template** - this is a framework to be combined with the data structures and operations which will be determined by the user;

- a set of **data modelling primitives** - these implement the operations which manipulate instances of the global data model structures - the operations themselves being instantiated for embedding in a data model;

- a set of **dialogue primitives** - these provide for interaction permitting the user to input and display values, choose items and so on;

- a set of user choices - these are the input by the CDMS user to generate a sub-system.

The effect of the user choices creates a complete modelling program by

1) selecting from the set of data modelling primitives;

2) instantiating the selected items as required;

3) selecting from the set of dialogue primitives and coupling them with interaction elements;

3) embedding them in the appropriate environment.

The template, the data modelling primitives and dialogue primitives, in a sense, implement the global data model, while the CDMS programs implement the instantiation process.

Currently the CDMS system is implemented by the following modules (refer to Figure 7.2):

- 'CDMST.N' declares the types to be used in the CDMS;

- 'InitCDMS.N' creates the global environment *CDMS*, its subsidiary environments *gModel*, *gDialogue*, *utilities*, *routines* and the map *Models*, and then stores the generic data modelling primitives and dialogue primitives into environments *gModel* and *gDialogue* respectively;

- 'CDMSU.N' stores common utilities to be used in the system into the environment *utilities*;

- 'CDMSM.N' stores model manipulation programs into the environment *routines*;
- 'CDMSI.N' stores interface manipulation programs into the environment *routines*;
- 'CDMSS.N' stores schema manipulation programs into the environment *routines*;
- 'CDMSD.N' stores database manipulation programs into the environment *routines*;
- 'RunCDMS.N' runs the system.

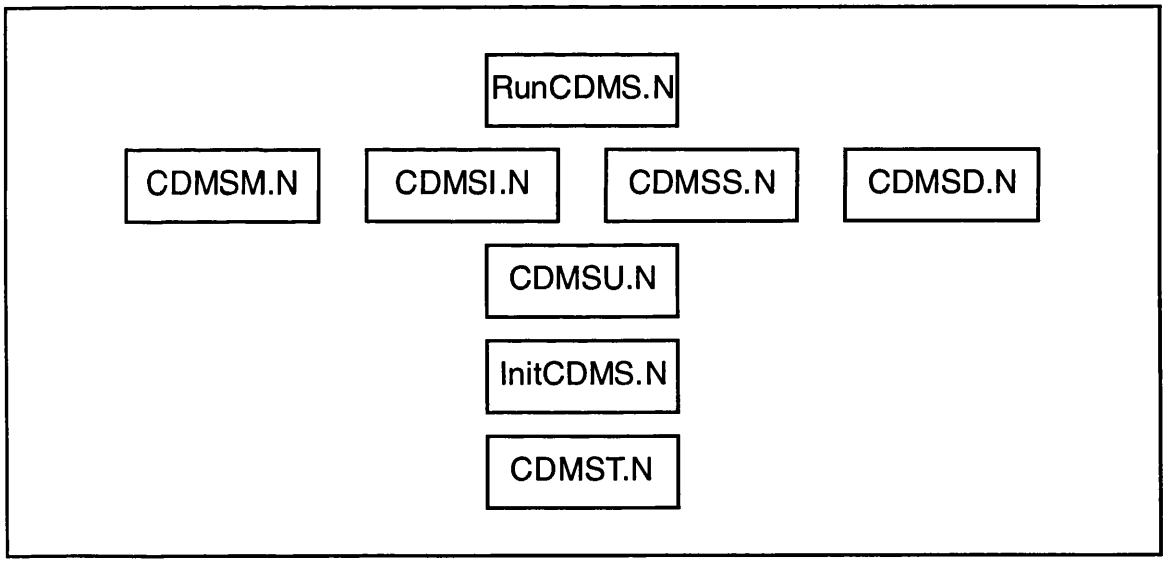

Figure 7.2 Software Modules

Actually, 'RunCDMS.N' allows the user to select system functionality.

'Create A Model', 'Edit A Model', 'Display a Model' and 'Remove A Model' are options prepared for DBEs.

When 'Create A Model' is selected, a string editor will be initially provided to receive the new model's name, and after checking that the name has not already been used for any existing model, a model editing window will appear on the screen; otherwise, the string editor will be provided again with some directive message. If an empty string is inputted, nothing will be done but the global level menu will appear again.

When 'Edit A Model' is selected, a menu including the names of all existing models will be provided to determine the model to be edited, then a model editing window will appear on the screen. If 'Cancel' is selected from the menu, nothing will be done but the global level menu will appear again.

When 'Display A Model' is selected, a menu including the names of all existing models will be provided to determine the model to be displayed, then a model display window will appear on the screen. If 'Cancel' is selected from the chooser, nothing will be done but the global level menu will appear again.

When 'Remove A Model' is selected, a menu including all names of the existing models will be provided to determine the model to be removed. If 'Cancel' is selected from the chooser, again, nothing will be done but the global level menu will return.

'Create An Interface', 'Edit An Interface', 'Display An Interface' and 'Remove An Interface' are options prepared also for DBEs. When any of these four options is selected, a menu including all names of the existing models will immediately be provided to determine the model which the relevant interface corresponds to, then they will require the interface's name in the same way as four model operations described above require the model's name. Finally, if one of the former three options is selected and an appropriate name is given, either an interface editing window or an interface display window will be provided accordingly to continue the process. If 'Remove An Interface' and appropriate names are selected, the system will simply do so and return to the global menu right away.

Interface manipulations run the programs against the environment uniquely for the corresponding model.

'Create A Schema', 'Edit A Schema', 'Display A Schema' and 'Remove A Schema' are prepared mainly for DBAs, while 'Create A Database', 'Edit A Database', 'Display A Schema' and 'Removing A Database' are prepared mainly for database end users.

Schema manipulations run the programs against the environments for the corresponding model and an appropriate interface. Data manipulations run the programs against the environments for the corresponding model and schema, as well as an appropriate interface. Any model level constraints should be assigned when a model is created or edited; any schema level constraints should be assigned when a schema is created or edited.

7.3 The Global Model

The global model consists of:

1) *meta-base* and *meta-complex*;

2) the meta-connection types *meta-relating* and *meta-inheriting*;

3) the meta-constraint types, including global meta-metadata constraints, global metadata constraints and global data constraints;

4) behavioural primitives for input/output and data manipulation; and

5) dialogue primitives.

7.4 The Instantiation Process

7.4.1 The Creation of a Data Model

The process of instantiating the global model to create a data model starts from the data model template.

The model builder then submits choices to the program via the interface described in Section 7.6. The choices take the following form:

- a selection of the global construct to be instantiated;
- a name for the instantiated construct;
- constraints on the use of the construct.

For example, the following choices are involved in the creation of the ER model:

- select *meta-base*, call it *simple attribute*;
- select *meta-complex*, call it *entity*;
- select *meta-complex* again, call it *relationship*;
- select *meta-relating*, call it *attributing*, fix one end to be *entity* or *relationship* and the other to be *simple attribute*;
- select *meta-relating* again, call it *relating*, fix one end to be *relationship* and the other to be *entity.*

By making these choices the following are created:

- a base type *attribute*;
- complex types *entity* and *relationship*;
- connection types *attributing* and *relating* with their combination capacities.

7.4.2 Adding the User Interface

The operations as embedded are written in terms of abstract interaction. The user must select some concrete interaction activities to complete the code. In fact, an option exists to utilise a default interface.

To change this interface, the user must decide the following:

- the symbol for each base type;
- the symbol for each complex type;
- the line style for each connection type; and
- the other dialogue primitives for each interaction element.

7.4.3 Creating a Schema and a Database

To create a schema, the database designer must start with a data model. The process of creating a schema is very similar to the process of creating a data model. A data model has a set of constructs which is structured in the same way as the set of constructs in the global model.

The instantiation process then consists of choosing with the following steps;

- selecting a data model construct;
- naming the instance.
- providing constraints where appropriate.

The result is a schema program with embedded operations for manipulating any database in the schema.

Finally, to create a database, the user must start with a data schema. The process of creating a database is, again, similar to the processes of creating a data model and creating a data schema.

7.5 Storage Structure

The storage structure of the system is shown in Figure 7.3. All of the CDMS and its data are held in a Napier88 environment called *CDMS*, which contains a map, as well as four sub-environments.

The four sub-environments are:

- *gModel*, which holds all of the generic modelling primitives;
- *gDialogue*, which holds all of the dialogue primitives;
- *utilities*, which contains fundamental procedures of general usefulness;
- *routines*, which contains CDMS manipulation procedures.

The map is:

- *Models*, which holds the set of data models, and their affiliated interfaces, schemata and databases.

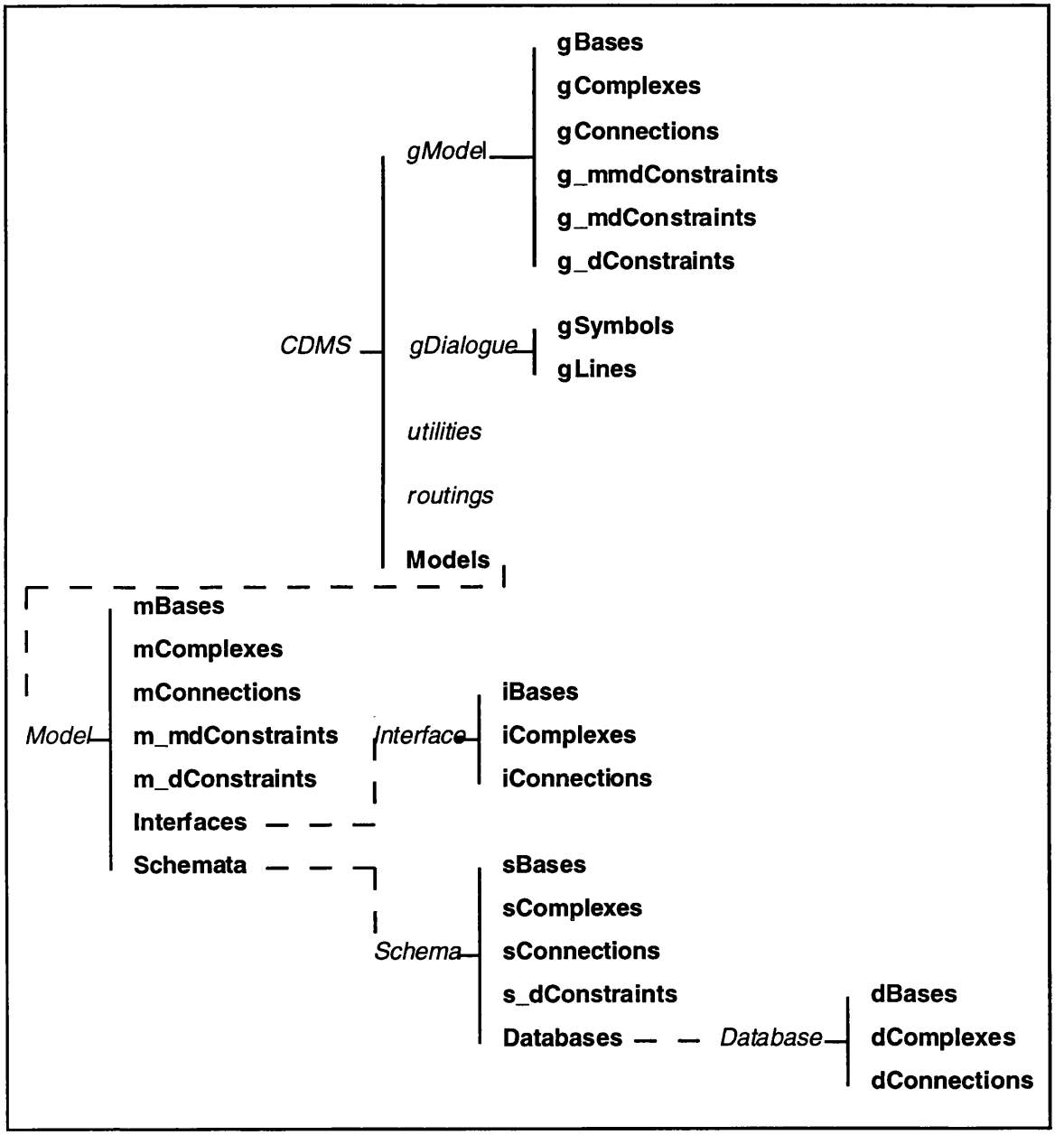

Figure 7.3 Storage Structure

When CDMS is delivered, it consists of the four environments fully populated, while the map is initially empty and is to be populated by usage.

gModel, in turn, contains the global data model represented by the following maps:

- *gBases* - the singleton meta-base type known to the CDMS;
- *gComplexes* - the singleton meta-complex type;
- *gConnections* - the set of meta-connection types;
- *g_mmdConstraints* - the set of global meta-metadata constraints;
- *g_mdConstraints* - the set of global metadata constraints;
- *g_dConstraints* - the set of global data constraints.

gDialogues contains the set of dialogue primitives implemented as procedures and performing such actions as inputting an integer, displaying a string, etc. These are held in the following maps:

- *gSymbols* - the set of symbols implemented;
- *gLines* - the set of line styles implemented.

Each model, when created, contains the model represented by the following maps:

- *mBases* - the set of base types;
- *mComplexes* - the set of complex types;
- *mConnections* - the set of connection types;
- *m_mdConstraints* - the set of model metadata constraints;
- *m_dConstraints* - the set of model data constraints.

Each model also contains the following maps, which, when the model is created, are empty and to be populated:

- *Interfaces* - this is a map of interfaces created for the model. A default interface is created with the model. Each interface is an environment containing the following maps:

    - *iBases* - a set of correspondences between symbols and base types in the model - there must be an entry here for every element in *mBases*;

    - *iComplexes* - a set of correspondences between symbols and complex types in the model for each entry in *mComplexes*;

    - *iConnections* - a set of correspondences between line styles and connection types for each entry in *mConnections*.

- *Schemata* - the schemata defined using the model. Each schema is an environment containing the following maps:

    - *sBases* - the set of base classes;

    - *sComplexes* - the set of complex classes;

    - *sConnections* - the set of connection classes;

    - *s_dConstraints* - the set of schema data constraints;

    - *Databases* - the databases framed by the schema. Each database as an environment then contains the following maps:

        - *dBases* - the set of base values;

        - *dComplexes* - the set of complex values;

        - *dConnections* - the set of connection values.

7.6 The User Interface to the CDMS Program

The capture of user choices has been achieved using a menu-driven interface.

Figure 7.4 shows the CDMS main menu, which allows a CDMS user to choose the CDMS functions.

As can be seen, the menu provides functions (create, edit, display and remove) to interact with each of the main components of CDMS. The top four functions support the maintenance of the set of data models as instances of the global model. The next four functions support the maintenance of the user interfaces of a particular model and so on. The help facility will give some useful information on the structure and functions of the CDMS in general, together with a brief description on all items in this menu in particular.

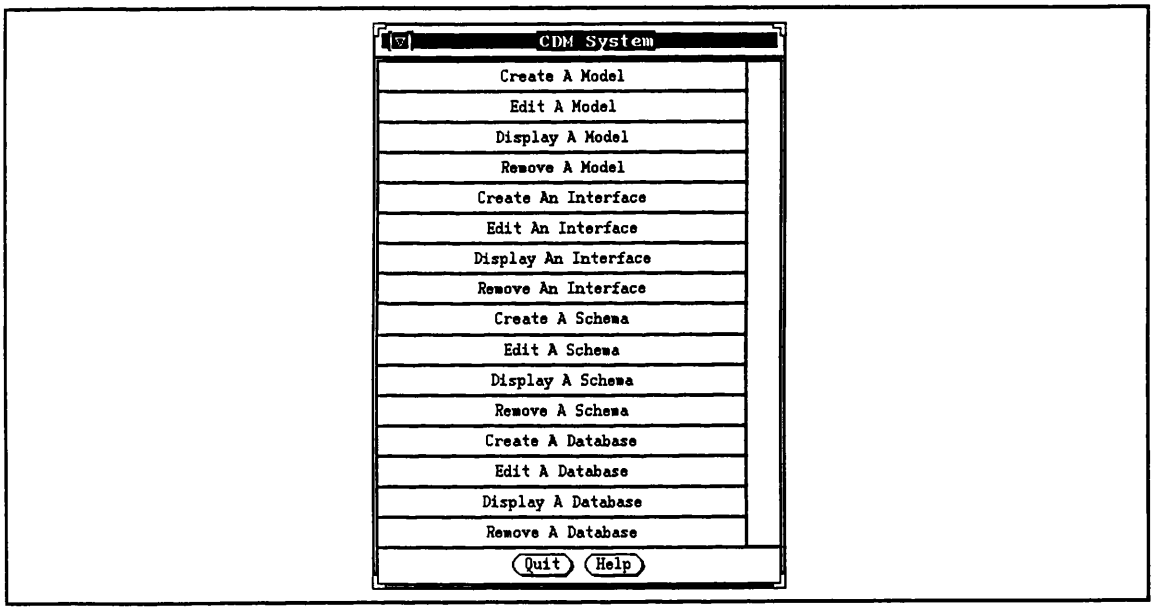

Figure 7.4 The CDMS Main Menu

The operations all follow a uniform mode of operation:

- **create** operations request a new name and then put the user into an editing window (described next);
- **edit** operations request an item by menu and then bring up the editing window;
- **display** and **remove** operations also bring up a menu of items and then either display the item or remove it.

All the editing windows (of which Figure 7.5 is an example) have the same format - one intended to support the process of instantiation which is the fundamental mechanism of CDMS. Recall that in CDMS, a database is an instance of a data schema,

which is an instance of a data model, which is, in turn, an instance of the global model. The overall process of instantiation consists of subsidiary processes of the instantiation of many individual elements. Thus each of the components in a data model is an instance of one of the global model components, etc.

An editing window supports this by providing a number of menus of the components in the higher level in the lower half of the window. The upper half contains a parallel set of the instances of these components, which have been created. For example, a schema editing window can be seen in Figure 7.5. A schema is built of instances of data model components. Thus the lower half of Figure 7.5 shows the menus of the components in the ER data model, while the upper half of the figure shows the components of a library schema which is being developed.

There are pairs of menus for each of the main kinds of components: constructs, connections, constraints and active values. To add a new element to the item being edited, one of the lower menu options, <O> say, is selected (with the left menu button) and a name, <N> say, is provided. This results in a new element, <O><N>, being added to the upper display. Moreover, a complete instantiation history of any component of any item is maintained, and this can be seen by selecting it with the middle mouse button. For an instance, in a global level menu, a single description (such as <meta-complex>) might appear. In a model level menu, items appear such as <meta-complex><relationship>. In the schema menu, this becomes <relationship><loan>, which can be expanded to show <meta-complex><relationship><loan> by pressing the middle mouse button. The database menu contains items like <loan><21>, which can be expanded to <meta-complex><relationship><loan><21>. Further light buttons are provided at the bottom of the screen for committing the changes or aborting them.

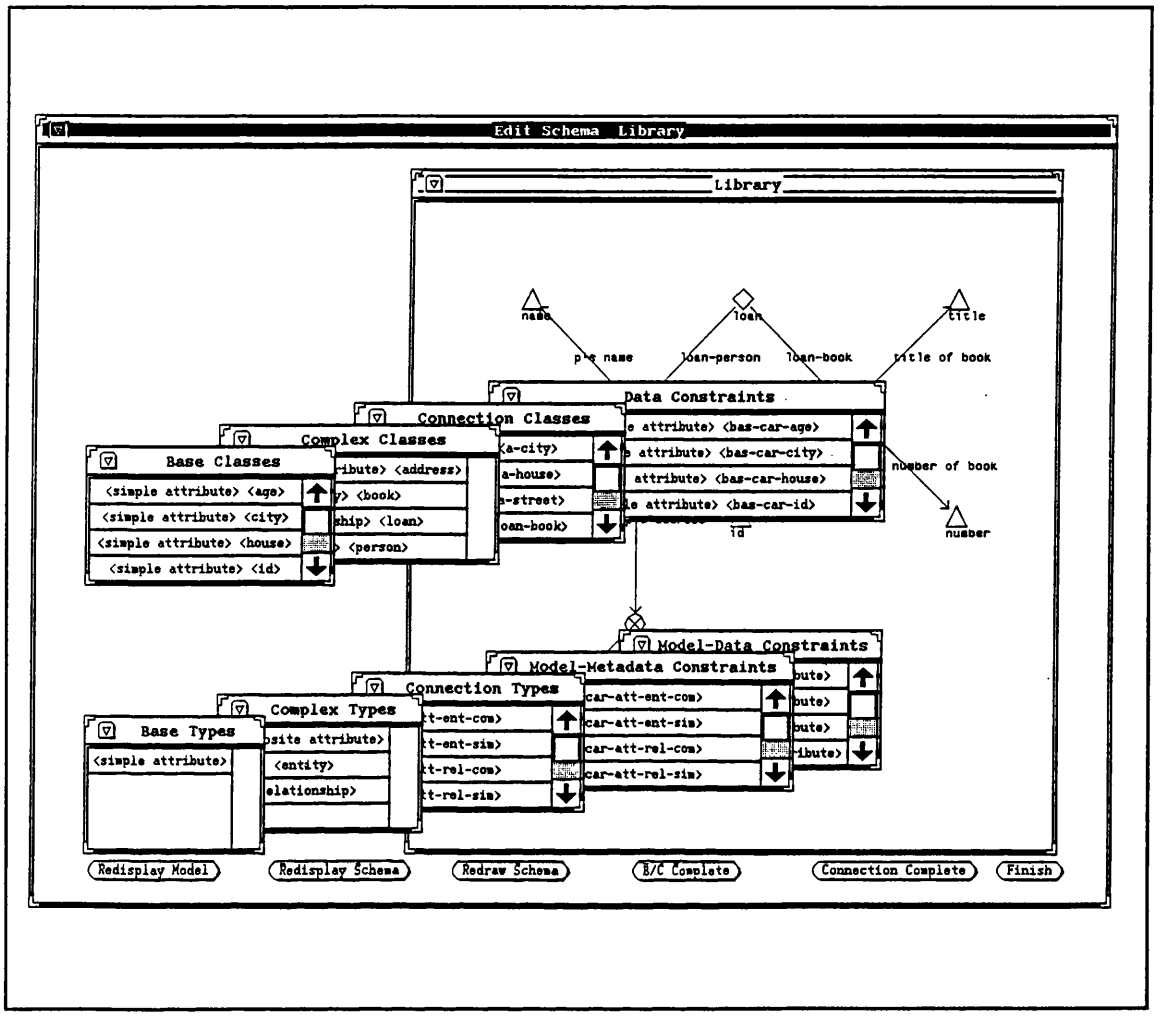

Figure 7.5 A Schema Editing Window

Also shown, in Figure 7.5 is a graphical representation of the schema. This is merely a passive display of the schema and is not manipulable. The actual graphical icons and line styles can be varied using a separate interface definition window. At the moment this is only available at the schema level and database level. Extensions to active displays of all four levels is possible.

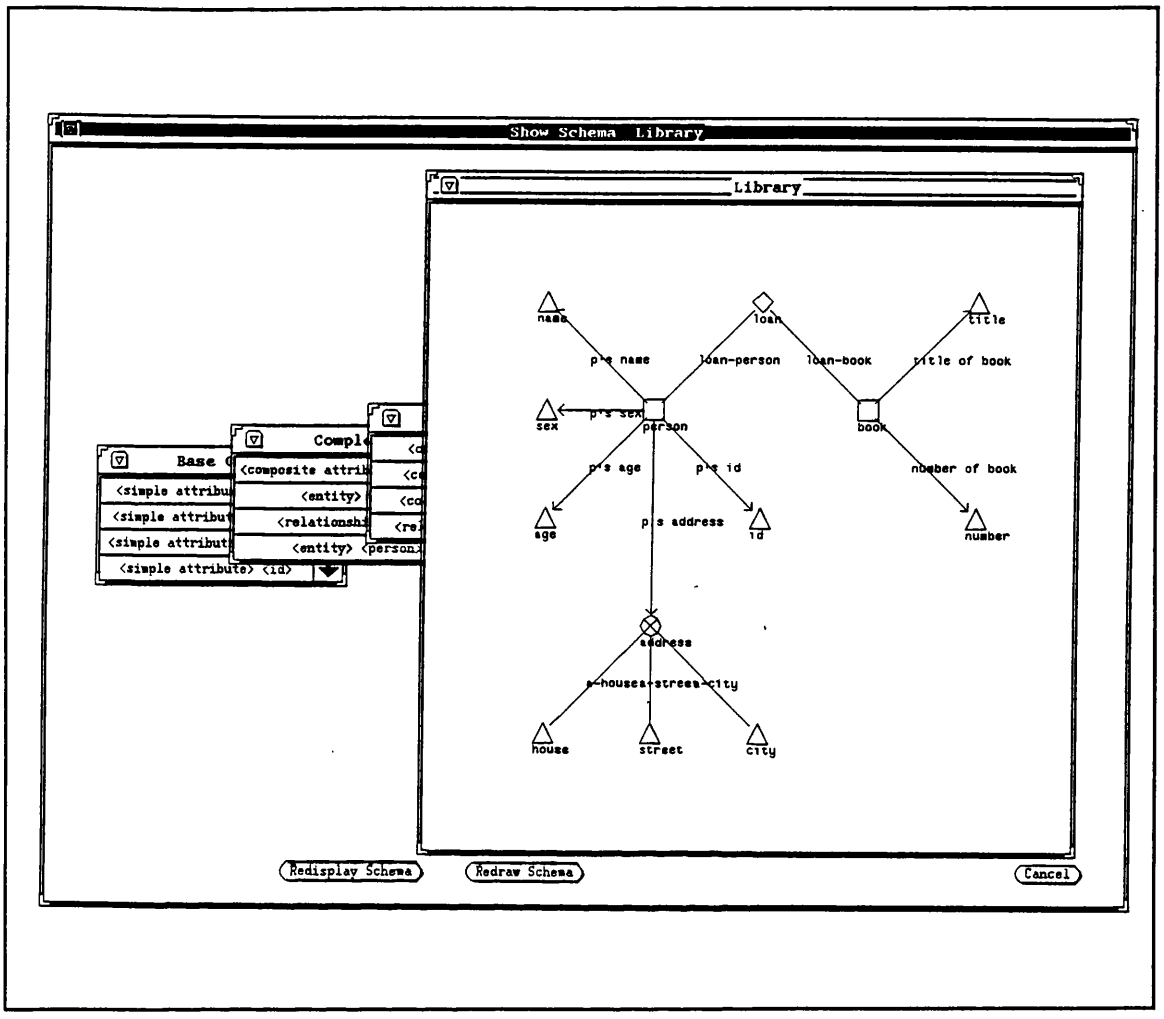

Figure 7.6 A Schema Display Window

A schema display window (Figure 7.6) is similar to a schema editing window, with the only difference being that it doesn't include the menus of its supporting model and it doesn't permit any schema editing operations. Once more, the model display window, interface display window and database display window are all similarly structured to the schema display window.

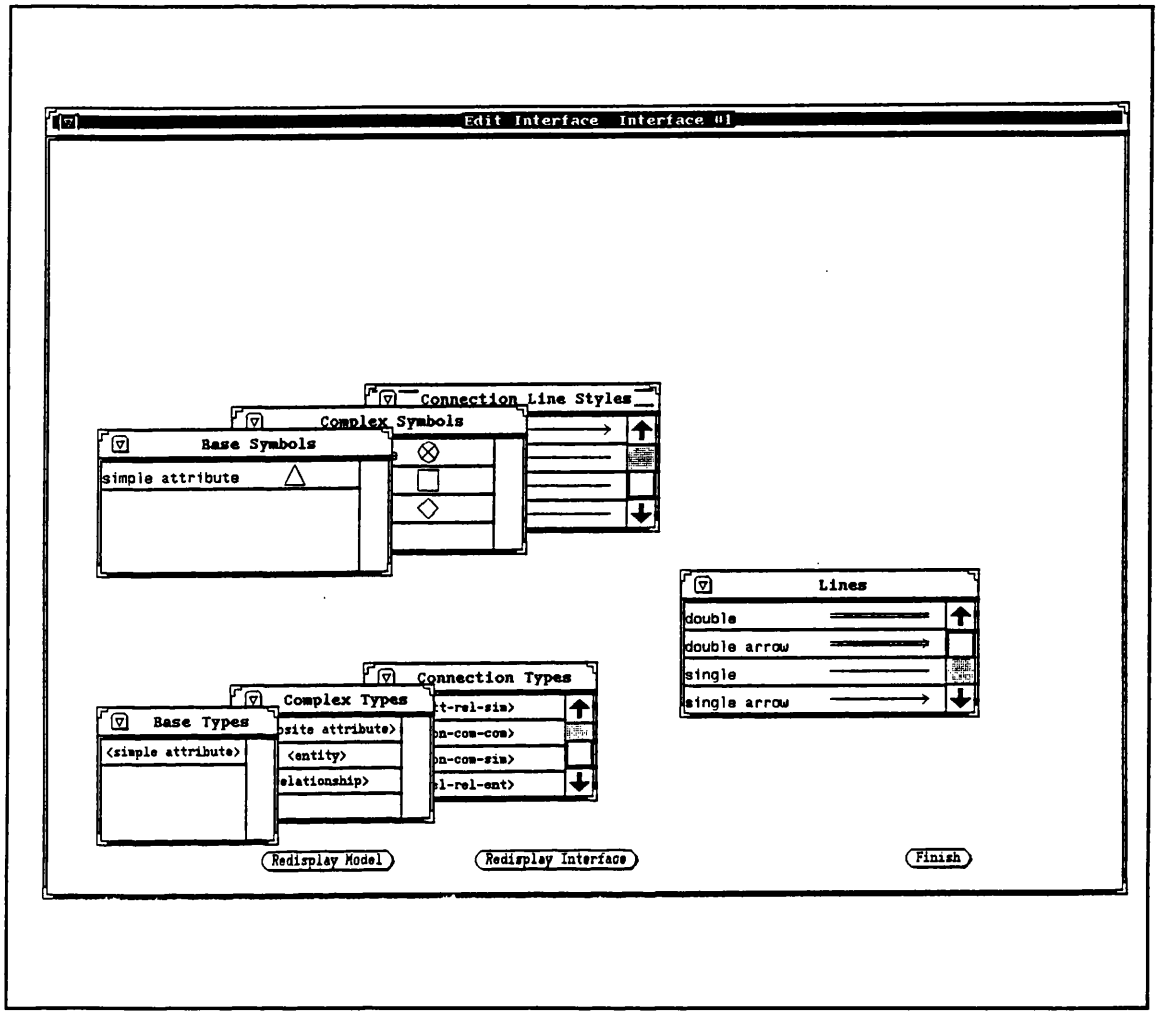

Figure 7.7 An Interface Editing Window

Similarly again, an Interface Editing Window is shown in Figure 7.7.

7.7 Current Status

The above description is of a fuller CDMS system. In the present implementation, the following limitations have been made due to pressure of time:

1) the behavioural aspect is not configurable at all, instead a limited set of operations are used at every level;

2) a full set of constraints is not available - ideally many more of the constraint kinds described in Chapter 4 would have been implemented;

3) the facilities for configuring the user interface are not extensive - only a graphical interface for database design can be configured.

The last point was an outcome of de-emphasising the user interface part of the work and concentrating on the data modelling aspects. Chapters 4 and 5 describe the complete strategy concerning constraints and behaviour, while this chapter shows the implementation approach which should be used to implement that approach.

However, the system as implemented is capable of managing a wide range of modelling constructs. Data models, schemata and databases can all be created and stored. The next chapter gives some examples of the use of the implementation.

7.8 Discussion

In the design and implementation of the CDMS, the features of Napier88, including orthogonal persistence, graphical data types, the ability to model complex objects, the availability of first-class procedures, especially the environments and reflection were very helpful.

Environments are extensively used in the current implementation. The global data model, set of dialogue primitives, models, interfaces, schemata, and databases are all represented by environments in the CDMS. All programs, which manipulate models, interfaces, schemata and databases, as well as utility routines of the CDMS are also stored in environments. On the other hand, reflection was extensively used in the early stage of the implementation. Napier88 is really appealing for data-intensive applications.

There were some difficulties, however. Such problems, which have largely been fixed, were experienced during the implementation as slowness, lack of software engineering tools, and lack of good library. More generally, the lack of inheritance is a drawback and the following syntactic features: using round brackets for fields, arrays and procedure calls; lack of recursive values - so one has to use variants to describe recursive types; lack of a bottom type - one can not use null to stand for a value of any type; and the packaging required to use any are all ones which once incurred problems.

8 The Application of the CDMS

This chapter demonstrates typical applications of the CDMS. To prove the usability of the system developed in this research, the constructions of a number of classical and semantic data models will be presented as examples.

In this chapter, Section 8.1 gives a general introduction on usage of the CDMS in configuring data models and user interfaces. Section 8.2 gives a formal syntax for the descriptions as used in Sections 8.3 through 8.6, which describe the constructions of the relational model, the entity-relationship model, the functional data model, and IFO, respectively. Section 8.7 concludes the chapter.

8.1 Introduction

Based on the implemented facilities presented in Chapter 7, the usage of the CDMS can be divided into four stages, which are as follows:

- the configuration of a data model;
- the configuration of an interface for a particular data model;
- the creation of a data schema supported by a particular data model;
- the construction of a database framed by a particular data schema.

Following some necessary preparation in Section 8.2, Sections 8.3 through 8.6 concentrate on the configuration of data models, giving the illustrations of instantiation paths of the components of the relevant data models. These models can then be used to

describe, via appropriate interfaces, various schemata and databases, thus supporting applications occurring in various fields.

8.2 Meta-metadata Syntax

This section gives a formal syntax for the descriptions as used in meta-metadata columns of the figures in the following four sections.

A model includes three kinds of meta-metadata, which represent constructs, metadata constraints, and data constraints of the model respectively. These meta-metadata are specialised from three kinds of modelling primitive, that is, constructs, metadata constraints, and data constraints of the global model.

Figure 8.1 defines the syntax for the descriptions representing model constructs; Figure 8.2 defines the syntax for the descriptions representing model-metadata constraints; while both Figures 8.3 and 8.4 define the syntax for the descriptions representing model-data constraints.

In all the figures, <base> represents a base type, <complex> represents a complex type, and <A> represents an atomic type; that is, a base type or a complex type. Similarly, <relating> represents a relating type, <inheriting> represents an inheriting type, and <C> represents a connection type; that is, a relating type or an inheriting type. Further explanations are included in the relevant figures as comments.

modelling primitives	meta-metadata	comments
meta-base	<base>	
meta-complex	<complex>	
meta-relating	<relating> <A_1> - <A_2>	the instance of *<relating>* is directed from an instance of *<A_1>* to an instance of *<A_2>*
meta-inheriting	<inheriting> <A_1> - <A_2>	the instance of *<inheriting>* is directed from an instance of *<A_1>* to an instance of *<A_2>*

Figure 8.1 Constructs

modelling primitives	meta-metadata	comments
connection type cardinality	<C> <A_1> [<fmin>,<fmax>] - <A_2> [<tmin>,<tmax>]	the total number of the instances of *<C>* that are directed from a particular instance of *<A_1>* to any instance of *<A_2>* must be between *<fmin>* and *<fmax>*; the total number of the instances of *<C>* that are directed from any instance of *<A_1>* to a particular instance of *<A_2>* must be between *<tmin>* and *<tmax>*
connection types cardinality	<C> <A> [<fmin>,<fmax>] - {<A_1>, ... ,<A_n>} <C> {<A_1>, ... ,<A_n>} - <A> [<tmin>,<tmax>]	the total number of the instances of *<C>* that are directed from a particular instance of *<A>* to any instance of *<A_i>* (i∈{1, ... ,n}) must be between *<fmin>* and *<fmax>* the total number of the instances of *<C>* that are directed from any instance of *<A_i>* (i∈{1, ... ,n}) to a particular instance of *<A>* must be between *<tmin>* and *<tmax>*
connection acyclicity	<C> {boolean}	*<C>: {true}* prohibits any cycle consisting of instances of *<C>*

Figure 8.2 Metadata Constraints

modelling primitives	meta-metadata	comments
base class uniqueness	<base> {boolean}	*<base>: {true}* demands that each instance of a base class specialised from *<base>* must take a distinct value
base class range	<base> {boolean,integer,real,string}	each instance of a base class specialised from *<base>* must take its value from a subset of {boolean,integer,real,string}
base/complex class cardinality	<A> [<min>,<max>]	the total number of the instances of a base/complex class specialised from *<A>* must be between *<min>* and *<max>*
connection class cardinality	<C> <A_1> [<fmin>,<fmax>] - <A_2> [<tmin>,<tmax>]	the total number of the instances of a particular connection class specialised from *<C>* that are directed from a particular instance of the class specialised from *<A_1>* to any instance of the class specialised from *<A_2>* must be between *<fmin>* and *<fmax>*; the total number of the instances of a particular connection class specialised from *<C>* that are directed from any instance of the class specialised from *<A_1>* to a particular instance of the class specialised from *<A_2>* must be between *<tmin>* and *<tmax>*
connection classes cardinality	<C> <A> [<fmin>,<fmax>] - {<A_1>, ... ,<A_n>}	the total number of the instances of all the connection classes which are specialised from *<C>* and directed from a particular class specialised from *<A>* to any class specialised from *<A_i>* ($i \in \{1, ... ,n\}$) that are directed from a particular instance of the class specialised from *<A>* to an instance of a class specialised from *<A_i>* ($i \in \{1, ... ,n\}$) must be between *<fmin>* and *<fmax>*
	<C> {<A_1>, ... ,<A_n>} - <A> [<fmin>,<fmax>]	the total number of the instances of all the connection classes which are specialised from *<C>* and directed from any class specialised from *<A_i>* ($i \in \{1, ... ,n\}$) to a particular class specialised from *<A>* that are directed from an instance of a class specialised from *<A_i>* ($i \in \{1, ... ,n\}$) to a particular instance of the class specialised from <A> must be between *<tmin>* and *<tmax>*

Figure 8.3 Data Constraints (Part I)

modelling primitives	meta-metadata	comments
connection classes combination cardinality	<C> <A> - {<A_1>, ... ,<A_n>} [<tmin>,<tmax>]	the total number of the instances of a particular combination which consists of the connection classes each of which is specialised from *<C>* and is directed from a common class specialised from *<A>* to a class specialised from *<A_i>* (i∈{1, ... ,n}) that connect some instance of the common class and a particular combination of instances of other relevant classes must be between *<tmin>* and *<tmax>*
	<C> {<A_1>, ... ,<A_n>} [<fmin>,<fmax>] - <A>	the total number of the instances of a particular combination which consists of the connection classes each of which is specialised from *<C>* and is directed from a class specialised from *<A_i>* (i∈{1, ... ,n}) to a common class specialised from *<A>* that connect some instance of the common class and a particular combination of instances of other relevant classes must be between *<fmin>* and *<fmax>*
	<C> <A> - {<A_1>, ... ,<A_n>} (full) [<tmin>,<tmax>]	the total number of the instances of the unique combination which consists of all the connection classes each of which is specialised from *<C>* and is directed from a common class specialised from *<A>* to a class specialised from *<A_i>* (i∈{1, ... ,n}) that connect some instance of the common class and a particular combination of instances of other relevant classes must be between *<tmin>* and *<tmax>*

Figure 8.4 Data Constraints (Part II)

8.3 Constructing the Relational Model

The construction of the relational model consists of defining *domains*, *tuples* and *relations* together with the relationships between these and the constraints which must hold. Figures 8.5, 8.6 and 8.7 show how this is achieved in the CDMS. Each row shows one specialisation of a global model construct. The rows in Figure 8.5 show the structural constructs, while the rows in Figures 8.6 and 8.7 show the metadata constraints and data constraints respectively.

A domain is a set of base values, which are booleans, integers, reals or strings and fall within a limited range. Tuples and relations are both complex values; that is, they are made up of components, but do not have object identity. The only two relationships existing are those which connect a tuple to its attribute domains and those which group tuples into relations.

Two metadata constraints limit the ways in which domains, tuples and relations may combine at the schema level:

- A tuple has one or more attribute domains; the same domain can be attribute of one or more types of tuple.

- A relation consists of tuples of the same type; all relations having the same tuple type are of the same type.

The first two data constraints specify the nature of domains, other data constraints limit relationships at the data level - i.e. what values are allowed in a tuple and what tuples are allowed in a relation.

- One domain of a tuple has no more than one value; while a domain value may appear any number of times in different tuples.

- A relation is a set of tuples; that is, no duplicate tuples are allowed in a relation.

- No two tuples have exactly the same attribute values; that is, the full set of domain values must be unique and form a candidate key.

- A subset of domain values may be required unique, thus forming a candidate key.

A relation should be given a name as it is created.

modelling primitives	meta-metadata	comments
meta-base	domain	a domain contains base values
meta-complex	tuple relation	tuples and relations contain non-base values
meta-relating	attribute tuple - domain	a tuple relates to domains as its attributes
	grouping relation - tuple	a relation relates to tuples as its components

Figure 8.5 The Relational Model - Constructs

modelling primitives	meta-metadata	comments
connection type cardinality	attributing tuple [1,n] - domain [1,n] grouping relation [1,1] - tuple [1,1]	 a tuple has at least one domain, while a domain is in at least one tuple one-to-one correspondence exists between tuples and relations

Figure 8.6 The Relational Model - Metadata Constraints

modelling primitives	meta-metadata	comments
base class uniqueness	domain {true}	 a domain is a set of values
base class range	domain {boolean, integer, real, string}	 these base values are allowed to be further instantiated
connection class cardinality	attributing tuple [0,1] - domain [0,n] grouping relation [0,n] - tuple [1,1]	a particular domain of a tuple has at most one value, null value allowed, while a domain value may occur any number of times in tuples a relation may have any number of tuples, while a tuple belongs to a relation only once
connection classes combination cardinality	attributing tuple - {domain} (full) [0,1] tuple - {domain} [0,n]	 no two tuples have exactly the same value a subset of domain values may be required unique, forming a candidate key

Figure 8.7 The Relational Model - Data Constraints

8.4 Constructing the Entity-Relationship Model

The construction of the entity-relationship model consists of defining *entity*, *weak entity*, *simple attribute*, *composite attribute*, *relationship*, *identifying relationship* together with the relationships between these and the constraints which must hold. Figures 8.8, 8.9 and 8.10 show the way in which this is achieved in the CDMS. Again, each row shows one specialisation of a global model construct. The rows in Figure 8.8 show the structural constructs, while the rows in Figures 8.9 and 8.10 show the metadata constraints and data constraints respectively.

The main concepts of the ER model are entities with attributes, and relationships among entities. A weak entity does not have any key attributes of its own. An identifying relationship relates a weak entity to its identifying entities, so that instances of the weak entity can be identified. Attributes can be subdivided into simple attributes and composite attributes.

A simple attribute consists of base values. Composite attributes, entities, weak entities, relationships and identifying relationships are all complex values. Each of entities, weak entities, relationships and identifying relationships may have its own simple attributes and/or composite attributes. Each composite attribute consists of simple attributes and/or composite attributes. Each of relationships and identifying relationships relates to entities and/or weak entities.

The power of the ER model is that uses of the constructs are highly constrained. At the type level, there are connection type cardinality constraints on *relating*, and connection types cardinality constraints on *consisting* and *relating*:

- An identifying relationship relates to at least one weak entity, while a weak entity is related to at least one identifying relationship.

- A composite attribute consists of at least one simple attribute or composite attribute.

- A relationship relates to at least two entities or weak entities. One entity or weak entity is allowed to participate more than once if this entity or weak entity plays a different role each time.

- An identifying relationship also relates to at least two entities or weak entities. Again, one entity or weak entity is allowed to participate more than once if this entity or weak entity plays a different role each time.

At the class level, there are more constraints. Base class uniqueness constraints are fixed on simple attributes, whereas base class range constraints are offered as facilities to be instantiated as a schema is defined. Connection class cardinality constraints on *attributing* are also fully offered as facilities to be instantiated as a schema is defined. Referring back to Figure 4.23 under the line *connection class cardinality*, for instance, *person's name: [1,3], [0,n]* represent a multi-valued attribute, which requires that each person have one, two or three names. Similarly, *person's age: [1,1], [0,n]* represent a single-valued attribute. In these two examples, null values are prohibited. *Person's address [0,3], [0,n]* is another example of a multi-valued attribute. This time, however, there is no null value prohibition. *Number of book: [1,1], [1,1]* is an example of exhaustiveness, which requires that each id must be related to a person. *Title of book: [1,3], [1,n]* is an example of overlapping, which states that more than one book may have the same title.

On the other hand, connection class cardinality constraints on *consisting* restrict multi-valued components, while still being offered as facilities for other purposes. Connection class cardinality constraints on *relating* hold as model inherent constraints, requiring that concerning a particular relating class, a relationship (either ordinary or identifying) instance must relate to exact one entity (either ordinary or weak) instance. In addition, they are provided as model implicit constraints; that is, their instantiated forms will represent the structural constraints.

Furthermore, connection classes combination cardinality constraints on *attributing* are offered as facilities, which may represent key attributes.

modelling primitives	meta-metadata	comments
meta-base	simple attribute	a simple attribute consists of base values
meta-complex	entity weak entity composite attribute relationship identifying relationship	these contain non-base values
meta-relating	attributing entity - simple attribute entity - composite attribute weak entity - simple attribute weak entity - composite attribute relationship - simple attribute relationship - composite attribute identifying relationship - simple attribute identifying relationship - composite attribute	an entity, weak entity, relationship or identifying relationship may have simple attributes and/or composite attributes
	consisting composite attribute - simple attribute composite attribute - composite attribute	a composite attribute consists of simple attributes and/or composite attributes
	relating relationship - entity relationship - weak entity identifying relationship - entity identifying relationship - weak entity	a relationship or identifying relationship relates to entities and/or weak entities

Figure 8.8 The Entity-Relationship Model - Constructs

modelling primitives	meta-metadata	comments
connection type cardinality	relating identifying relationship [1,n] - weak entity [1,n]	an identifying relationship relates to at least one weak entity, while a weak entity is related to at least one identifying relationship
connection types cardinality	consisting composite attribute [1,n] - {simple attribute, composite attribute}	a composite attribute consists of at least one simple attribute or composite attribute
	relating relationship [2,n] - {entity, weak entity} identifying relationship [2,n] - {entity, weak entity}	a relationship or identifying relationship relates to at least two entities and/or weak entities

Figure 8.9 The Entity-Relationship Model - Metadata Constraints

modelling primitives	meta-metadata	comments
base class uniqueness	simple attribute {true}	a simple attribute is a set of values
base class range	simple attribute {boolean, integer, real, string}	these base values are allowed to be further instantiated
connection class cardinality	attributing entity [0,n] - simple attribute [0,n] entity [0,n] - composite attribute [0,n] weak entity [0,n] - simple attribute [0,n] weak entity [0,n] - composite attribute [0,n] relationship [0,n] - simple attribute [0,n] relationship [0,n] - composite attribute [0,n] identifying relationship [0,n] - simple attribute [0,n] identifying relationship [0,n] - composite attribute [0,n]	fully open to schema definition when being instantiated, they may represent single or multi-valued attributes, prohibition of null values, exhaustiveness, overlapping etc
	consisting composite attribute [0,1] - simple attribute [0,n] composite attribute [0,1] - composite attribute [0,n]	multi-valued components prohibited; may represent prohibition of null values, exhaustiveness, overlapping etc
	relating relationship [1,1] - entity [0,n] relationship [1,1] - weak entity [0,n] identifying relationship [1,1] - entity [0,n] identifying relationship [1,1] - weak entity [0,n]	in terms of a particular relating one relationship instance relates to one entity instance, when being instantiated they may represent structural constraints
connection classes combination cardinality	attributing entity - {simple attribute, composite attribute} [0,n] weak entity - {simple attribute, composite attribute} [0,n] relationship - {simple attribute, composite attribute} [0,n] identifying relationship - {simple attribute, composite attribute} [0,n]	open to schema definition when being instantiated, they may represent candidate keys

Figure 8.10 The Entity-Relationship Model - Data Constraints

Unfortunately, some useful applications can not be described by the ER model defined above. One example is shown in Figure 8.11, where a *book* is only able to be one of: on order from *supplier*, at the *binder*, or on loan to a *person*. This condition is named an exclusive alternative relationship.

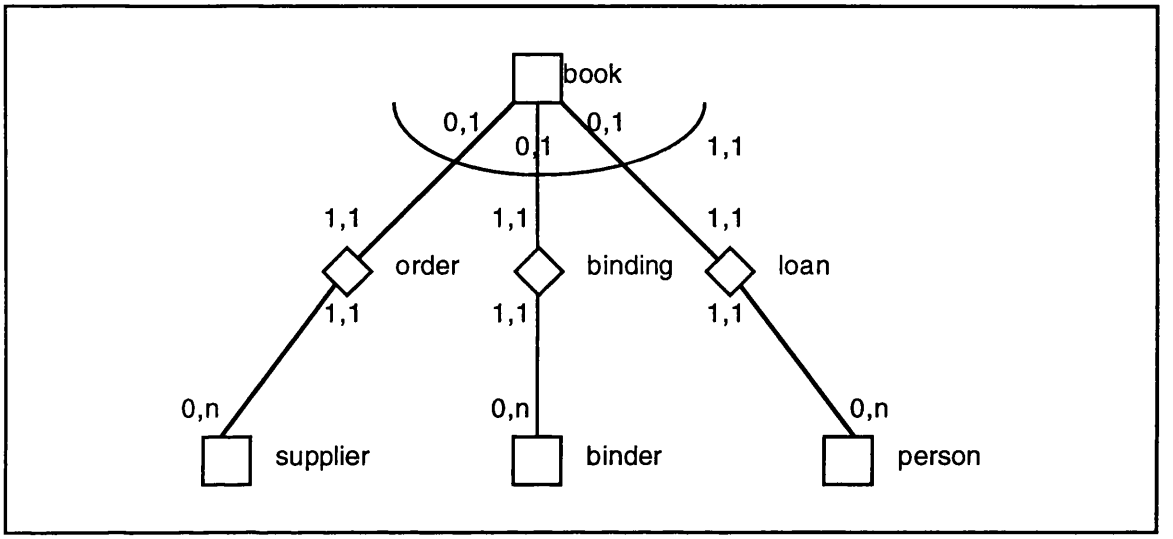

Figure 8.11 Exclusive Alternative Relationship

In order to describe this kind of constraint, the model has to be modified and expanded. Firstly, the connection class cardinality constraint *relating: relationship [1,1] - entity [0,n]* should become *relating: relationship [0,1] - entity [0,n]*. Secondly, the following connection classes cardinality constraint should be added: *relating: {relationship} - entity [0,n]*. The schema then will be constrained by:

- order [1,1] - book [0,1];
- binding [1,1] -book [0,1];
- loan [1,1] - book [0,1]; and
- {order, binding, loan} - book [1,1].

These mean that a book may or may not be related to an order, may or may not be related to a binding, and may or may not be related to a loan, but a book must be related to one instance from order, binding or loan. Again, this example well demonstrates the power of the CDMS, within which appropriate data models may probably be created to suit the various application situations in the real world.

8.5 Constructing the Functional Data Model

The construction of the functional data model consists of defining *printable*, *entity*, *combination* together with the relationships between these and the constraints which must hold. This process is shown in Figures 8.12, 8.13 and 8.14. Each row shows one specialisation in the CDMS. The rows in Figure 8.12 show the structural constructs, while the rows in Figures 8.13 and 8.14 show the metadata and data constraints respectively.

The main concepts of the functional data model are *entities* and *functions*. In order to represent multi-argument functions, an assistant type called *combination* is included, which is, in fact, the Cartesian product of multiple entities.

At the type level, there are connection type cardinality constraints on *combining* and connection types cardinality constraints on *function*. In fact, a combination, which combines at least two entities, must have at least one function. An example of combination is the Cartesian product of student and course. Only by introducing this combination can a function named grade with both arguments, student and course, be defined. Nevertheless, if no function is intended to be defined on these, then there should be no reason to introduce such a combination. It should be noted that one entity can be used more than once in a given combination if this entity plays a different role each time.

At the class level, connection class cardinality constraints on *function* are offered as facilities. These constraints, when being instantiated, may represent various meanings as commented in the figure. This is similar to the situation of the ER model, which has been analysed item by item in the last subsection. Connection class cardinality constraints on *combining* require that a combination instance always combine exactly one instance from each combined entity class, while the concept of exhaustiveness, overlapping can still be represented on the other side as the constraints are instantiated. Connection classes combination cardinality constraints hold on both *function* and *combining*. The former is a facility, which can be instantiated for representing candidate keys, while the latter requires that there is one to one correspondence between a combination and its component entities.

modelling primitives	meta-metadata	comments
meta-base	printable	a printable contains base values
meta-complex	entity combination	entities and combinations contain non-base values
meta-relating	function entity - printable printable - entity entity - entity combination - printable combination - entity combining combination - entity	possible domain and range couples in terms of a function a combination combines entities

Figure 8.12 The Functional Data Model - Constructs

modelling primitives	meta-metadata	comments
connection type cardinality	combining combination [2,n] - entity	a combination combines at least two entities
connection types cardinality	function combination [1,n] - {printable, entity}	a combination has at least one function

Figure 8.13 The Functional Data Model - Metadata Constraints

modelling primitives	meta-metadata	comments
base class uniqueness	printable {true}	a printable is a set of values
base class range	printable {boolean, integer, real, string}	these base values are allowed to be further instantiated
connection class cardinality	function entity [0,n] - printable [0,n] printable [0,n] - entity [0,n] entity [0,n] - entity [0,n] combination [0,n] - printable [0,n] combination [0,n] - entity [0,n] combining combination [1,1] - entity [0,n]	fully open to schema definition when being instantiated, they may represent single or multi-valued functions, prohibition of null values, exhaustiveness, overlapping etc a combination instance combines exactly one instance from each combined entity class
connection classes combination cardinality	function entity - {printable, entity} [0,n] combination - {printable, entity} [0,n] combining combination - {entity} (full) [1,1]	open to schema definition they may represent candidate keys one-to-one correspondence between instances of a combination and its component entities

Figure 8.14 The Functional Data Model - Data Constraints

8.6 Constructing IFO

IFO is constructed using the global data model in this section. Figure 8.15 shows the construct part. Figure 8.16 shows the metadata constraint part, while both Figure 8.17 and Figure 8.18 show the data constraint part.

The main concepts of IFO are atomic types - *printable*, *abstract* and *free*; and type constructors - *set* and *aggregate*. In an IFO schema, the instances of these are properly connected by instances of *attribution*, *grouping*, *aggregation*, *specialisation* and *generalisation*.

Printables are base values. Abstracts are used for atomic objects with no underlying structure. Free types are inherited from other types. Sets represent multi-valued objects, and aggregates represent single objects consisting of component parts (refer back to Subsection 2.3.2).

At the metadata level, there are connection types cardinality constraints and connection acyclicity constraints.

The connection types cardinality include the following:

- A set node relates to exactly one node by grouping; the node being related to can be printable, abstract, free, set or aggregate (refer to Figure 4.17).

- an aggregate node aggregates one or more nodes, which may be printable, abstract, free, set and/or aggregate (refer to Figure 4.18).

- A free node is either specialised or generalised from one or more abstract, free, set and/or aggregate nodes.

At the class level, there are base class uniqueness constraints, base class range constraints, connection class cardinality constraints, connection classes cardinality constraints and connection classes combination cardinality constraints. Their forms are given in the figures.

It is similar to the case in the entity-relationship model that all connection class cardinality constraints on attribution are fully open to schema definition. As such a constraint is instantiated, it may represent single-valued attribute, multi-valued attribute, and possibly prohibit null values at one end, and represent exhaustiveness and overlapping at the other.

Connection class cardinality constraints on grouping are open to schema definition. As such a constraint is instantiated, it may represent the range of number with which an instance of a set groups instances. For instance, a research team relating to research staff may group 10 to 20 individual research workers.

Connection class cardinality constraints on aggregation require that an instance of an aggregate node relates to exactly one instance of each associated node (refer to Figure 4.18). That is, for instance, an address which aggregates house, street and city must consist of one house number, one street name and one city name. Other examples can be seen in Figure 4.14, where an instance of a relationship class relates to exactly one instance of each participating entity class. In the same figure, an instance of a composite attribute class contains exactly one instance of each consisting attribute class.

Both connection class cardinality constraints on specialisation and generalisation require that an instance of subtype must exist in its supertype, that is what inheritance means.

Furthermore, connection classes cardinality constraints on specialisation are open to schema definition. For instance, *specialisation: free [0,n] - {free}* can be instantiated to be the following:

- *specialisation: vehicle [0,1] - {two-wheeler, three-wheeler}* requires that a vehicle must not be both two-wheeler and three-wheeler, thus the specialisation is a disjoint one (refer to Figure 4.10).

- *specialisation: car [1,n] - {drivable, faulty}* requires that a car must be drivable and/or faulty, thus the specialisation has the covering nature (refer to Figure 4.11).

- *specialisation car [1,1] - {manual, automatic}* requires that a car must be either manual or automatic, but not both, thus the specialisation has the disjoint covering nature (refer to Figure 4.12).

Connection classes cardinality constraints on generalisation requires that any generalisation must have the disjoint covering nature (refer to figure 4.15), of which Figure 4.16 is an example.

Connection classes combination cardinality constraints on attribution and aggregation are both open to schema definition. When such a constraint is instantiated, it may represent a candidate key. In this case the integer couple [0,n] becomes [0,1]. An example of this is shown in Figure 4.13, where *title* and *author's name* forms a candidate key of *book*; that is, no more than one book may have a particular combination of title and author's name.

modelling primitives	meta-metadata		comments
meta-base	printable		a printable node contains base values
meta-complex	abstract; free set; aggregate		abstract, free, set and aggregate nodes contain non-base values
meta-relating	attribution		
	abstract - printable		an attribution is directed from a complex node to any node
	abstract - abstract;	abstract - free	
	abstract - set;	abstract - aggregate	
	free - printable		
	free - abstract;	free - free	
	free - set;	free - aggregate	
	set - printable		
	set - abstract;	set - free	
	set - set;	set - aggregate	
	aggregate - printable		
	aggregate - abstract;	aggregate - free	
	aggregate - set;	aggregate - aggregate	
	grouping		
	set - printable		a grouping is directed from a set node to any node
	set - abstract;	set - free	
	set - set;	set - aggregate	
	aggregation		
	aggregate - printable		an aggregation is directed from an aggregate node to any node
	aggregate - abstract;	aggregate - free	
	aggregate - set;	aggregate - aggregate	
meta-inheriting	specialisation		
	abstract - free;	free - free	a specialisation is directed from a complex node to a free node
	set - free;	aggregate - free	
	generalisation		
	abstract - free;	free - free	a generalisation is directed from a complex node to a free node
	set - free	aggregate - free	

Figure 8.15 IFO - Constructs

modelling primitives	meta-metadata	comments
connection types cardinality	grouping set [1,1] - {printable, abstract, free, set, aggregate} aggregation aggregate [1,n] - {printable, abstract, free, set, aggregate} {specialisation abstract, free, set, aggregate} or {generalisation abstract, free, set, aggregate} - free [1,n]	a set node relates to exactly one printable, abstract, free, set or aggregate node by grouping an aggregate node aggregates at least one printable, abstract, free, set and/or aggregate nodes a free node is either specialised or generalised from at least one abstract, free, set and/or aggregate nodes
connection acyclicity	specialisation {true} generalisation {true}	instances of specialisation form no cycle in a schema instances of generalisation form no cycle in a schema

Figure 8.16 IFO - Metadata Constraints

modelling primitives	meta-metadata		comments
base class uniqueness	printable {true}		a printable node is a set of values
base class range	printable {boolean, integer, string, picture}		these are allowed to be further instantiated
connection class cardinality	attribution abstract [0,n] - printable [0,n]		fully open to schema definition
	abstract [0,n] - abstract [0,n];	abstract [0,n] - free [0,n]	when being instantiated,
	abstract [0,n] - set [0,n];	abstract [0,n] - aggregate [0,n]	they may represent
	free [0,n] - printable [0,n]		single or multi-valued attributes,
	free [0,n] - abstract [0,n];	free [0,n] - free [0,n]	prohibition of null values,
	free [0,n] - set [0,n];	free [0,n] - aggregate [0,n]	exhaustiveness,
	set [0,n] - printable [0,n]		overlapping etc
	set [0,n] - abstract [0,n];	set [0,n] - free [0,n]	
	set [0,n] - set [0,n];	set [0,n] - aggregate [0,n]	
	aggregate [0,n] - printable [0,n]		
	aggregate [0,n] - abstract [0,n];	aggregate [0,n] - free [0,n]	
	aggregate [0,n] - set [0,n];	aggregate [0,n] - aggregate [0,n]	
	grouping		open to schema definition
	set [0,n] - printable [0,n]		when being instantiated, they may represent
	set [0,n] - abstract [0,n];	set [0,n] - free [0,n]	the range of number with which an instance
	set [0,n] - set [0,n];	set [0,n] - aggregate [0,n]	of a set groups instances, etc
	aggregation		
	aggregate [1,1] - printable [0,n]		an instance of an aggregate relates to exactly one
	aggregate [1,1] - abstract [0,n];	aggregate [1,1] - free [0,n]	instance of each associated node
	aggregate [1,1] - set [0,n];	aggregate [1,1] - aggregate [0,n]	
	specialisation		
	abstract [0,n] - free [1,1];	free [0,n] - free [1,1]	an instance of subtype exists in its supertype
	set [0,n] - free [1,1];	aggregate [0,n] - free [1,1]	
	generalisation		
	abstract [1,1] - free [0,1];	free [1,1] - free [0,1]	an instance of subtype exists in its supertype
	set [1,1] - free [0,1];	aggregate [1,1] - free [0,1]	

Figure 8.17 IFO - Data Constraints (Part I)

modelling primitives	meta-metadata	comments
connection classes cardinality	specialisation abstract [0,n] - {free}; set [0,n] - {free}; free [0,n] - {free} aggregate [0,n] - {free} generalisation {abstract, free, set, aggregate} - free [1,1]	open to schema definition when being instantiated they may represent disjointness, coverage, partition etc an instance of supertype exists in one of its subtypes
connection classes combination cardinality	attribution abstract - {printable, abstract, free, set, aggregate} [0,n] free - {printable, abstract, free, set, aggregate} [0,n] set - {printable, abstract, free, set, aggregate} [0,n] aggregate - {printable, abstract, free, set, aggregate} [0,n] aggregation aggregate - {printable, abstract, free, set, aggregate} [0,n]	open to schema definition when being instantiated they may represent candidate keys when being instantiated they may represent candidate keys

Figure 8.18 IFO - Data Constraints (Part II)

8.7 Conclusions

The CDMS provides a platform on which different data models and interfaces can be defined within an integrated environment. This facilitates the following practice:

- Since different models and interfaces can be created and modified in a single environment without repetitive coding, they can be easily tested and compared for their suitability to particular applications and particular user groups.

- Potentially, with further development of the system, the data reflecting the same miniworld should be easier to access from different data models and/or user interfaces. This has particular relevance as has been analysed in Chapter 2 for federated database systems.

However, the current implementation of the CDMS is still incomplete. The variety and management of the constraints the system is able to deal with are limited, while the active aspects are still missing.

A fuller analysis on the contributions and further work will be given in Chapter 9 - the concluding chapter.

9 Conclusions

This thesis addresses the general issues of how to provide multiple data modelling facilities in an integrated environment. To this end, the semantic data modelling methodologies were reviewed, based on which a theory of Configurable Data Modelling is proposed and this provides a sound ground for the construction of a Configurable Data Modelling System. The main contributions of the CDMS depend on the decomposibility and reconfigurability of data models. This approach was then extended to encompass other data models.

In the context of the configuration of data models, the role of constraints has been given a strong emphasis. That is, both the configuration of the inherent constraints and that of the facilities for specifying implicit constraints constitute an important part of the concept of the configuration of data models. Thus it is possible to constrain the schemata which can be specified using the data model. Alternatively it is possible to put constraint specifying facilities into the data model.

Behavioural issues have also been given due attention in this research. In general, configuring a data model includes configuring the operations with which the user manipulates the data model. However, owing to time limitation, implementation work on this respect has to be deferred into potential further research projects. This will be referred to in the further work section.

The ability to configure the user interface has also been tested in the system. However, this has not been the main focus of the work and really requires a significant effort in its own right - this will be another topic for further work.

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 Global Data Model, but also in the implementation of the demonstratable CDMS.

The chapter concludes with a discussion of current and further research activities relating to the CDMS.

9.1 Contributions

The CDMS is an unusual DBMS in that it enables users to create and maintain various data models, data schemata and databases. In short, the CDMS supports the processes of creating as well as using data models for database applications.

The CDMS approach is distinguished from the traditional DBMS approach by a number of characteristics. These can be summarised as follows:

1) **Self-containment**. The CDMS contains not only the data and schema of a database but also a complete definition of the data model which supports the schema. Just as a data schema is represented by **metadata**, a data model is represented by **meta-metadata**. The CDMS software describes the databases, schemata and data models in terms of the meta-metadata. The same software is thus able to access databases supported not only by different data schemata but also by different data models.

2) The configurability and management of **integrity constraints** that must hold on the metadata and data. This means that the CDMS can carry out sophisticated treatment of metadata and data correctness to ensure that the application more accurately reflects the situation of the relevant miniworld. This can be done automatically to reduce the users' burden.

3) **Program-metadata independence**. CDMS access programs are written to function independently of any specific metadata files. Owing to the existence of the structure of metadata in the system, there is no need for the structure of any metadata to be embedded in the access programs. Nevertheless the structure of the meta-metadata must be involved in the CDMS access programs to permit the programs to refer to the structure of metadata as necessary.

4) Support of **multiple data models**. A traditional DBMS supports only one or a few fixed data models, while the CDMS permits various data models to be configured and then used within its integrated environment. Federated database systems support multiple data models as well, but in a limited manner. A comparison between the CDMS approach and the federated database system approach is given in Figure 9.1 below.

Methodology	the CDMS	Federated Database Systems
Semantic Data Model Supportability	Yes	
Heterogeneity	Flexible	Fixed
Kernel Model	Yes (deliberately created)	Yes for one architecture only
Model Configurability	Yes	No
Interface Configurability	Yes	
Model Visibility	Yes	
Constraint Management	Yes	

Figure 9.1 Comparison between the CDMS and FDSs

5) Support of **multiple user interfaces**. A traditional DBMS provides only a few fixed interfaces, while the CDMS permits the configuration and utilisation of various interfaces to the same data model and then used within its integrated environment.

6) **Sharing of data**. The further development of the CDMS should allow multiple users to access a database via different data models and interfaces.

The principal contributions of the research are:

1) the creation of a coherent meta-model of the structural constructs found in data models;

2) a coherent analysis of the nature of constraints in database systems, including how they are structured and where they are placed;

3) a similar analysis of the behavioural aspects of database applications from the operations available to the user through transaction systems down to low-level details;

4) an understanding of how to manage a variety of user interfaces to the above;

5) the development of an architecture in which the above can be configured without recourse to low-level programming;

6) the implementation of such a system for managing structure and constraints.

The global data model can also be used in a non-persistent environment owing to its independent nature.

On the one hand, the CDMS can be implemented in a non-persistent language. The ability to build interpreters and store them as files covers the reflection. The generation of a support tool to manage the set of interpreters should cover the software engineering aspects. On the other hand, non-persistent applications may potentially be generated using the global data model. The objective contents and inner-relationships of the real world may actually be abstracted and further analysed based on the constructs and constraints offered by the global data model. Using the concepts offered by the global data model, for instance, different data models can be created, their suitability for describing the miniworld in various application circumstances can be compared with each other, and the data models can then be refined. No database instance has to be involved in this process at all.

9.2 Further Work

As discussed in Chapter 6 and Chapter 7, the CDMS has been realised in the persistent programming language Napier88. One of the main achievements of the CDMS work is the production of a platform for further research and development activities. This is a consequence of the layered, open architecture which facilitates the further improvements of individual parts.

The Global Data Model and its realisation in terms of the CDMS, is seen as a beginning rather than an end. It opens the way for further research work. There are four general directions of research. These are:

1) supporting more complete constraint configuration mechanisms;

   The Global Data Model of the CDMS has a fixed number of facilities for the configuration of structural constraints. These take two general forms. Firstly, there are facilities for the configuration of metadata constraints; secondly, there

are facilities for the configuration of data constraints. As has been demonstrated in Chapter 8, most constraints in prominent semantic data models can be supported by these facilities, but there remain some that cannot. Connection classes combination cardinality constraints have been proposed, but not implemented as yet; while FDM general constraints and the like still have not been analysed in depth. It is intended that a more complete mechanism for the configuration of both metadata and data constraints should be provided in future versions of the CDMS.

2) supporting more sophisticated constraint management mechanisms;

In Chapter 4, a brief description of a general mechanism for checking and managing violations of the constraints was given, but these functions have not been realised fully. It is expected that a sophisticated constraint management sub-system will be developed and involved in future versions of the CDMS.

3) supporting configuration of operation activities involved in the system itself as behavioural aspects;

The behavioural aspect of the CDMS has been hardly touched in the current CDMS implementation. To implement this in the context of the CDMS, careful analysis and synthesis must be done. Chapter 5 constitutes a significant advance in this respect, however, more detailed work is vital to allow the work so far carried out to be realised fully. In particular, the management of constraints can only be achieved effectively by integrating their checking within code fragments - for instance using transaction systems. Moreover, the integration of process modelling with DBMS would be more easily achieved.

4) support for user interface configuration.

The effective delivery of the functionality described above demands effective user interfaces. The DBMS community has severely lagged user interface research and even now is approaching the provision of user interfaces in an *ad hoc* manner. The CDMS approach promises a coherent account of all user interaction with graphical, form-based or textual elements.

The idea of providing a database system in which it is possible for advanced users to determine the ways databases are managed is a new one. This thesis has provided

considerable evidence that such a system can be realised. In order to achieve this, the fundamental activity must be to categorise and analyse the ways in which users conceive of the information they are managing. This include three principal aspects: how the information is structured; how it is constrained; and how it is used. The work presented here has made a significant contribution to this activity by providing a coherent account of these three areas, as well as formally specifying and implementing large parts of the result of these analysis.

Bibliography

[Abiteboul and Hull, 1987]
S. Abiteboul and R. Hull, 'IFO: A Formal Semantic Database Model', *ACM TODS*, 12, 4, 525-565, December 1987.

[Abrial, 1974]
J.R. Abrial, 'Data Semantics', *Data Base Management*, North-Holland, Amsterdam, 1-59, 1974.

[Atkinson *et al*, 1988]
M.P. Atkinson, R. Morrison and O.P. Buneman, 'Binding and Type Checking in Database Programming Languages', *The Computer Journal*, 31, 2 99-109, 1988.

[Atkinson *et al*, 1993a]
M.P. Atkinson, P.W. Trinder and D.A. Watt, 'Bulk Type Constructors', *Fide/93/61*, 1993.

[Atkinson *et al*, 1993b]
M.P. Atkinson, P.J. Bailey, D. Christie, K. Cropper and P.C. Philbrow, 'Towards Bulk Type Libraries for Napier88', *Fide/93/78*, 1993.

[Atzeni and Torlone, 1993]
P. Atzeni and R. Torlone, 'A Metamodel Approach for the Management of Multiple Models and the Translation of Schemes', *Information Systems*, 18, 6, 349-362, 1993.

[Bachman, 1969]
C.W. Bachman, 'Data Structure Diagrams', *Data Base*, 1, 4-10, Summer 1969.

[Bancilhon, 1988]
F. Bancilhon, 'Object-Oriented Database Systems', *Proc of the ACM SIGACT-SIGART Conference on the Principles of Database Systems*, Austing, Texas, May 1988.

[Bernstein, 1995]
P.A. Bernstein, Microsoft Corp, for VLDB 1995.

[Brodie *et al*, 1984]
M.L. Brodie, J. Mylopolous and J.W. Schmidt (eds), 'On Conceptual Modelling', Springer-Verlag, New York, 1984.

[Brown *et al*, 1990]
A.W. Brown, R.K. Took, W.G. Daly, 'Design and Construction of Graphical Interfaces using Surface Interaction', *Proceedings of the 8th British National Conference on Databases*, York, England, Pitman Publishing, 128 Long Acre, London WC2E 9AN, London, England, 243-262, July 1990.

[Bruynooghe *et al*, 1991]
R.F. Bruynooghe, J.M. Parker and J.S. Rowles, 'PSS: A System for Process Enactment', ICL Kidsgrove, Staffordshire, ST7 1TL, UK, 1991.

[Carrick *et al*, 1987]
R. Carrick, A.J. Cole and R. Morrison, 'An Introduction to PS-algol Programming', *Persistent Programming Research Report 31*, Universities of Glasgow and St Andrews, 1987.

[Chen, 1976]
P.P. Chen, 'The Entity-Relationship Model - Toward a Unified View of Data', *ACM TODS*, 1, 1, 9-36, 1976.

[Codd, 1970]
E.F. Codd, 'A Relational Model of Data for Large Shared Data Banks', *CACM*, 13, 6, 377-387, June 1970.

[Cooper, 1987]
R.L. Cooper, 'Applications Programming in PS-algol', Persistent Programming Research Report 25, Universities of Glasgow and St Andrews, 1987.

[Cooper, 1989]
R.L. Cooper, 'Persistent Languages Facilitate the Implementation of Software Version Management', *Proc of 22nd Annual Hawaii Conference on System Sciences*, (ed. B.D. Shriver), vol II, Software, 56-66, January 1989.

[Cooper, 1990]
R.L. Cooper, 'Configurable Data Modelling Systems', *Proc of 9th International Conference on the Entity Relationship Approach*, 35-52, Lausanne, October 1990.

[Cooper, 1993]
R.L. Cooper (ed), 'Interfaces to Database Systems 1992', *Proc. 9th International Conference on the Entity Relationship Approach*, Lausanne, Switzerland, 35-52, October, 1990.

[Cooper, 1994]
R.L. Cooper, 'Configuring Database Query Languages', *Proc. 2nd International Workshop on Interfaces to Database Systems*, P. Sawyer and R.L.Cooper (eds), Springer Verlag, 1994.

[Cooper and Qin, 1990]
R.L Cooper and Z. Qin, 'An Implementation of the IFO Data Model', *Data Modelling Research at Glasgow University 1990-1*, R.Cooper (ed), Technical Report CS 91/R14, Department of Computing Science, University of Glasgow, 17-24, August 1991.

[Cooper and Qin, 1991]
R.L. Cooper and Z. Qin, 'Constraint Management in a Configurable Data Modelling System', *Data Modelling Research at Glasgow University 1990-1*, R.Cooper (ed), Technical Report CS 91/R14, Department of Computing Science, University of Glasgow, 25-38, August 1991.

[Cooper and Qin, 1992]
R.L. Cooper and Z. Qin, 'A Graphical Data Modelling Program with Constraint Specification and Management', *Advanced Database Systems*, P.M.D.Gray and R.J.Lucas (eds), *Lecture Notes in Computer Science 618*, Springer-Verlag, 192-208, July 1992.

[Cooper and Qin, 1994]
R.L. Cooper and Z. Qin, 'A Generic Data Model for the Support of Multiple User Interaction Facilities', *Proc of 13th International Conference on the Entity Relationship Approach*, Manchester, 351-368, December 1994.

[Cooper *et al*, 1987]
R.L. Cooper. M.P. Atkinson, D. Abderrahmane and A. Dearle, 'Constructing Database Systems in a Persistent Environment', *Proc. 13th International Conference on Very large Databases*, Brighton, England, 117-126, September 1987.

[Cox, 1986]
B.J. Cox, 'Object Oriented Programming: An Evolutionary Approach', Addison-Wesley, Reading, Mass, 1986.

[Cutts *et al*, 1989]
Q. Cutts, A. Dearle, G. Kirby and C. Marlin, 'WIN: a Persistent Window Management System', *Persistent Programming Research Report 73*, Universities of Glasgow and St Andrews, 1989.

[Dahl and Nygard, 1966]
O. Dahl and K. Nygard, 'Simula, an Algol-based Simulation Language', *CACM*, 9, 9, 671-678, September 1966.

[Draper and Waite, 1991]
S.W. Draper and K.W. Waite, 'Iconographer as a Visual Programming System', 1991.

[Durand *et al*, 1993]
J. Durand, H. Brunner, R. Cuthbertson, S. Fogel, T. McCandless, R. Sparks and L. Sylvan, 'Data Model and Query Algebra for a Model Based Multi-Model User Interface' in Cooper, 1993.

[Elmasri and Navathe, 1989]
R. Elmasri and S.B. Navathe, 'Fundamentals of Database Systems', Addison Wesley, 1989.

[Fikes and Kehler, 1985]
R. Fikes and T. Kehler, 'The Role of Frame-Based Representation in Reasoning', *Communications of the ACM*, 28, 9, 904-920, September 1985.

[Goldberg and Robson, 1983]
A. Goldberg and D. Robson, 'Smalltalk-80: The Language and Its Implementation', *Addison Wesley*, Reading, Mass, 1983.

[Gomma and Scott, 1981]
H. Gomma and D.B. Scott, Prototyping as a tool in the specifications of user requirements. *Proc of the 5th International Conference on Software Engineering* (San Diego, Calif, Mar), ACM/IEEE, New York, 1981.

[Gray *et al*, 1992]
P.M.D. Gray, K.G. Kulkarni and N.W. Paton: 'Object-Oriented Databases: a Semantic Data Model Approach', Prentice Hall International Series in Computer Science. Prentice Hall, 1992.

[Greenspan, 1984]
S.J. Greenspan, 'Requirements Modelling: A Knowledge Engineering Approach to Software Requirements Definition', *Technical Report CSRG-155*, University of Toronto, March 1984.

[Hammer and McLeod, 1981]
M. Hammer and D. McLeod, 'Database Description with SDM: A Semantic Database Model', *ACM TODS*, 6, 3, 351-386, September 1981.

[Harder and Reuter, 1983]
T. Harder and A. Reuter, 'Principles of Transaction-Oriented Database Recovery', *ACM Comp. Surv. 15, No. 4*, December 1983.

[Hartson and Hix, 1989]
H.R. Hartson and D. Hix, 'Human-Computer Interface Development: Concepts and Systems for Its Management', *ACM Computing Surveys*, 21, 1, 5-92, March 1989.

[Hewitt *et al*, 1973]
C. Hewitt, P. Bishop and R. Steiger, 'A Universal ACTOR Formalism for Artificial Intelligence', *Proc of the International Joint Conference on Artificial Intelligence*, Palo Alto, California, August, 1973.

[Hsiao, 1992]
D.K.Hsiao, 'Federated Databases and Systems', *VLDB Journal*, 1, 127-179 and 2, 285-310, 1992.

[Hull and King, 1987]
R. Hull and R. King, 'Semantic Data Modelling: Survey, Applications and Research Issues', *ACM Computing Surveys*, 19, 3, 201-260, September 1987.

[Kim, 1989]
W. Kim, 'A Model of Queries for Object-Oriented Databases', *Proc. 15th International Conference on Very Large Databases*, Amsterdam, Netherlands, 423-430 September 1989.

[Kent, 1979]
W. Kent, 'Limitation of Record-Based Information Models', *ACM TODS*, 4, 1, 107-131, 1979.

[King and McLeod, 1984]
R. King and D.McLeod, 'A Unified Model and Methodology for Conceptual Database Design', in Brodie *et al*, 1984.

[King and Novak, 1987]
R. King and M. Novak. 'Freedom: A User-Adaptable Form Management System', *Proc. the 13th International Conference on Very Large Databases*, Brighton, England, 331-338, September 1987.

[King and Novak, 1989]
R. King and M. Novak, 'FaceKit: A Database Interface Design Toolkit', *Proc. the 15th International Conference on Very Large Databases*, Amsterdam, Netherlands, 115-123, August 1989.

[Kulkarni and Atkinson, 1987]
K.G. Kulkarni and M.P. Atkinson, 'Implementing an Extended Functional Data Model using PS-algol', *Software Practice and Experience*, 17, 3, 171-185, March 1987.

[Laguna Beach, 1989]
The Laguna Beach Participants, 'Future Directions in DBMS Research', *ACM SIGMOD Record*, 18, 1, 17-26, March 1989.

[Lecluse *et al*, 1988]
C. Lecluse, P. Richard and F. Velez, 'O2, an Object-Oriented Data Model', *Proc of the ACM SIGMOD International Conference on Data Management Systems*, Chicago, 424-433, June 1988.

[Lee, 1992]
D.S.H. Lee, 'Adding Constraint and Active Rule Management to A Semantic Data Model', MSc Dissertation, University of Glasgow, September 1992.

[Leler, 1988]
W. Leler, 'Constraint Programming Languages: Their Specification and Generation', Addison-Wesley, 1988.

[Liscov *et al*, 1977]
B.H. Liskov, A. Snyder, R. Atkinson and C. Schaffert, 'Abstraction Mechanisms in CLU', *CACM*, 20, 8, 564-576, August, 1977.

[Lum,1992]
R. Lum, 'Process Modelling', MSc Dissertation, University of Glasgow, September 1992.

[Maier *et al*, 1986]
D. Maier, J. Stein, A. Otis and A. Purdy, 'Development of an Object-Oriented DBMS', *Proc ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 472-482, September-October 1986.

[Matthews, 1985]
D.C.J. Matthews, 'Poly Manual', *SIGPLAN Notices*, 20, 9, September 1985.

[Meyer, 1988]
B. Meyer, 'Object-Oriented Software Construction', *Prentice-Hall International Series in Computer Science*, 1988.

[Milner, 1984]
R. Milner, 'A Proposal for Standard ML', *Proc of the 1984 Symposium on Lisp and Functional Programming*, Austin, Texas, 1984.

[Morrison *et al*, 1989]
R. Morrison, F. Brown, R. Connor and A. Dearle, 'The Napier88 Reference Manual', *Persistent Programming Research Report 77*, Universities of Glasgow and St Andrews, 1989.

[Mylopoulous *et al*, 1980]
J. Mylopolous, P.A. Bernstein and H.K.T. Wong, 'A Language Facility for Designing Database-Intensive Applications', *ACM TODS*, 5, 2, 185-207, 1980.

[Naish and Thom, 1983]
L. Naish and J. Thom, 'The MU-PROLOG Deductive Database', Technical Report 83/10, Department of Computer Science, University of Melbourne, 1983.

[Nassif, Qiu and Zhu, 1990]
R. Nassif, Y. Qiu, J. Zhu, 'Extending the Object-Oriented Paradigm to support Relationships and Constraints', *IFIP Working Group 2.6 Workshop on Object-Oriented Databases: Analysis, Design and Construction*, Windermere, England, July, 1990.

[Papadimitriou, 1986]
C. Papadimitriou, 'The Theory of Database Concurrency Control', Computer Science Press, 1986.

[Paton *et al*, 1994]
N. Paton, R. Cooper, H. Williams and P. Trinder, 'Advanced Database Systems', Prentice Hall, 1994.

[Peckham and Maryanski, 1988]
J. Peckham and F. Maryanski, 'Semantic Data Models', *ACM Computing Surveys*, 20, 3, 153-189, September, 1988.

[ProcessWise, 1993]
ICL Ltd, ProcessWise Documentation, 1992.

[PS-algol, 1987]
'The PS-algol Reference Manual - Fourth Edition', *Persistent Programming Research Report 12*, University of Glasgow and St. Andrews, 1987.

[Qin, 1993]
Z. Qin, 'Integrating Process Modelling within a Configurable Generic Data Modelling System', IOPENER,Vol 2, no 2, 11-15, Dec 1993.

[Radermacher, 1993]
K. Radermacher, 'An Extensible Graphical Programming Environment for Semantic Modelling", in Cooper,1993.

[Ricardo, 1990]
C. Ricardo, 'Database Systems: Principles, Design and Implementation', McMillan, 1990.

[Richardson and Carey, 1987]
J. Richardson and M.J. Carey, 'Programming Constructs for Database System Implementation in EXODUS', *Proc of the ACM SIGMOD International Conference on Data Management System*, San Francisco, 208-219, May 1987.

[Schmidt, 1977]
J.W. Schmidt, 'Some High-level Language Constructs for Data of Type Relation', *ACM TODS*, 2, 3, 247-261, 1977.

[ServioLogic, 1987]
ServioLogic Corporation, 'Programming in OPAL', 15025, S.W. Koll Parkway, 1A, Beaverton, Oregon 97006, 1987.

[Shipman,1981]
D.W. Shipman, 'The Functional Data Model and the Data Language DAPLEX', *ACM TODS*, 6, 1, 140-173, March 1981.

[Smith and Smith, 1977]
J.M. Smith and D.C.P. Smith, 'Database abstractions: Aggregation and Generalisation', *ACM TODS*, 2, 2, 105-134, 1977.

[Stefic and Bobrow, 1985]
M. Stefic and D.G. Bobrow, 'Object-Oriented Programming: Themes and Variations', *The AI Magazine*, 40-62, 1985.

[Stonebraker *et al*,1993]
M. Stonebraker, R. Agrawal, U. Dayal, E.J. Neuhold and A. Reuter, 'DBMS Research at a CrossRoads: The Vienna Update', *Proc VLDB19*, 688-692, Dublin, August 1993.

[Stroustrup, 1984]
B. Stroustrup, 'The C++ Programming Language', *Addison Wesley*, Reading, Mass, 1984.

[Sutton, 1991]
S.M. Sutton, 'A Flexible Consistency Model for Persistent Data in Software Process Programming Languages', to be in Implementing Persistent Object Bases: Principles and Practice (eds A. Dearle, G.M. Shaw and S.B. Zdonik), Morgan Kaufmann, 1991.

[Tai, 1991]
Y. Tai, 'The Constraint Management Project', MSc Dissertation, University of Glasgow, September 1991.

[Tan, 1991]
L.K. Tan, 'Constraint Management in A Configurable Data Modelling System', MSc Dissertation, University of Glasgow, September 1991.

[Teo, 1991]
L.N. Teo, 'Form Management', MSc Dissertation, University of Glasgow, September 1991.

[Teorey, 1986]
T.J. Teorey, D. Yang and J.P. Fry, A logical design methodology for relational databases using the extended entity-relationship model, *ACM Computer Survey 18*, 2 (June), 197-222, 1986.

[Wasserman and Shewmake, 1982]
A.I. Wasserman and D.T. Shewmake, 'Rapid prototyping of interactive information systems', *ACM SIGSOFT Softw. Eng Notes*, 1-18, December 1982.

[Zloof, 1977]
M.M. Zloof, 'Query-by-Example, A New Data Base Language' in *IBM Systems Journal*, 16, 4, 324-344, 1977.