29

Exploiting persistence in CASE technology

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Ricardo Figueira
February 1997

Abstract

A Design Workbench has been built for Napier88 [MBC+94] as part of the natural progression towards developing better product systems and improving software construction tools. The system includes a Metamodeller (enabling users to specify the data and process models they prefer), a Model Builder which supports multiple co-existing models and a Target System Generator. Experience using the Workbench has shown that it is easy to use, increases productivity, improves programming standards and facilitates code sharing.

This thesis demonstrates the benefits of orthogonal persistence for Computer-Aided Software Engineering by describing an initial design environment and its subsequent extension to include support for multiple co-existing models.

Contents

1 Introduction 7

2 Background 11

2.1 Persistence and Napier88 11

2.2 Software Development Environments 13

2.2.1 Introduction 13

2.2.2 History of CASE 14

2.2.3 Next generation CASE tools 15

2.2.4 Persistence and CASE 16

2.2.5 Conclusion 18

2.3 Data modelling 18

2.3.1 Overview 18

2.3.2 Introduction 18

2.3.3 Review of major data models 19

2.3.4 Comparison 27

2.3.5 Conclusion 27

3 A Prototype Systems Development Environment 28

3.1 The Data Repository 29

3.2 Data Model Component 31

3.2.1 Model Input 31

3.2.2 Type Generation 34

3.2.3 Type Handlers 38

3.3 Process Model Component 40

3.3.1 Structure Chart Specification Language 41
3.3.2 Code Generation 42
3.4 User Interface 42
3.4.1 Graphical Visualisation 42
3.4.2 Design Querying 42
3.5 Conclusion 43

4 Supporting Multiple Co-existing Models **44**
4.1 Choosing Additional Data Models 45
4.2 Integration of a functional model 45
4.2.1 Overview 45
4.2.2 Daplex 46
4.2.3 Integrating FDM in the repository 47
4.2.4 Conclusion 49
4.3 Constraints in an SDE 50
4.3.1 Introduction 50
4.3.2 Constraint representation in the repository 50
4.3.3 Generation of Napier88 code 52
4.3.4 Using constraints 54
4.3.5 Conclusion 54
4.4 Integration of OMT 54
4.4.1 The Object Model 55
4.4.2 The Dynamic Model and the Functional Model 55
4.4.3 Changes to the repository 55
4.5 Co-existing data models in a general data repository 57
4.5.1 Model Integration 58
4.5.2 Changing Model Perspective – Example 58
4.5.3 Conclusion 60

5 The Persistent Workshop **61**
5.1 Workshop abstraction 61
5.2 Integrating the new workbench 62
5.2.1 Updating program code 62
5.2.2 Integration of the development environment 62
5.3 Interaction between workbenches 64
5.4 Benefits 64
5.5 Conclusion 65

6 The Metamodeller **66**
6.1 Model input and edit 67
6.1.1 The Repository Library 67
6.1.2 Model grammars 68
6.1.3 Model Builder generation 70
6.2 Model query 70
6.2.1 View definition 71
6.2.2 Query specification 71
6.2.3 Query execution 72
6.3 Metasystem query implementation 72
6.3.1 Overall approach 72
6.3.2 Data structure 74
6.3.3 Code generation 75
6.4 Conclusion 78

7 Experimentation **80**
7.1 Testing System Utility 80
7.2 Supporting multiple models 81
7.2.1 Accommodating SDM in the data repository 82
7.2.2 Viewing SDM-input designs in other models 83
7.2.3 Designing with different models 84
7.2.4 Conclusion 85
7.3 Change Propogation 85

Bibliography **102**

8 Conclusion **87**

8.1 Thesis Summary 87

8.2 Discussion 88

8.3 Future work 90

8.4 Conclusion 90

A Metasystem experiments **92**

A.1 Viewing in a different model 92

A.2 Using alternative models to specify a model 98

List of Figures

2.1 Three mappings with Two Support Systems 12
2.2 One mapping on an Intergrated Support System 12
2.3 Entity Relationship version of University Database 21
2.4 SDM version of the University Database 23
2.5 Functional Data Model version of University Database 24
2.6 IFO version of University Database 26

3.1 A system overview 29
3.2 The major DR data structures 30
3.3 ER diagram of a parts-supplier model 32
3.4 ER input specification of a parts-supplier model 33
3.5 Example of a read handler: `readPart` using the relational approach . 39
3.6 Example of a read handler: `readPart` using the reference approach . 40
3.7 Structure chart of `orderPart` with its corresponding input specification 41

4.1 Original (ER-based) repository data structure 48
4.2 Data repository representation of `Grade` 49
4.3 Data repository representation of `hasCourses` 49
4.4 Representation of constraint `NativeHead` 51
4.5 `Venue(TA,Student,Lecturer)` is represented using **next-op** nodes 51
4.6 Constraint before FDM functions identified 52
4.7 Constraint after FDM functions identified 52
4.8 Representation of the OMT-based repository 56
4.9 Part of FDM output of an ER model 59

6.1 Context-free ER input grammar 69
6.2 Simplified view definition for an ER model 70
6.3 Model view specification grammar 71
6.4 Diagram of query subsystem data structures 75
6.5 A simplified **output**_modelName procedure 76
6.6 A simplified example of an **output**_table procedure 77

7.1 Similarities of data models implemented 84

A.1 The tanker monitoring application SDM model 93
A.2 The tanker monitoring application OMT model 95
A.3 The tanker monitoring application FDM model 97
A.4 Initial DAPLEX specification of part of part-supplier schema 99
A.5 ER specification of part of part-supplier schema 99
A.6 DAPLEX output of combined DAPLEX and ER part-supplier schema 100
A.7 Subsequent DAPLEX specification of part of schema 101
A.8 Daplex output of combined DAPLEX, ER and DAPLEX schema . . . 101

Chapter 1

Introduction

Persistent object systems allow data to be manipulated independently of longevity, freeing programmers from learning two different languages for transient and database objects respectively, from managing transfers between memory and secondary storage, and from coding type conversions as data migrates between these stores. They claim to be better than conventional database systems at supporting complex data, as needed eg in computer-aided software engineering (CASE). This thesis describes a complex application which substantiates this claim by showing how a sophisticated Systems Design Workbench [BF96] can be built in Napier88 [MBC+94]. The end-product is a useful toolkit for Napier88 programmers, which has been integrated into a structured software construction system called the Persistent Workshop [WPA+95]. The Design Workbench incorporates a metasystem whereby user-defined data and process models can be specified, and permits using multiple co-existing models when designing a single target system.

This chapter explains why CASE systems are needed, and why they are difficult to build using conventional database technology. After outlining the reasons for choosing Napier88 as the specific persistent system to use in our implementation, the current approach to creating Napier88 software is sketched to show how this could benefit from automated aids. The research questions which this work addresses are then presented, along with an overview of the CASE product developed. The chapter concludes with an outline of the thesis.

Software systems are typically roughly designed and then interactively refined and structured so that they can be translated into a working system. Designs are composed of data and process specifications. The information includes relationships between parts of the design (both data and process components). Changes in one design component can affect many others. A designer must know what the relationships are between the components and where change must be propagated through the system. Recording and interpreting these complex relationships can be difficult for a designer since these systems can be large , with many design components and many interrelationships between

them.

The advent of complex systems development methodologies and the need to automate the development process precipitated automated Computer-Aided Software Engineering (CASE) tools. Central to modern CASE tools is a complex data repository of documents, code, diagrams and models. File system and conventional database implementations of such a repository are unsuitable since they cannot represent the complex nature of the design without translating the structures into a flat structure. This process is time-inefficient (due to the translation), space-inefficient (due to the repetition in the stored information), unproductive (since this conversion must be programmed) and prone to errors [IBM78].

Persistent programming languages were designed to support large, complex, long-lived application systems, allowing data of any type to be stored in a database, including complex objects and code. Systems like CASE are named as typical applications [AM95] that could benefit from using persistence. To date, however, very little exists in terms of persistent software engineering [Coo90, Tea93, Wet94, KCCM92, AM95]. Consequently, persistent programmers do not use sophisticated Software Engineering Environments in practice.

Napier88 was chosen as the vehicle for studying persistent CASE because of its data types, its approach to implementing persistence and its many special features such as hyperprogramming and linguistic reflection [MBC+94]. These aspects are described in chapter 2 and their usefulness highlighted in the remainder of the thesis. As programming-oriented CASE tools to support the Implementation Phase of the software lifecycle already existed for Napier88 [WPA+95], a Design Workbench was created to complement this and study the application of persistence in automated *design* tools.

Napier88 systems are typically constructed in the following way. Many textual specifications are developed by team members, on paper and in Unix directories. It is difficult to see which represent alternatives, which are successive refinements of an approach, how they relate to each other and where to find documentation on any one subsystem. It is also difficult for programmers to remain aware of structures and code produced by other team members, so software reuse is limited. Ensuring that all members use the same type definitions and the same interfaces to shared procedures requires some effort; it is harder still to ensure that components adhere to conventions so that the system is easy to understand and modify. Browsers support exploration of the stable store, but typically only give the name and type of objects; a hyperprogramming browser [KCCM92] can show associated source; but additional information is needed before software reuse can confidently be done. Information on types is not available, and specifications and documentation are not generally maintained on the store.

We discern three developing strands of provision for software construction in persistent languages: a bottom-up approach exemplified by hyperprogramming [KCCM92]; a top-down approach (see for example [CT92, Wet94]); and a methodological approach

(for example [ABC+93]). Napier88 has a hyperprogramming browser and a software engineering workshop of program construction tools, such as program builders, compilers, editors and pretty printers, called the Glasgow Workshop. These concentrate primarily on the implementation stage; there is a need to complement them with good systems design aids.

This thesis investigates the utility of persistent systems in automated software design, aiming to answer questions such as

- are persistent languages convenient for building CASE Environments?
- if so, in what ways?
- what features of persistent languages are useful for CASE and what problems exist?
- what trade-offs exist in persistent SE?
- can a persistent CASE system extend the boundaries of CASE technology, and if so, how?
- will persistent programmers use CASE products and what tools would they find most beneficial?

In order for a design toolkit to be used in practice, it must be quick and easy to use, offer significant short-term benefits and permit programmers to use their own favourite methodology [Mar88]. To be used in practice, it is essential that design tools be sufficiently flexible to allow everyone to use the models they prefer. We also believe it is essential that the systems design component be integrated with a good program development environment. Work began with a prototype system that input data (ER) and process (structure chart) models, generated types and useful procedures that manipulated instances of the generated types, produced procedure stubs and contained a query subsystem that allowed the models and generated code to be queried. The system was extended to include other data and process models including object-oriented (OMT [RBP+91]) and functional (Daplex [Shi81]) models and integrated in the Persistent Workshop. The system has been used in several experiments where its usefulness was very encouraging; it shortened the programming task (since parts of the implemention were generated) and querying the repository was beneficial during modifications, particularly to new team members learning about a design in progress.

This thesis describes the design, implementation, experimentation and results of our persistent software design system. Chapter 2 introduces the topics explored – persistent systems, the Napier88 programming language, Computer Aided Software Engineering and existing work in persistent software engineering. Since the Workbench focuses on data aspects, some background on conceptual modelling is also given there. Chapter three describes the prototype modelling system that formed the preliminary

work on which this thesis was based [FIJK94]. Chapter 4 shows how multiple co-existing models can be supported and the next chapter outlines the Workshop and how new workbenches can be added to this. This leads us to describing the Metamodeller in Chapter 6, which permits a specific model to be defined. Chapter 7 discusses some experiments performed with the system, and in the Conclusion we summarise our results and suggest some avenues for future research.

Chapter 2

Background

As this thesis involves the use of persistent systems for automated software environments, we present an overview of these two topics in this chapter. The first section describes Napier88, a persistent programming language. Section 2 introduces CASE environments and the role of persistence in automating software engineering. The chapter concludes with an outline of semantic data modelling, one of the areas of systems design that we have focussed on.

2.1 Persistence and Napier88

Persistent Object Systems (POSs) [AM95] were designed to support large, complex, long-lived application systems such as CASE, CAD and GIS. They support long transactions and arbitrarily complex persistent data. A POS is the underlying database architecture together with some (persistent) programming language.

Orthogonal persistence is defined [AM95] as the achievement of three principles: a program has the same form whether it uses transient or persistent data (*persistence independence*); objects of any type may be persistent (*data type orthogonality*); and persistence identification is independent of the type system (*persistence identification*).

Atkinson and Morrison [AM95] summarise the simplification achieved by orthogonal persistence in Figures 2.1 and 2.2. Figure 2.1 shows a persistent application system that has programming language and database components, with a different model for each component that represent the same real world system. Consistency must be obtained between the models whether accessed directly or via programs. Figure 2.2 shows the simpler model provided by orthogonal persistence of a single model and a single mapping.

Orthogonal persistence ensures that all data of any type (including graphics and object code) can be stored in the database. The programmer is freed from the task of fetching

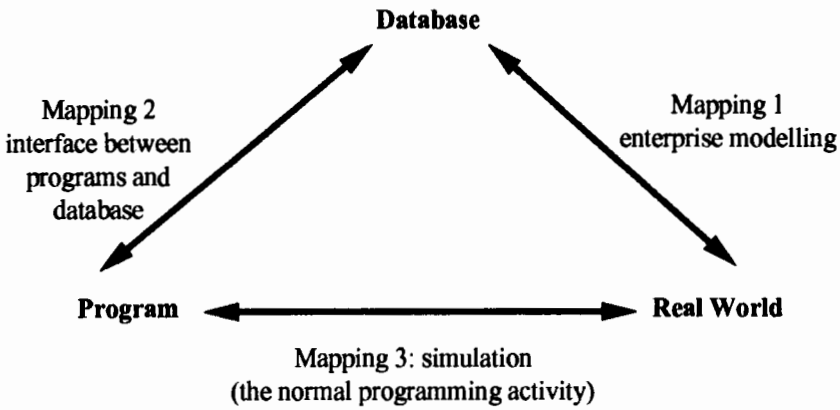


Figure 2.1: Three mappings with Two Support Systems

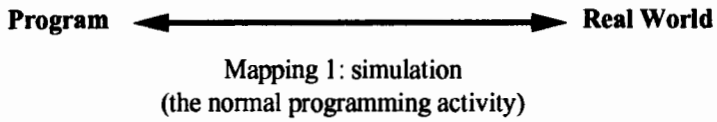


Figure 2.2: One mapping on an Intergrated Support System

and storing data from the database and from converting data from the database representation to the program representation. When persistence is applied through reachability, all objects referenced by a persistent one automatically become persistent too. Making the transitive closure of all references from a persistent root persist, referential integrity is guaranteed. This means that all referenced data is available, producing a safer programming environment.

Persistent programming languages "[reduce] the cost of software and project support throughout its life cycle" [MDB+85]. Persistent application systems (PASs) are easier to design, build and maintain because the persistence abstraction doesn't distinguish between persistent and transient data. In a conventional application, on the other hand, a large amount of effort is spent migrating objects between an external DBMS or files and a program [IBM78].

Napier88 is a strongly-typed persistent programming language that provides facilities [MBC+] for orthogonal persistence, parametric polymorphism, type completeness, first-class procedures, abstract data types, collections of bindings, a strongly typed stable store, graphical data types and linguistic reflection. Persistence is implemented via reachability from a single persistent root, so referential integrity of the stable store is guaranteed. The programming system includes a stable store populated by typed data objects that may be updated atomically.

It has a rich type system including base types (procedures, integers, reals, booleans, strings, environments (called **env**), **any**, images, line drawings (call **pic**), pixels, files and null values) and contructor types (structures, vectors, variants and abstract data types).

The language has parameterised types and procedures allowing parametric polymorphism. It contains an **env** type (environments) that is an extensible set of name–value bindings, and a dynamic type **any**, which is the infinite union of all possible types. Values in environments are dynamically bound and type-checked. Environments give name-space control within Napier88 programs, and the **any** type can be used to delay run-time type checking.

The store is structured as a graph of environments, each environment containing bindings of any type (including procedures and environments). The Napier88 Libraries [KBC+94], which consists of a set of environments of procedures for compiling programs, browsing the persistent store, performing I/O, constructing GUI's, etc., are stored in this way on the store. Strongly-typed runtime linguistic reflection is possible because the compiler is one of the procedures available on the store. The result it returns can be incorporated in the program execution after a call, enabling executions to be tailored to runtime requests.

One extension of the Napier88 Libraries (called the Glasgow Libraries) provides a uniform collection of bulk types such as lists and Maps. A Map [ALPR91] is a set of (*domain,range*) tuples, where *domain* values are unique. For example, the Map `Map[int,string]` has a domain of integers and range of strings. Map initialisation requires two procedures: an equality and an ordering one. These are used by the Library procedures to index Map entries when building Maps and efficiently implementing high-level operations such as filtering, intersection, etc.

2.2 Software Development Environments

2.2.1 Introduction

Computer-Aided Software Engineering (CASE) is the name given to any automated software construction tool, including conceptual system design, project management, configuration management and language-oriented tools (such as program editors and compilers). CASE products were developed to enforce software engineering principles in software development using automated tools. The terms **Integrated Project Support Environment** (IPSE), **Software Development Environment** (SDE) and **Software Engineering Environment** (SEE) all refer to CASE environments that support the entire software lifecycle (from design to implementation).

Dart *et al* [DEFH87] identified the following categories of SDEs: language-centered, structure-oriented, toolkit and method-based environments. The first type are program-

ming language specific tools which usually don't support the full software lifecycle. Structure-oriented SDEs are editor-centered environments that provide syntactic and semantic program information. Toolkit environments are a collection of tools each devoted to a stage in program coding. Method-based SDEs use a particular software engineering method for software development, such as particular conceptual modelling and lifecycle models.

The next subsection outlines the development of CASE environments from the early years to existing systems. Section 2.2.3 describes the next generation, giving some of the challenges facing CASE developers. The section concludes with a discussion on recent research into persistent CASE technology.

2.2.2 History of CASE

Two generations of CASE tools were identified by Norman and Chen [NC92]. The first generation was based on structured system analysis and design methods (eg. Gane and Sarson [GS77] and De Marco [DM78]). These methodologies were first proposed in the mid-1970s, and the automated CASE tools were text-based and ran on mainframes (such as PSL/PSA [TH77]). The tools primarily used data flow diagrams (DFDs) to model system processes and data models such as entity-relationship (ER) diagrams [Che76]. The second generation (developed in the early 1980s) made use of graphical terminals to support graphical notations of structured methods. Project dictionaries were first used in second generation tools to allow different tools within the same CASE environment to share development information. Over subsequent years, with the emergence of new design methodologies, more specialised CASE tools were produced. Better programming tools were also developed during this time, and graphical front-ends and database support were added.

Current CASE technology

Recent CASE environments focus on the integration of tools. This subsection outlines some recent CASE environments, most of which provide mechanisms for extending the SDE to include alternative methodologies.

Garden [Rei87] is a SDE that allows a number of techniques to be used when designing a system. It provides a Lisp-like language to define the type structure of a model's design objects (such as the type structure "entities" in an ER design). The language is also used to define code to evaluate design objects (such as inserting a new entity in an ER diagram). Garden uses an object-oriented database to store the code and designs.

IStar [Dow87] is an integrated workbench that was designed to allow new tools to be easily added to the environment. It contains project and change management facilities together with design and programming tools. A foreign tool is integrated by producing an IStar tool process to interact with the tool.

Genera [WMWM87] is a file-based SDE that allows rapid production of a final system. It uses data-level integration of tools and supports incremental change in designs. It allows programmers to alter the SDE to suit their needs by altering the SDE tools.

HyperCASE [CR92] applies hypertext to the components of CASE, using an integrated framework. It consists of graphical editing, knowledge-base and data dictionary subsystems. The hypertext is used to link the different aspects of the design and implementation, providing a means to navigate the design in an ad hoc way. The data dictionary is stored in a relational database using an embedded query language.

2.2.3 Next generation CASE tools

The first two generations of CASE tools supported aspects of programming (such as editors and compilers) and design (such as conceptual modelling). There are a number of issues put forward [DEFH87, NC92, Gri94, DD95] to be addressed by the next generation of CASE tools.

- **integration**. There is a need for independent tools and design methodologies to be combined in a single system. This is necessary because a number of tools have been developed that cannot be used together, because integrating a new tool in a SDE can be cumbersome, and because separate tools mean that control over the entire software development process is difficult.
- **extensible/meta environments**. Software designers tend to use a number of different software development methods (eg. for concept modelling, software process and project management) when designing and implementing a system [Bro87]. A number of design methodologies exist. These include data modelling methodologies (such as ER, FDM [Shi81], OMT [RBP+91] and SDM [HM81]), process modelling (such as DFD, OMT and Petri net [Pet77]) and software development methodologies (such as Jackson [Jac83], Spiral [Boe88] and Waterfall [Roy70] models). Present tools usually only implement a few of these methodologies in their system. It is desirable that a single SDE could be extended to include any of these methodologies.
- **collaborative** CASE. A number of designers and programmers may be distributed over a network. Such a CASE tool needs to provide mechanisms to allow distribution and sharing of designs.
- **knowledge-based** CASE. These provide intelligent tool support, assisting parts of the system development such as design construction or design correctness. They use domain-specific knowledge to provide a more efficient tool that helps produce higher quality designs [LW94].

The most challenging of these issues is extensible/meta environments. SDEs need to provide either mechanisms for specifying new software development methods (such as the Garden environment) or conceptually simplistic tools that can be interpreted by the designer in any way (such as the "Designer's Notepad" [HS90]). The Designer's Notepad presents a pencil-and-paper view of systems design. A designer identifies the entities and inserts arrows to identify relationships. This unstructured method can be used to produce data or process models. In contrast, Dawson and Dawson [DD95] propose a management structure for the software development models that can accommodate alternative models. This makes use of a non-deterministic structure to represent different software development models.

2.2.4 Persistence and CASE

Present CASE tools either store information in an external database (usually relational) or in a file system.

Software engineering repositories stored in a database (such as PMDB [Pen87] and IS-tar [Dow87]) need a mapping from the database to the CASE view of the data. This has the disadvantage of causing possible inconsistencies. Dowson [Dow87] describes how integrating a database can restrict the extension of IPSEs. File system implementations (such as "Software through Pictures" [WP87]) suffer from the additional disadvantage that objects must be "flattened". Dependencies between data and complex structures must be separated in order for them to be stored.

Software Engineering Databases (SEDBs) need to support the following characteristics of SEEs:

- design data is complex (with many dependencies) and potentially large
- transactions are long
- design data ranges from very small to very large objects

The elements of a SEE design are typically the design objects (such as entities in an ER diagram or stores in a DFD), the diagrams themselves (the image and its internal representation), management information (such as configuration and version control), documentation and program code (source and object). Compared to conventional database applications, design data typically comprises a very large number of different types to cover all design elements, with few instances of each type. Interactive transactions in a SEE (such as during conceptual modelling) are usually long since the transaction can only be committed once several related design components have been created or changed together.

Traditional database management systems (DBMS), such as relational systems, do not usually support these characteristics since they don't directly cater for complex objects and transactions tend to be very short.

Persistent programming languages

CASE tools, like the target systems they help to design, to implement and to maintain, should therefore be easier to develop in an orthogonally Persistent Object System.

Existing work in Persistent Software Engineering

A rather limited number of applications have been developed using orthogonal persistence to support software engineering and extend software engineering research. GOODSTEP [Tea93], an extension of the object-oriented database management system O_2 [O2 93] to support software development environments, includes support for process modelling, analysis and enactment. The STYLE Workbench [WMS95] is a tool for the construction of customized development environments, implemented in the persistent programming language Tycoon [Mat95]. In addition, some work has been undertaken in the context of Napier88 :–

- **Implementing data models** in persistent programming environments. Configurable data modelling [Coo90] allows a model designer to specify the data model constructs and operations required for an application domain. Data model description constructs can be translated to type declarations and the declaration of associated procedures. With the use of linguistic reflection these types and procedures can be rapidly implemented from the data model. This work was extended to deal with constraints in configurable data modelling [CQ91b].

  [CQ91a] has demonstrated how data models can be implemented using persistent programming languages (such as implementing the IFO[AH87] data model), and how these can be used to evaluate data models and allow rapid prototyping of persistent application systems.

- **Visualising software engineering environments**

  Two visualisation tools [Lav94] were created that use human-computer interaction (HCI) methods to visualise the Napier88 persistant store. A "Search Path Editor" was written that generates a Napier88 program preamble (a list of locations of program identifiers) given a set of paths in the store, and a store visualiser was produced that graphically displays the store contents as a graph.

- **hyper-programming**. A hyper-program [KCCM92] is a program that contains both text and persistent values. Hyper-programming is the creation of programs by interspersing source code text with links to persistent objects – a hyper-programming system provides a persistent store browser to facilitate this process. Hyper-programming has the advantage that program checking is done early, links may exist between object code and its associated source, and code becomes more succinct.

- **Glasgow Programmers' Workshop**. The Workshop [AM95] was an experiment to demonstrate how orthogonal persistence can provide extra functionality to automated software development, and to provide a programming environment for persistent programmers. It was built with a single workbench to support program construction, having compilers, editors, build managers, pretty printers, etc. as tools; the work items are programs or type definitions.
- **Build manager**. A build manager helps in the construction of large-scale Napier88 systems. Its design was based on the study of the impact of schema evolution on Napier88 programs[Sjø92]. This study included the analysis of construct usage in Napier88 programs using a meta data management tool called the thesaurus-based information tool (TSIT) [Sjø91].

2.2.5 Conclusion

CASE is the automation of the software development process where tools apply software engineering methodologies to assist in the construction of software.

Proposals for third generation CASE tools include more general and integrated environments with mechanisms to support different methodologies in such a way that different tools can still be used together.

Orthogonally peristent object systems present a model of persistence that simplifies persistent system construction of long-lived complex systems. As such they offer several benefits to CASE implementors.

2.3 Data modelling

2.3.1 Overview

This section presents the major data modelling features and reviews prominent data models, including the Entity-Relationship model, the Semantic Data Model, the Functional Data Model and the IFO model. Each model is described in terms of how data is represented, and the constructs for representing data. A university database is used to describe the models. The section concludes with a comparison of the prominent data models.

2.3.2 Introduction

A data model is a set of abstraction tools used to conceptually represent data and its interrelationships. A good data model is able to conceptually model a database by providing a rich set of constructs to represent the requirements.

Data models can be divided into three categories [KS86]: object-based logical models, record-based logical models and physical data models.

- Examples of object-based logical models are entity relationship [Che76] and semantic data models [AH87].
- Record-based logical models include the relation, network and hierarchical models.
- Physical models

Object-based logical models are the most useful for high level design since they concentrate on abstractions instead of the physical implementation. This section considers only object-based models.

Comparisons of data models emphasize the expressiveness or semantic power of the model's constructs. The two main conceptual tools in data modelling are aggregation and generalization [SS77, PM88].

- **Generalization** allows objects of one type to inherit properties of another type. For example, `Person` is generalization of `Student`. Properties of `Person`, such as `Name` and `Address`, are considered part of (inherited by) `Student`.
- **Aggregation** of a set of objects groups them as a single object. For example, `Person` is an aggregation of `Name` (a string) and `Age` (a number). `Person` is considered a single object.

Other conceptual modelling tools include derivation, classification and association.

- **Derivation** of values in the database allows different views of the database to be defined, by setting a property of an entity to a derived value.
- **Classification** is where instances with the same meaning and behaviour are associated with a single category or type.
- **Association** presents relationships between model objects as objects themselves.

2.3.3 Review of major data models

The major models are described below and their modelling constructs identified.

The running example used to describe the different models is a university database with departments, courses, students and instructors.

Instructors teach multiple courses, courses each have one instructor, students take multiple courses, a course can be presented only in one department, and each department has an instructor as its head.

Persons are generalizations of students and employees, and employees are a generalization of instructors.

The Entity-Relationship Model

The entity relationship model [Che76] is widely used and the model most closely associated with relational databases.

The primary data element is the *entity set*, made up of "printable" values called *attribute sets*. These attributes constitute properties of the entity. If an attribute collection uniquely defines an entity, it is called a key for the entity.

Associations between entities, called "relation sets" (hereafter called *relationships*) map entity sets to other entity sets. They can have role names for each associated entity set and attribute sets can be associated with the relationship. For each entity set related in a particular relationship, a cardinality can be specified. The cardinality of relationships are specified as many-to-many, one-to-many and one-to-one.

The ER model allows for dependent relationships. These are constructed using a relationship to depict a dependence between the two entity sets. The dependent entity set is called "weak". For example, in Figure 2.3, **Dependant** is a weak entity which is related to **Employee**.

Rectangles represent entity sets, diamonds represent relationships, rounded rectangles are attribute sets. Role names of entity sets in relationships are labeled on the arc between the entity set and the relationship. Keys are represented as underlined atttribute sets. Weak entity sets are represented with double rectangles.

Extended ER models [PM88] include generalization and subset hierarchies and constraints (such as assertions of ranges on attribute values). Subset hierarchies are a set of entities where the general entity can also be an instance of the related entity. These are represented as hollow arrows with the head pointing to the general entity. A generalization hierarchy (ISA-relationship) is one where an entity is segmented depending on the value of one of its attributes. This is represented as hollow arrows for each segmented part pointing to the segmented entity.

For the university example, an EER diagram is depicted in Figure 2.3.

Person, **Employee** and **Course** are examples of entity sets. **Name** and **Age** are attributes of **Person**. **Name** is a key of **Person**.

Employee has the role name "**Manages**" and "**ManagedBy**" in the **Employment** relationship.

Dependant is a weak entity determined by the relationship between it and **Employee**.

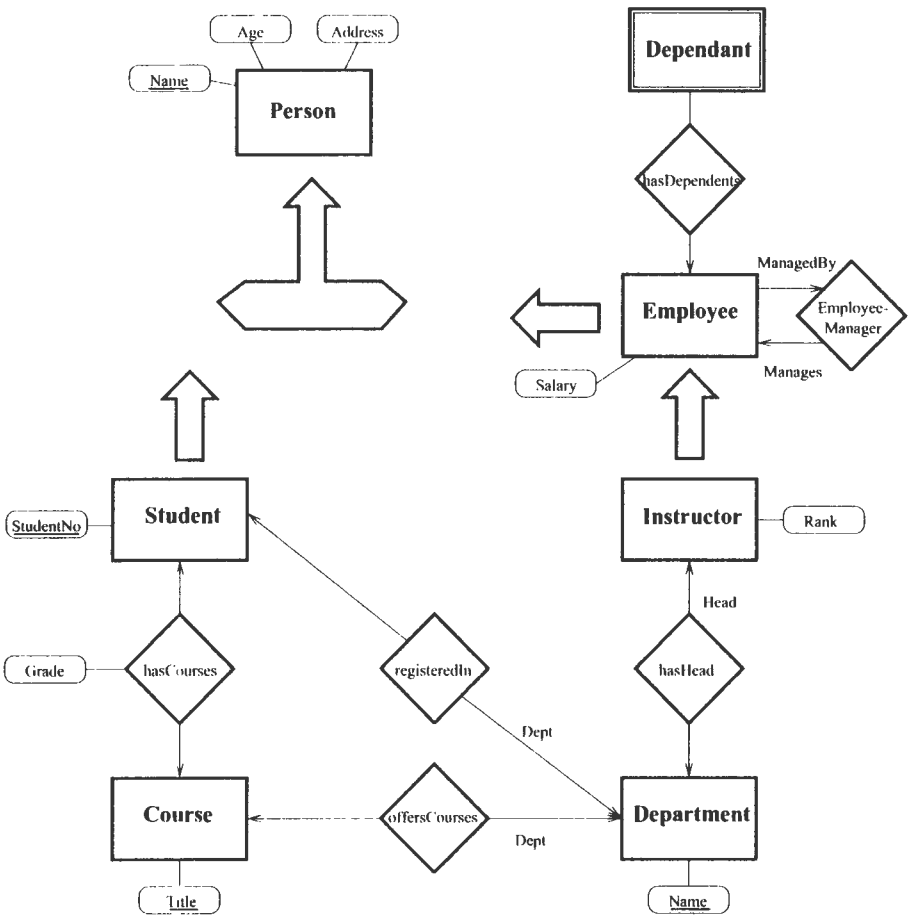


Figure 2.3: Entity Relationship version of University Database

The Semantic Data Model

The Semantic Data Model [HM81] provides a varied and comprehensive range of modelling constructs including aggregation, association, generalization, classification and derivation.

The main construct is the *class*, a collection of database entities of the same kind. Generalization and groupings of classes are called *interclass connections*. Connections and attributes are encapsulated in the class structure.

Attributes can be specified as "member" or "class" attributes. Member attributes are properties of each instance of a class, while class attributes apply to the class in general. An example of this would be: a `Person` class, where `Name` is a member attribute and `NumberOfPersons` is a class attribute.

Classes are either base or nonbase. Base classes are not defined in terms of other classes, and have a unique identifier, similar to a key in the ER model.

Interclass connections can either be subclass or group connections. Subclass connec-

tions allow generalization networks to be created. There are three forms of grouping construct that have been identified in this model. These are *expression-defined*, *enumerated* or *user-controllable* grouping. Membership of a class of the former grouping is based on a class's attributes satisfying some expression. The latter grouping allows users to arbitrarily define their own grouping. Members of an enumerated grouping class are explicitly listed.

Member attributes of a class can be derived from other values in a number of ways, including ordering (the value is the position of the member in a class), existence (the value is a boolean), arithmetic expressions, set operations (union, intersection or difference of other member attributes) and membership count. An attribute can be specifed as being the inverse of another existing attribute.

For the university example, the SDM representation is shown in Figure 2.4.

Subclasses are specified through interclass connections; for example, **EMPLOYEES** are subclasses of **PERSONS**. The **where specified** clause is used because there is no attribute of **PERSON** (such as **Occupation**) that could specify the subclass relationship.

Derivations are illustrated in the **COURSES** declaration: **Results** is students, ordered by **Mark**, and **NumCourses** is the number of **COURSE** objects in the class.

The SDM provides a large number of modelling constructs, and uses classes to encapsulate relationships between objects. The SDM, however, does not provide an explicit aggregation operator: aggregation is only available at the attribute-level.

The Functional Data Model

The functional data model (FDM) and its specification language DAPLEX have *functions* and *entities* as constructs for representing data. Entities represent real-world objects. Standard types like strings are also represented as entities.

The form of the function stipulates the concept being modeled. Entities are defined as functions with no inputs and the key word **ENTITY** as range. An entity definition of Person would be "**Person() -> ENTITY**". Attributes are functions that have a domain of entities and a range of some standard type. Functions with multiple arguments represent aggregate relationships between many entities. Generalization is defined via a function of entities onto entities. For example the definition of Student: "**Student() -> Person**" can be read as "Student is a Person".

The FDM allows cardinality constraints to be specified between entities. The cardinality of relationships is specified as to-one or to-many only. A student takes many courses is defined as "**Takes(Student) ->> Course**". Properties can be derived from other values in the database schema in a number of ways.

Figure 2.5 is an FDM representation of the university database.

```
PERSONS

  member attributes:
          Name          value class: STRINGS
          Age           value class: INTEGERS
          IDNo          value class: STRINGS
  class attributes:
          Number        value class: INTEGERS
  identifiers:
          Name
          IDNo


COURSES

  member attributes:
          Title         value class: STRINGS
          Dept          value class: DEPARTMENTS
                        inverse: Courses
          Teacher       value class: EMPLOYEE
          Students      value class: STUDENTS
                        inverse: Courses multivalued
  class attributes:
          NumCourse     value class: INTEGERS
                        derivation: number of members in class
          Results       value class: STUDENTS
                        derivation: order by Mark
  identifiers:
          Title


FACULTIES

  member attributes:
          Title         value class: STRINGS
          Dean          value class: STRINGS
          Departments value class: DEPARTMENTS
  identifiers:
          Title


DEPARTMENTS

  member attributes:
          Title         value class: STRINGS
          Head          value class: HEAD
          Courses       value class: COURSES
                        inverse Dept multvalued
          Employees     value class: EMPLOYEES
                        inverse: Dept multivalued
  identifiers:
          Title


EMPLOYEES

  interclass connection: subclass of PERSON where specified
  member attributes:
          EmployeeNo    value class: INTEGERS
          Dept          value class: DEPARTMENTS
                        inverse: Employees
          Rank          value class: STRINGS
          Salary        value class: REALS


HEADS

  description: Heads of departments
  interclass connection: subclass of EMPLOYEES where Rank="Head"
  member attributes:
          Secretary     value class: EMPLOYEES


STUDENTS

  subclass of PERSONS where specified
  member attributes:
          StudentNo     value class: STRINGS
          Faculty       value class: FACULTIES
          Courses       value class: COURSES
                        inverse: Students multivalued
          Mark          value class: STRINGS
```


Figure 2.4: SDM version of the University Database

The functional data model does not provide as many constructs as other models, but provides a convenient functional representation of relationships. This gives greater expressive power by providing simpler constructs. It can be argued [PM88] that not providing aggregation or grouping constructs, but rather using direct relationships, creates less complicated schemas.

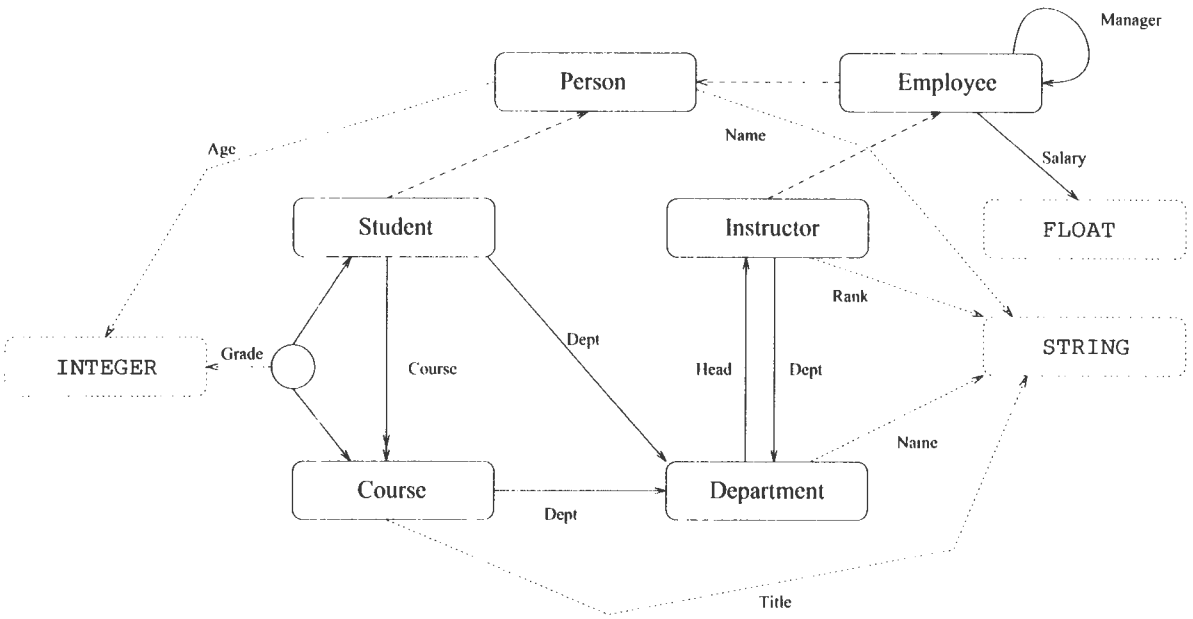


Figure 2.5: Functional Data Model version of University Database

The DAPLEX representation of the Figure 2.5 is given below:

```
declare Person() ->> ENTITY
declare Department() ->> ENTITY
declare Course() ->> ENTITY
declare Employee() ->> ENTITY
declare Student() ->> Person
declare Instructor() ->> Employee

declare Name(Person) -> STRING

declare Dept(Student) -> Department
declare Course(Student) ->> Course

declare Title(Course) -> STRING
declare Dept(Course) -> Department
declare Instructor(Course) -> Instructor
```

```
declare Salary(Employee) -> INT
declare Manager(Employee) -> Employee

declare Rank(Instructor) -> STRING
declare Dept(Instructor) -> Department

declare Name(Department) -> STRING
declare Head(Department) -> Instructor

Grade(Student, Course(Student)) -> INT
```

The IFO model

The IFO model [AH87] is a formal graph-theoretical model designed to investigate the structural aspects of semantic data models.

The IFO model comprises three constructs: objects, fragments and ISA relationships.

Objects can be one of three types: printable, abstract and free. Printable objects, represented by rectangles, are the standard system types like "integer". Abstract objects, equivalent to entity sets in the ER model, are represented as diamonds. Free objects are the underlying entities inherited from ISA relationships. Free objects are represented as circles.

Polyatomic types can be constructed using two mechanisms: grouping and aggregation. The "star-vertex" operator ⊛ is equivalent to grouping in the Semantic Data Model. For example, ⊛`[STUDENTS]` is a set of students. The "cross-vertex" operator ⊗ is the cartesian product operator used for aggregation of objects. An example is the aggregation of instructor and class to form a lecture: `LECTURE = ⊗[INSTRUCTOR,CLASS]`.

Fragments are equivalent to functions in the FDM, except that fragments can be defined over a set of objects. This is a more general type of nesting than that provided by the FDM. Fragments are represented as labelled arcs in an IFO diagram.

ISA relationships are categorized into specialization and generalization relationships. Generalization appears as thick arrows and specialization relationships are hollow arrows. In Figure 2.6, `Instructor` and `Student` are specializations of `Person`.

Figure 2.6 shows an IFO diagram of the university database.

Other models

- Object-oriented models

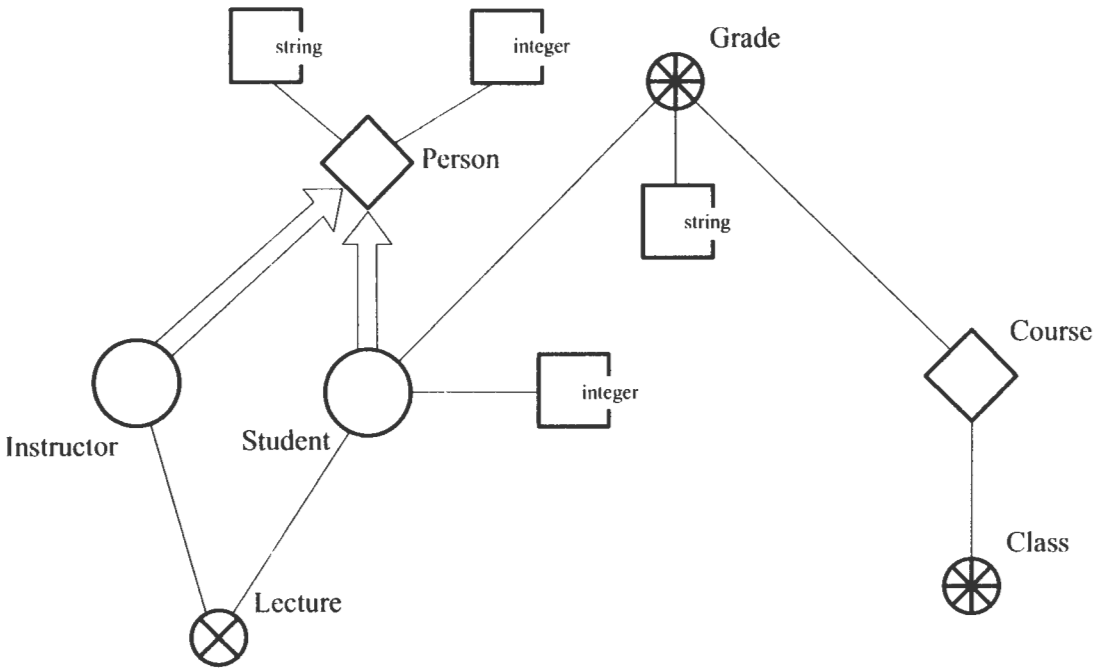


Figure 2.6: IFO version of University Database

Object-oriented models encapsulate data and the operations (methods) on the data within a class. Hierarchies of classes allow inheritance of properties (data and methods).

Object-oriented models are different to semantic models since they encapsulate behaviour at the local level: methods describe the interactions with other classes. Semantic models, however, attempt to provide simpler interactions between data at a higher level by providing general constructs that specify standard relationships and ensure consistency.

- Binary models

  Binary models use two constructs to represent data: *entity sets* and *binary relations*. An example of a binary model is the Semantic Binary Data Model (SBDM) [Abr74]. The schema representation consists of nodes for the entity sets and labeled edges for the binary relationships. Extensions to the SBDM include cardinality constraints on relationships and ISA relationships [Ris85, Ris86].

  Binary models are equivalent to other semantic data models in representing data. Some constructs, such as aggregation, need to be simulated to produce the same effect – for example, the aggregation of **`Person`** as an **`Address`**, **`Name`** and **`Age`** is treated as three relationships.

2.3.4 Comparison

Table 2.1 identifies the main modelling constructs in semantic data models, and compares how the prominent models implement each.

Attribute aggregation refers to attributes being associated with a single object, rather than a set of objects being associated with a object.

Generalization is implicit in the FDM since these subtypes are not defined as having an ISA relationship with a supertype. The SDM and IFO models allow generalization networks where an object can inherit properties from more than one object.

	EER	**FDM**	**SDM**	**IFO**
aggregation	attributes	attributes	explicit	explicit
generalization	explicit hierarchy	implicit hierarchy	explicit network	explicit network
derivation	none	varied	varied	varied
association	explicit	implicit	implicit	explicit
grouping	none	none	exists	exists

Table 2.1: Comparison between prominent data models

2.3.5 Conclusion

Prominent data models have been described and compared by illustrating the conceptual modelling tools of each model. All data models are considered semantically equivalent in most aspects [PM88]. The differences lie in the treatment of the different conceptual modelling tools. Models like SDM provide complex abstractions to cover most modelling situations while others like the binary and functional data models provide fewer, simpler constructs that are combined to provide more complex relationships.

Chapter 3

A Prototype Systems Development Environment

This chapter introduces the prototype Systems Design Environment (SDE) which was the forerunner of the Design Workbench. The Design Workbench retained the overall organisation and functionality of this prototype (DEN [FIJK94]), and extended its capabilities in several ways. This chapter therefore describes only the structure and functionality of DEN; subsequent chapters will present the extensions, outline implementation issues and highlight different approaches used in the Design Workbench.

The main components of the system are the Data Modeller (DMC) and Process Modeller (PMC), the Data Repository (DR) and the User Interface (UI). Figure 3.1 gives an overview of the system. In the prototype, the DMC accepted only entity-relationship (ER) models and the PMC used the structure chart approach.

The Data Repository resides on the persistent store and contains all procedures, types, models, programs, specifications, etc. designed by the user or produced by DEN.

The Data Modeller accepts and checks designs, and automatically generates Napier88 types and bulk data objects based on this. Standard procedures for manipulating these types (*handlers*) can also be generated, eg to input or display an instance. Users input both process and data models from a file; textual input is preferred over graphical design because it is faster [Mar88]. Process models are designed, and skeleton code generated and organised in a hierarchy on the persistent store by the Process Modeller. The UI includes a graphical display feature, and a query facility which allows users to retrieve information on aspects of the developing system.

This chapter presents each of the four components of DEN in turn: Data Repository, Data Modelling, Process Modelling and User Interface components.

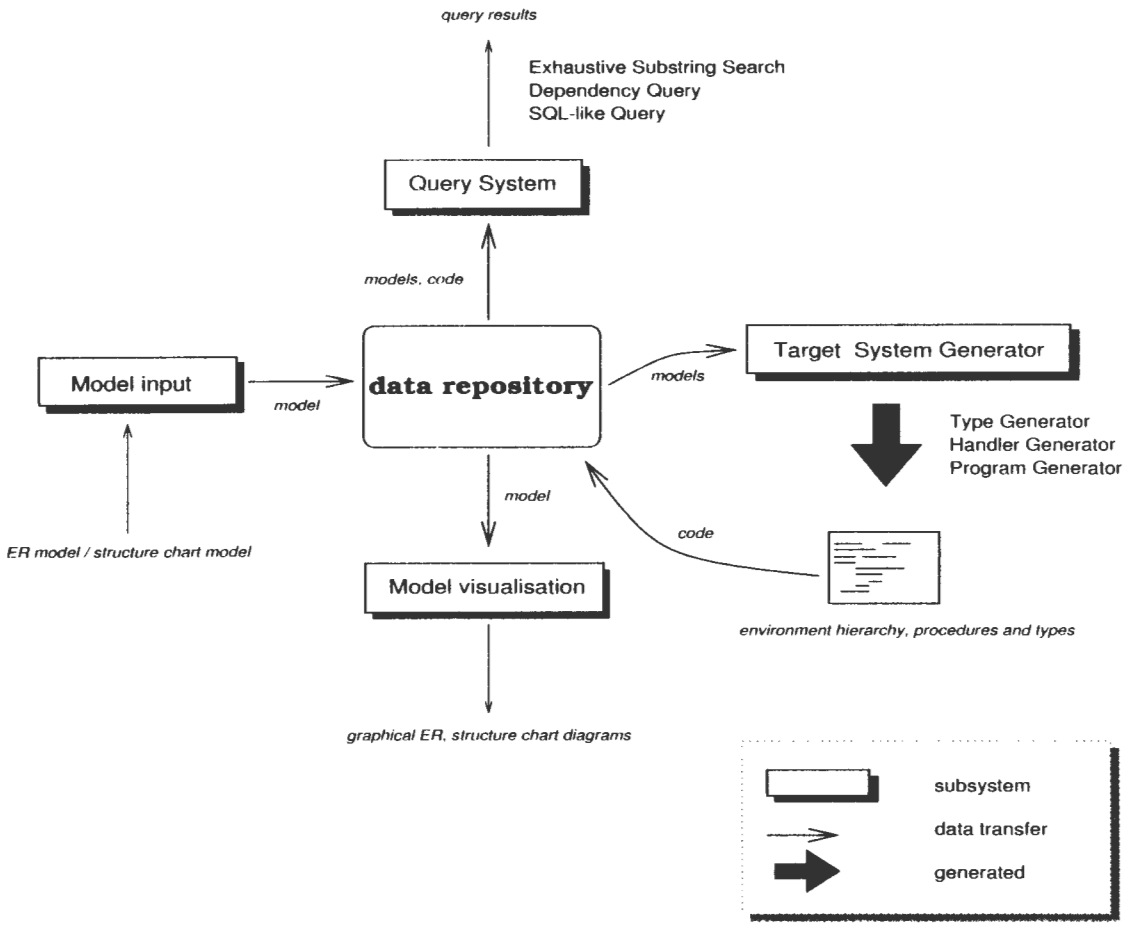


Figure 3.1: A system overview

3.1 The Data Repository

The Data Repository serves as the central store of all software development information, including documentation, design models (data and process) and target system (procedures, types and bulk variables). Figure 3.2 is a representation of the Data Repository data structures. All components have access to the repository, so that the information produced by one component is available to another component. This shared data provides a framework necessary for an integrated environment where individual tools (in the different components) have a common view of the data.

The prototype SDE creates a separate environment on the persistent store for each target system. The Data Repository in each target environment is made up of Maps for the main objects in the system design, to support the five main parts of the design process: data model design, process model design, program code generation, generation

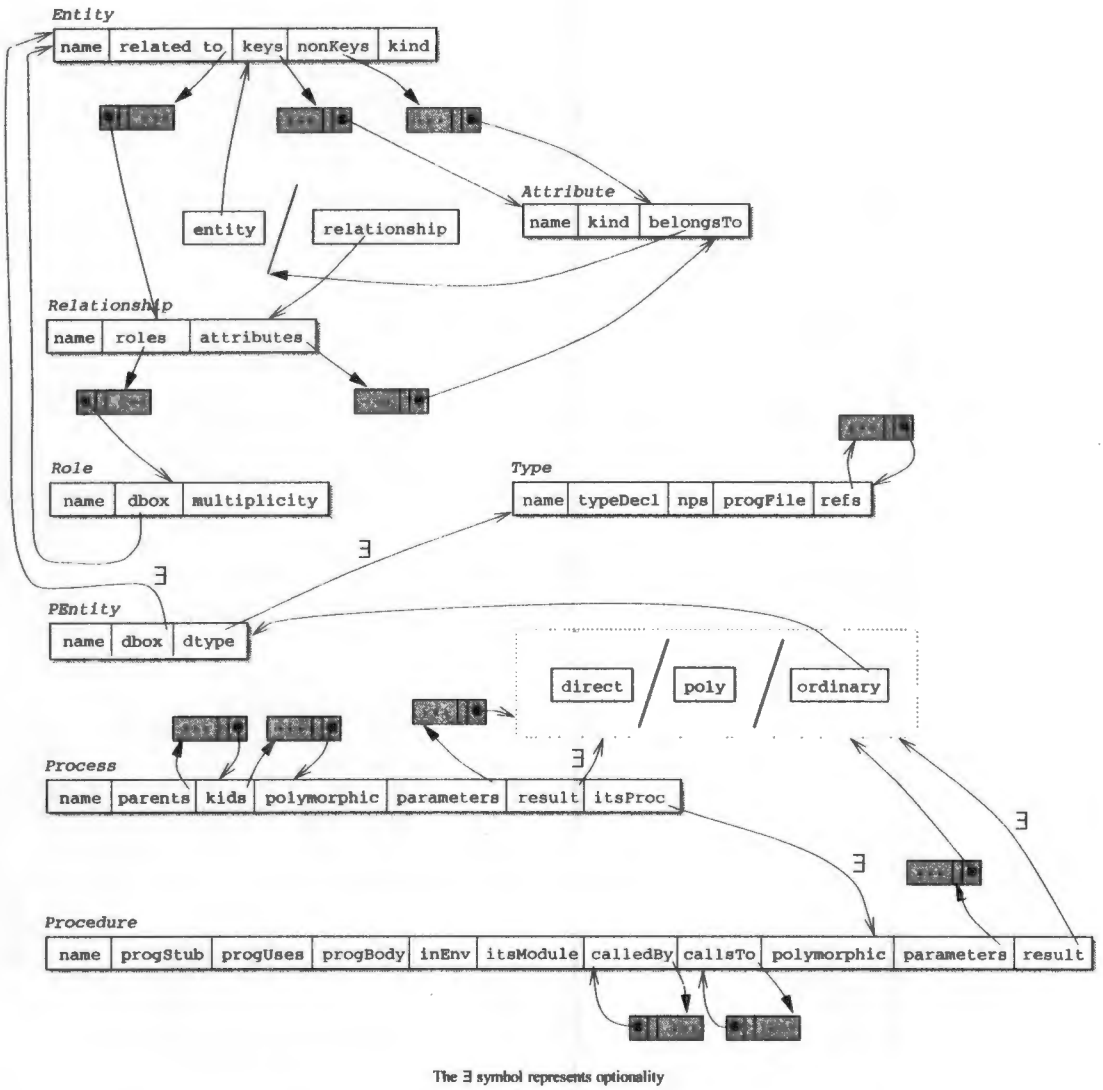


Figure 3.2: The major DR data structures

of types and generation of bulk variables.

The main data model objects are **DataModel**, **Entity**, **Relationship**, **Attribute**, **Role** and **Type**.

Process model objects reference each other to reflect the system decomposition. The result and parameters of the process object can be system types (such as strings) or entity sets (**Entity**). The code of the design object is also referenced through the **Process** structure. The procedure's position in the store, its parameters and result are also stored. The code is stored in a structure called **Procedure** which holds the different parts of the user code including the headers (called the *preamble*, a listing of where program items exist on the store). The generated types for each data model object refered to in the **Process**'s parameters and result are also referenced from the

`Procedure` structure.

The Napier88 types generated from the data model are stored in structures that hold type declarations. Each Napier88 variable generated is stored in a structure called `Variable` which has the declaration code of that bulk variable, as well as references to the data model design objects that were used to generate the object.

Each structure has a *tag* field used in change impact analysis to avoid cycles when traversing the DR.

3.2 Data Model Component

Data models are input in a model specification language, since textual input is faster than graphical input. This text is parsed, and data model objects (eg. `Entitys` and `Relationships`) are inserted into the Data Repository for that particular target system.

Napier88 types can be automatically generated from the data model objects. Bulk variables of these types can also be generated.

From the types stored in the Data Repository, handler procedures on these types can be generated and placed on the store. To illustrate how this can be done, three such procedures are currently generated for any type. These will input, print and initialise an instance of the type respectively.

The first subsection below describes how data models are parsed and stored in the DR. Then section 3.2.2 describes how Napier88 types are generated from stored models and the section concludes with a description of the Handler Generator subsystem.

3.2.1 Model Input

The ER input language was designed to be concise but clear. The language needed to capture the main elements of an ER diagram: (weak or strong) entity sets, attributes, keys, roles, multiplicity and relationships.

Figure 3.3 shows an ER diagram for a parts-supplier model and Figure 3.4 gives the corresponding textual input specification.

The language to specify a data model is shown below, with the steps involved once the line is parsed. Tokens are distinguished by enclosing them in quotes, eg. "[".

- *entity* "--" [["["] *attribute* ["]"]]*

  This specifies a new entity set and/or its new attributes.

  If the entity set exists, the attributes are added to the entity set. Duplicate attribute names are ignored. Keys are specified by enclosing the attribute(s)

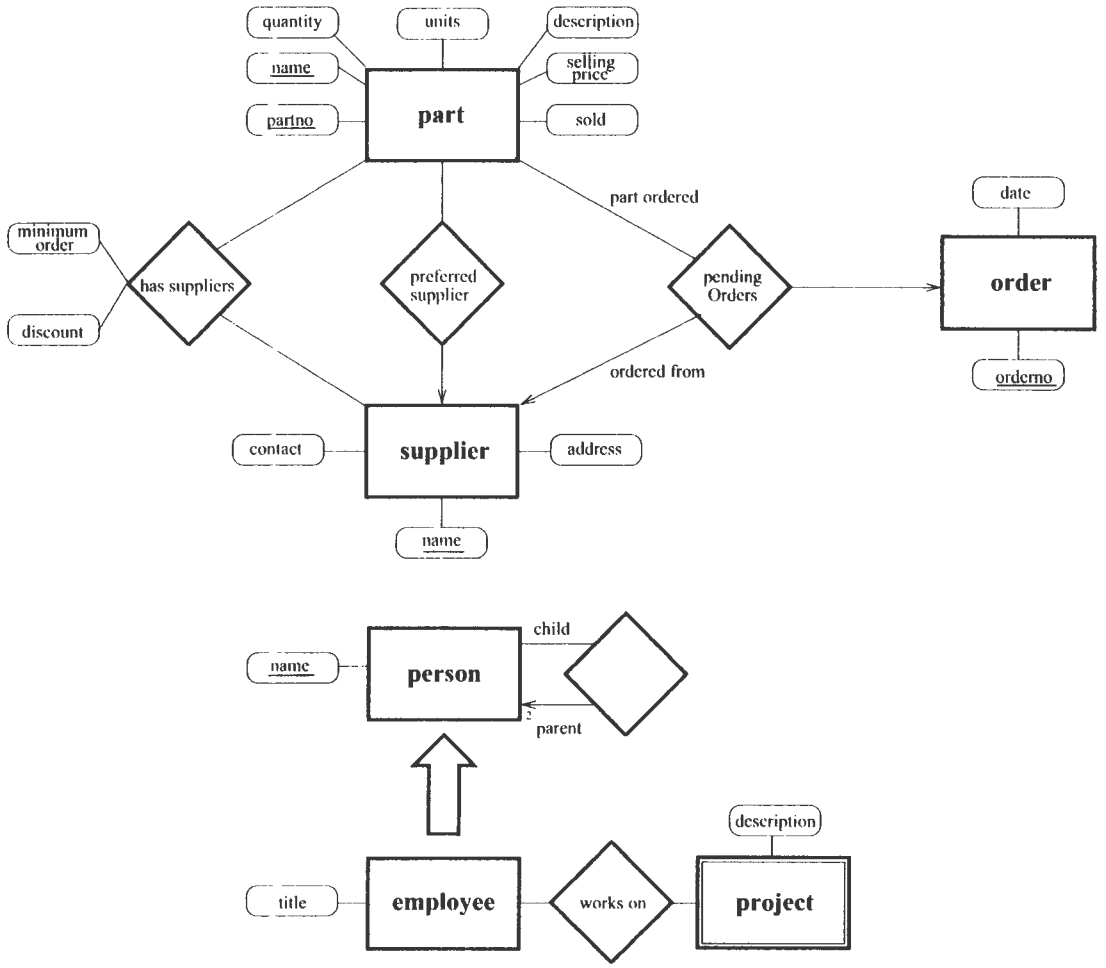


Figure 3.3: ER diagram of a parts-supplier model

in square brackets. Multiple simple or composite keys can be catered for; in Figure3.4, **name** and **partno** are multiple simple keys. If these attributes were combined to form a composite key, it would be specified as "**[partno name]**".

- *entity* ["<"|"<" number]"<" *entitySpec* [*entitySpec* | *attribute*]* ["(" *name* ")"]

  where *entitySpec* is "->" [*number* ">" | ">"] *entity* ["(" *rolename* ")"]

  This specifies a relationship by giving the entity sets and possibly also the attributes of the relationship.

  N-ary relationships can be specified between entity sets: each identifier on the right of the arrows is checked to see if it exists as an entity set. If not, the identifier is an attribute of the relationship. Role names can be specified by enclosing the role name in round brackets after the name of the entity set. Unspecified many-valued relationships are given as double arrowheads, while relationships

```
ER parts-supplier

part -- [partno] [name] quantity units description
        selling_price sold
supplier -- contact [name] address
order -- date [orderno]
person -- [name]
employee -- title
project -- description

part <<-> supplier "preferred_supplier"
part <<-> supplier order "pendingOrders"
part (part_ordered) <<->> supplier (ordered_from)
                          minimum_order discount "has_suppliers"

person (child) <<->2> person (parent)
employee ISA person
employee <=>> project "works_on"

int: sold quantity units minimum_order
real: selling_price discount
```

Figure 3.4: ER input specification of a parts-supplier model

with multiplicity n (having exactly n associated values) are specified as **<*n*<** or **>*n*>**.

In Figure 3.3, **person** is in a relationship with **person** with a multiplicity of 2 for **parent** and many-valued for **child**. This is specified as "**person (child) <<->2> person (parent)**".

A relationship may be named by specifying its name in quotes at the end of the line; otherwise, the relationship is given a default name (the concatenation of the names of its entity sets). If more than one relationship exists between two or more entity sets (resulting in the same default name), the default name is the concatenation of the names of its entity sets and a unique number.

The first relationship in the parts-supplier input specification is between **part** and **supplier** and is called **preferred_supplier**. If this relationship was not named, the default name would be **part_supplier**, and if the other relationship between **part** and **supplier** (named **has_suppliers**) was not named either, it would be called **part_supplier2**.

- *entity* "**<=>>**" *entity* ["**"** *name* "**"**]

This specifies a weak entity set.

The strong entity set is on the left, the weak is on the right. If both entity sets exist in the data model, this relationship is associated with them. The weak entity set's *kind* field is set to "weak", and references this relationship. The **works_on** relationship defines the weak entity set **project** associated with an **employee**.

- *simple-type* ":" *attribute* [*attribute*]*

  This defines attribute types.

  The attributes are searched for in the current data model, and the *kind* field is set to <*simple-type*>, which is one of int, bool, string or real. In the example, **selling_price** and **discount** are of type real.

The first line of the textual input specifies the name of the model. If this data model does not exist, a new model is initialised.

3.2.2 Type Generation

The Type Generator will automatically generate type declarations from a given data model on request. Each data model **Entity** is converted to a Napier88 type. For an ER data model, each entity set becomes a structure and its attributes are converted to components of the resulting structure:

<*attribute-name*> : <*attribute-type*>

The basic type declaration is therefore of the form:

```
type <entity-name> is
        structure (
                    (∀i)(<attribute_i-name> : <attribute_i-type>) ;
                    extra:any
                  )
```

An extension field **extra** is appended to the type definition, so that changes to the type can be absorbed without changing the type definition. We consider it good practice to build such fields into all structures and thus encourage its use in target systems.

In addition to a type declaration, a variable to represent the extent of a type (the entity set occurences) is also generated, and stored in the Repository and in the target system. This takes the form of a Map, with the key attribute as domain and the type of that entity set as range.

Two strategies for representing relationships were identified: the **Relational** and the **Reference** approaches. A system designer can choose between these two approaches

on a target-system-dependent basis. Three examples are used to illustrate the two approaches. Their ER input specification is given below (using the ER input language in section 3.2.1):-

```
part (parts) <<->> supplier (suppliers)                    (A)
person (parent) <2<->> person (child)                      (B)
part <-> supplier (preferred_supplier)                     (C)
```

- **Relational approach**

  For each entity set, a Map is generated whenever the entity set is involved in relationship with another entity set. The Map's domain is that of the key of the entity set and its range is a list, a vector or a single reference (whose type is the type generated for the related entity set). A list is generated for to-many relationships, a vector for "to N" associations, and a single reference for to-one relationships.

  For example (A) above, a many-to-many relationship, the type declarations generated would be
  `type Part is structure( partno:int; ··· )` and
  `type Supplier is structure( name:string; ··· )`.

  The Maps generated will be `Parts:Map[string,List[Part]]` and `Suppliers:Map[int,List[Supplier]]`. The generated code that creates these variables is:

```
let lessthan_int := proc(a,b:int); a < b
let lessthan_string := proc(a,b:string); a < b
let equality_int := proc(a,b:int); a = b
let equality_string := proc(a,b:string); a = b

in TargetEnv let
  Suppliers := m_empty[int,List[Supplier]]( equality_int, lessthan_int )
in TargetEnv let
  Parts := m_empty[string,List[Part]]( equality_string, lessthan_string )
```

  The types of the Maps for example (B) are
  `Parent:Map[string,*Person]` and `Child:Map[string,List[Person]]`,
  and for (C),
  `Preferred_Supplier:Map[int,Supplier]` and
  `inverse_Preferred_Supplier:Map[string,Part]`.

  These **bulk variable** declarations are stored as `Variables` in the Data Repository. A Map's declaration consists of two procedures ("equal" and "less than") that aid the storage of the Map internally. These initialisation procedures are

generated, and depend on the domain of the Map. Where role names are omitted, the name of the Map is the concatenation of the names of the entity sets making up the relationship, and has the form `m<entity`$_{from}$`>_<entity`$_{to}$`>` or `inverse_m<entity`$_{from}$`>_<entity`$_{to}$`>` .

- **Reference approach**

  The reference approach embeds entity set components within other entity sets for all relationships. As with the Relational approach, the reference to the related object is a list, a vector or a single reference, depending on whether the relationship is multi- or single-valued. Since relationships can lead to mutually recursive types, recursive type declarations using variants with null alternatives are employed in the usual Napier88 way. This method is also used for Vectors for the same reason. This variant type is called `Optional` in the following examples.

  The type declarations for example (A) are:

```
rec type Part is structure( partno:int; ··· suppliers:List[Supplier] )
& type Supplier is structure( name:string; ··· parts:List[Part] )
```

  Example (B)'s generated type is:

```
rec type Person is structure( ··· parent:Optional[*Person];
                              child:List[Person] )
```

  The type declarations for example (C) are:

```
rec type Part is structure( partno:int; ···
                            preferred_supplier:Optional[Supplier] )
& type Supplier is structure( name:string; ··· part:Optional[Part] )
```

Finally, we note that many-to-many relationships that have attributes are distinguished by the type generator: a structure type having these attributes as fields is created for each such relationship, in the same way as is done for entity sets. For one-to-many / many-to-one relationships with attributes, these are incorporated as fields in the "to-many" entity structure; for one-to-one relationships, their attributes become fields of just one of the participating entities, to avoid redundancy.

Difference between the Reference and Relational approaches

Programming language structures typically represent relationships between objects using what we call a Reference Approach – references (or pointers) to related entities are nested directly within objects, with these references collected into Lists (or Vectors) if the relationship is to-many (or to-N) respectively.

In contrast, a relational database system represents "to-many" associations using a relation where entities are identified by their primary keys. A specific entity appears in N tuples of that relation if it is associated with N objects. The advantages of the relational system are:

- Relationships are represented using the same mechanism as entities, making the model simple to understand and use.
- In an M-ary relationship, there will be M attributes giving the identifiers of the M participating entity sets, and indexes can be created for any of these. Each index gives rapid access to information, given the primary key or identifier of an entity participating in that association. Thus, although a relation may in fact represent several indexes, it is seen as a single construct which can be used in a uniform manner to obtain information given any of these M key values. This makes such relations very flexible and easy to use.

The Reference approach has obvious advantages in a persistent programming language like Napier88 - embedding these references directly within objects makes it fast and easy to access related entities. This is thus the default approach adopted in type generation.

However, DEN includes the option of a relational approach to representing associations, in order to capture some of the advantages of relational systems mentioned above. The system user (typically the leader of the design team) can select the Reference or Relational approaches *or both*, when requesting types to be generated.

If the Relational approach is selected, the system gains the advantage of supporting fast access given the key of any object participating in a relationship. This is achieved by creating a Map from that key onto the related object (or onto a List of these). Unfortunately the uniformity and simplicity of the relational model is not retained, for two reasons. Firstly, a Map distinguishes between domain and range objects, and is purely an index on domain values. It is therefore not possible to have eg. `Deliveries:Map[Part,Supplier]` which can be used to access information via either Parts or Suppliers - a separate Map is required for the inverse association. Secondly, domain values have to be unique, so it is not possible to have a `Map[key`$_A$`,Z]` if an `A` entity can be associated with many `Z` objects; instead a `Map[key`$_A$`, List[Z]]` is required. This means that in order to process information using this structure, a mixture of Map and List manipulation is required.

Nevertheless, by providing an optional relational approach, it is possible to create relationship Maps of the form `Map[key`$_A$`,Z]` or `Map[key`$_A$`, List[Z]]` for rapid access to `Z` objects given the key of an `A` occurrence. Without this, it would be necessary to first access the relevant `A` object via its "extent" Map `Map[key`$_A$`,A]`, and only then reach the `Z`(s) associated with it. Operations on a relationship set can also be more efficiently done by scanning its relationship Map rather than going via an extent Map.

If both Reference and Relational approach are selected, the List of objects associated with a specific entity is not duplicated, but shared, to save space/maintenance costs. That is, if A→Z is a to-many relationship and both approaches are requested, then the entry for a specific `A` object in the `Map[ key`$_A$`, List[Z]]` will be the same list as is referenced by the `List[Z]` field within the `A` object.

3.2.3 Type Handlers

Handler procedures can be generated for each entity set in the data model. The three types of handlers in DEN are called **read**, **write** and **make** handlers that input, output and initialise instances of an entity set respectively.

Each attribute *attribute* of simple type *type* that is an attribute of a particular entity set or many-to-many relationship set in the Data Repository is converted to the corresponding code:

- read the attribute: `instance_name(`*attribute*`) := read`*type*`()`
- write the attribute: `write`*type*` ( instance_name (`*attribute*`) )`
- initialise the attribute: `defaultValue(`*type*`)`

where `defaultValue` is the default value of a type. For example, in the case of integers, the default value is 0. This method works for strings, integers, reals, nulls, any's and booleans. User-specified types for attributes could be supported in a similar fashion, but this was not implemented.

For each relationship an entity set is involved in, code is generated to find the related entities in the extent (Map) variables generated by the Type Generator. This code updates the Map generated for that association (using the Relational approach) and/or updates the corresponding field (if the Reference approach was used).

An example of a read handler for a Part-Supplier data model is `readPart`, shown in Figure 3.5. This figure shows the generated handler to input an instance of the type Part which was constructed using the Relational approach. Figure 3.6 is the read handler that would be generated if Part was made using the Reference approach. In the data model, parts have many suppliers.

In the initial Handler Generator, relationships were treated differently by the generated procedures. Instead of locating related items in the Map for that entity set using their key, the handler procedure for that entity was invoked. Thus, inputting a Part was accomplished by reading its simple attributes and then calling the read procedure for all related entity sets, eg. Suppliers. This led to problems with cyclic references, and had to be replaced by the current approach which ensures that either the referenced entities are already present in the target system or they are given a null value to indicate that they are absent. This required changing the Type Generator to include a variant field for each related entity set. The alternative of allowing references to non-existent objects, and creating dummy entities to represent these, seemed much less secure; but the system could easily be changed to use this approach instead if preferred. The *initialise* handlers simply set the variant field to indicate that the related entity set is absent, and *write* handlers print the key values of referenced objects.

```
readPart:= proc( -> Part )
begin
 part:=makePart()
 writeString("PartNo:"); part(PartNo):= readInt()
 writeString("Stock:"); part(Stock):= readInt()
 writeString("SinglesPrice:"); part(SinglesPrice):= readReal()
 writeString("CasePrice:"); part(CasePrice):= readString()
 writeString("Name:"); part(Name):= readString()
 writeString("Units:"); part(Units):= readInt()

! part <<->> supplier
 writeString("Enter number of Supplier''s: "); let numSupplier:=readInt()
 writeString("Enter Name'n")
 let tmpSupplierName:=""
 let Supplier_list:=l_make[Supplier]()
 for i = 1 to numSupplier do
 begin
        writeInt(i); writeString(": ")
        tmpSupplierName:=readString()
        if ~m_contains[string,Supplier](mSupplier, tmpSupplierName) then
                writeString("Error: '"Supplier'" does not exist'n)
        else Supplier_list:=l_isu_append[Supplier]( Supplier_list,
                m_find[string,Supplier](mSupplier, tmpSupplierName) )
 end
 if i ~= 0 do
   m_isu_insert[int,List[Supplier]]( mPart_Supplier, part(PartNo),
      Supplier_list )

! part <<-> order
 writeString("Enter OrderNum of Order: "); let tmpOrderNum:=readInt()
 if ~m_contains[int,Order](mOrder,tmpOrderNum) then
        writeString("Error: no such Order exists'n")
 else m_isu_insert[int,Order]( mPart_Order, part(PartNo),
        m_find[int,Order](mOrder,tmpOrderNum) )
part
end
```

Figure 3.5: Example of a read handler: **readPart** using the relational approach

The Handler Generator was also changed in a more fundamental way when the prototype was replaced by the new system. DEN had used the generated types as a basis for creating handler procedures. This approach parsed each generated type declaration string to ascertain the names and types of fields. Unfortunately, this meant that as the Repository changed to handle additional data models and modelling constructs, the Handler Generator had to be altered as well – which was fairly cumbersome, involving much string processing and use of linguistic reflection. The approach was changed to work from the data model instead, because more information was then available (such as the multiplicity of an entity set in a relationship set). This meant that changes to the Repository were more easily implemented in the Handler Generator and ensured that the handlers became more versatile and were not limited by the semantics of the generated Napier88 types.

The handlers are stored in the process hierarchy (of the Repository) so that the process model can refer to these procedures. The process modelling component generates the stubs for these handlers and organises their position in the store's environment

```
readPart:= proc( -> Part )
begin
 part:=makePart()
 writeString("PartNo:"); part(PartNo):= readInt()
 writeString("Stock:"); part(Stock):= readInt()
 writeString("SinglesPrice:"); part(SinglesPrice):= readReal()
 writeString("CasePrice:"); part(CasePrice):= readString()
 writeString("Name:"); part(Name):= readString()
 writeString("Units:"); part(Units):= readInt()

! part <<->> supplier
 writeString("Enter number of Supplier''s: "); let numSupplier:=readInt()
 writeString("Enter Name'n")
 let tmpSupplierName:=""
 for i = 1 to numSupplier do
 begin
        writeInt(i); writeString(": ")
        tmpSupplierName:=readString()
        if ~m_contains[string,Supplier](mSupplier, tmpSupplierName) then
                writeString("Error: '"Supplier'" does not exist'n)
        else part(Supplier_list):=l_isu_append[Supplier]( part(Supplier_list),
                m_find[string,Supplier](mSupplier, tmpSupplierName) )
 end

! part <<-> order
 writeString("Enter OrderNum of Order: "); let tmpOrderNum:=readInt()
 if ~m_contains[int,Order](mOrder,tmpOrderNum) then
        writeString("Error: no such Order exists'n")
 else part(Order):=Exists[Order](exists:m_find[int,Order](mOrder,tmpOrderNum))
part
end
```

Figure 3.6: Example of a read handler: **readPart** using the reference approach

hierarchy. Once the handlers are generated, a user may edit them.

The Handler Generator can easily be extended to generate other types of handlers (such as GUI-based input and output). Different approaches within handlers can also be made available to the user, such as different forms of loops for inputting to-many relationships (the **readPart** example has a for-loop with a simple terminating condition). The more versatile the Handler Generator, the more useful the system becomes for rapidly obtaining prototype target systems.

3.3 Process Model Component

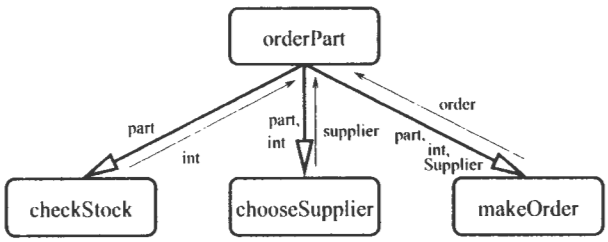
The PMC provides tools for the user to specify a process model, to manage the organisation of functions on the persistent store, and to automatically produce target system code. The system is generated according to the **incremental construction methodology** described in [Atk93].

This section describes how process models are input and stored in the Data Repository, and how procedures are generated from the stored models.

3.3.1 Structure Chart Specification Language

This was designed to make model input and edits as fast and simple as possible. Each input line begins with a command: **`create`**, **`link`**, **`break link`**, **`alter`** and **`remove`**. These initialise a new process, assign the children and parents of a process, remove children or parents of a process, alter the process description and delete a process respectively. The **`create`** and **`alter`** commands specify polymorphism, parameters and result types. The **`create`**, **`link`** and **`break link`** commands additionally have two commands to set/unset a process's parents (**`used by`**) and children (**`uses`**).

Figure 3.7 gives a diagram of a structure chart with its corresponding input specification. The dark lines show the calling hierarchy and are labelled with the types of the inputs to the called process. The return type of the process is labelled on the lighter lines.




```
create orderPart
uses checkStock, chooseSupplier, makeOrder
(part -> order)
end

alter checkStock
!compares quantity in stock with threshold
(part -> int)
end

alter chooseSupplier
!evaluates which Supplier to use (depends on quantity to order)
(part,int -> supplier)
end

alter makeOrder
!updates the map of orders of Part with quantity from Supplier
(part,int,supplier -> order)
end

end
```

Figure 3.7: Structure chart of **`orderPart`** with its corresponding input specification

Process models are parsed by the PMC and the information is stored in the Data Repository.

3.3.2 Code Generation

The system can automatically generate procedure stubs and program headers (preambles) for the target system and organise them on the store. The incremental system construction methodology described by Atkinson [Atk93] is employeed, which encourages modularity and software reuse, and facilitates system modification. It requires that stubs for all system procedures be placed in suitably constructed environments on the stable store before any coding is done. A stub comprises the procedure heading (interface) and an appropriate return value.

Environments and procedures are generated starting from a specified process in the process model. All descendants are dealt with in a postorder walk. An environment hierarchy for the target system is generated according to the structure chart: it mirrors the top of the process model hierarchy, and is up to three levels deep, depending on the size of the structure chart.

3.4 User Interface

The user interface component is divided into model visualisation and design querying.

3.4.1 Graphical Visualisation

Graph layout algorithms were used to visualise structure charts and entity-relationship diagrams. The ER diagram is drawn in a clear, structured fashion by classifying entity sets into levels according to the "to one" relations emanating from them [Mar83]. Each level is then drawn in the same plane. The structure chart algorithm recurses down the tree, drawing a process's descendants. Backward arcs are drawn to previously encountered processes.

3.4.2 Design Querying

A query language modelled on SQL was implemented. The query language interpreter accepted heavily restricted one-level (unnested) queries of the form:

SELECT *<attribute-list>*
FROM *<table-name>*
WHERE *<where-list>*

Users queried a view of the repository which essentially presented the DR Maps as "tables", with the fields of the Map elements being the attributes of that table. The

<where-list> was a conjunction of clauses of the form *attribute* = *value*, where *attribute* is a field of *<table-name>* in the DR. The data model name was also required if the table name was one of entities, relations or attributes:

WHERE data model=*<data-model-name>*

This is required because the DR can contain multiple (named) data models. The DEN Query System was extended in the Design Workbench to permit more complex retrievals.

Two other queries were implemented: an exhaustive substring search and a change management aid. The substring query searches for a given string within any field of any Data Repository object. The change management query retrieves all data and process model objects that could need to be changed, if a particular design aspect were altered, ie. those which are directly or indirectly related to the change.

3.5 Conclusion

A prototype System Development Environment in Napier88 called DEN was built to capture system designs comprising data (ER) and process (structure chart) models. Types, handlers and bulk variable instances could be generated from data models, and an environment hierarchy of program skeletons and stubs built from process models. Process and data models could be visualised using graphical views. Users could query the Data Repository with SQL-like queries, and dependency data, useful in change management, could be retrieved.

The DEN prototype simplified system design and implementation by providing integrated tools to aid software development. Experiments using this initial product demonstrated its utility as well as some drawbacks. System implementation time was shortened since types, handlers, bulk variables and program hierarchies were generated from the designs.

However, we felt that DEN might not readily be used in practice because it offered a limited number of conceptual models for a designer to work with. The next chapter deals with extending DEN to include other data and process models. The new system is called the Design Workbench.

Chapter 4

Supporting Multiple Co-existing Models

The prototype SDE described in Chapter 3 was replaced by a system that had the same functionality as the prototype but was generalised to handle multiple models. This chapter describes the extension of the prototype to support multiple co-existing models. The implemented SDE was then integrated into the Glasgow Workshop as the Design Workbench as described in the next chapter. Chapter 6 describes the extension of the Design Workbench to support metamodelling.

A single repository can be used to represent a variety of data- and process-models. If correspondences across models can be made known to the CASE system, such a repository architecture can permit alternative views of a design, and allow it to be viewed, edited and queried using different models at different times. This allows a group of designers to change between models to get different perspectives, and individual designers on a team can choose to work with the model they prefer. This chapter outlines the steps involved in order to cater for a representative sample of data models (based on surveys such as [HK87, PM88]) within one software engineering repository, and permit changing from one model to another while working on a single design. We then consider how new modelling constructs that may arise in future might be accommodated. Note that the work has focused on data modelling, with less emphasis on the process modelling aspect.

The first section of this chapter describes what additional data models were chosen to be integrated in the SDE. Section 2 outlines integrating FDM into a system developed for ER models. FDM includes constraints; the next section describes how these were incorporated. An outline of the subsequent integration of the OMT [RBP+91] model is given in section 4. The chapter concludes with a description of how other data models can be handled.

4.1 Choosing Additional Data Models

Two experiments were performed to investigate what is required in integrating other process and data models in the existing data repository. The experiments involved including the Functional Data Model (FDM) [Shi81] and OMT [RBP+91] and recording how each model was integrated. These specific models were selected for the following reasons. Firstly, we believe that functions provide an intuitive method for describing databases. The FDM was considered an appropriate data model to implement since it is simple (with minimal constructs) yet suitably rich and powerful, and one of its specification languages, Daplex [Shi81], has a convenient textual input format. Daplex's primary goal, according to Shipman [Shi81] is to provide a "conceptually natural" language as an interface to database design. The FDM includes classification, generalization, aggregation, derivation and constraints.

OMT was chosen because it extends both the data model (ER) and process model (structure chart) parts of the data repository. Furthermore, its popularity is growing rapidly and catering for OMT thus seemed a necessary step in providing a more general data repository.

4.2 Integration of a functional model

The initial version of the Design Workbench repository was based on that of DEN, extended to include ER features such as singular isa hierarchies. This section describes the integration of Daplex by first outlining what Daplex is, then explaining how Daplex is stored in the data repository and finally sketching how functional data models in general could be integrated.

4.2.1 Overview

The data repository of DEN (the prototype SDE) was initially designed to be as general as possible to facilitate implementing any underlying data model. Boxes (`Entity`) and lines (`Relationship`) were identified as the fundamental constructs in data models. Integrating the FDM required mapping it onto these constructs and adding constraints to the data repository.

Models can be queried and types and handler procedures can be automatically generated. Incorporating the FDM thus meant that changes to the Repository required modification of these subsystems as well.

4.2.2 Daplex

The specification language for the Functional Data Model is given below. An FDM tool was built to parse models specified in Daplex [Shi81] and store them in the data repository. Daplex declarations can be divided into the following categories:

- **DECLARE** *function-id*() -> **ENTITY**

  DESCRIPTION: A function with an empty domain onto the range {**ENTITY**}

  This is an entity definition, with the name of the entity being *function-id*.

- **DECLARE** *function-id*() -> *id*

  DESCRIPTION: A function with an empty domain onto the range of entities.

  Generalisation is specified in this way, where *function-id* is a specialisation of *id*. An entity called *function-id* is defined in the same way as the previous category, with an ISA relationship between the two entities (ie. *function-id* ISA *id*).

- **DECLARE** *function-id*($expr_1$, $expr_2$, ...$expr_n$) -> *id*

  DESCRIPTION: A function from entities, attributes or relationships onto entities.

  These are association-type functions are stored as a $(n+1)$-ary "one-way" relationship (ie. the inverse is not named here) with role name *function-id*.

- **DECLARE** *function-id*($expr_1$, $expr_2$, ...$expr_n$) -> **INT** | **BOOL** | **STRING** | **REAL**

  DESCRIPTION: A function where the range is a "printable" type.

  These functions specify attribute definitions. If the expressions $expr_1$, $expr_2$, ...$expr_n$ are identifiers, then a relationship between the identifiers is created having an attribute with the name *function-id* and the specified type (**INT**, **BOOL**, **STRING** or **REAL**) associated with it. If there is only one *expr*, an attribute of that entity is created. Expressions that are functions are implemented using aggregation.

- **DEFINE** *function-id*$_1$(*expr*$_1$) -> **INVERSE OF** *function-id*$_2$(*expr*$_2$)

  DESCRIPTION: A function defining the inverse of another function.

  Here, $expr_1$ and $expr_2$ must be entities. The functions *function-id*$_1$ and *function-id*$_2$ are the names of the attributes or relationships. If *function-id*$_2$ is an attribute name, $expr_1$ must be the printable type of the attribute. This is used to specify a unique identifier for the entity. For example, the definition "**DEFINE Student(STRING) -> INVERSE OF Name(Student)**" makes the attribute **Name** a unique identifier of **Student**. If *function-id*$_2$ is a relationship name, $expr_1$ must be the name of another entity participating in that relationship. This is used to specify the inverse relationship, for example, "**DEFINE Courses(Students) ->> INVERSE OF Students(Course)**".

- **DEFINE** *function-id*(*expr*) -> *expression*

  DESCRIPTION: Derived data is given.

  These functions specify derivations or virtual data. The result of the function *function-id* is that returned by evaluating the expression *expression*.

  The *expression* can include **TRANSITIVE OF**, **INTERSECTION OF**, **UNION OF** or **COMPOUND OF** operators.

- **DEFINE CONSTRAINT** *constraint*(id_1, id_2, ... id_n) -> *expression*

  DESCRIPTION: Defines a constraint on a number of entities.

  The constraint described by the boolean *expression* must always be true.

  For example, the following constraint aborts any updates to **Department** if the head of a department is not a member of the department:

```
DEFINE CONSTRAINT NativeHead(Department) ->
     Dept(Head(Department)) = Department
```

4.2.3 Integrating FDM in the repository

The initial data repository

The original repository data structures of the SDE (as shown in Figure 4.1) had evolved to include ER constructs such as attributes, keys and weak relationships.

The DR types for the entity-relationship constructs are **Entity**, **Relationship** and **Attribute**. A **DataModel** is the collection of entities, relationships and attributes. These are stored in separate Maps. Attributes are indexed independently since it is useful for attributes to be queried in their own right: this does not require going through the entity or relationship that the attribute belongs to.

Differences between the ER and FDM models

The functional data model was introduced using the underlying data repository designed for entity relationship models. Although there is only one modelling primitive, the function, the Daplex parser distinguishes the different categories, so functions are converted to the corresponding ER construct and stored in the data repository. The FDM does not support role names, and provides for attributes of relationships, as can be seen in the university example, where the "**Grade(Student,Course) -> INT**" function can be converted to the ER model by introducing a relationship with an integer attribute "**Grade**" (see Figure 4.2; the implied relationship is called "**Student-Course**"). This is accommodated in the Repository by introducing a relationship between the entities and associating the attribute with this.

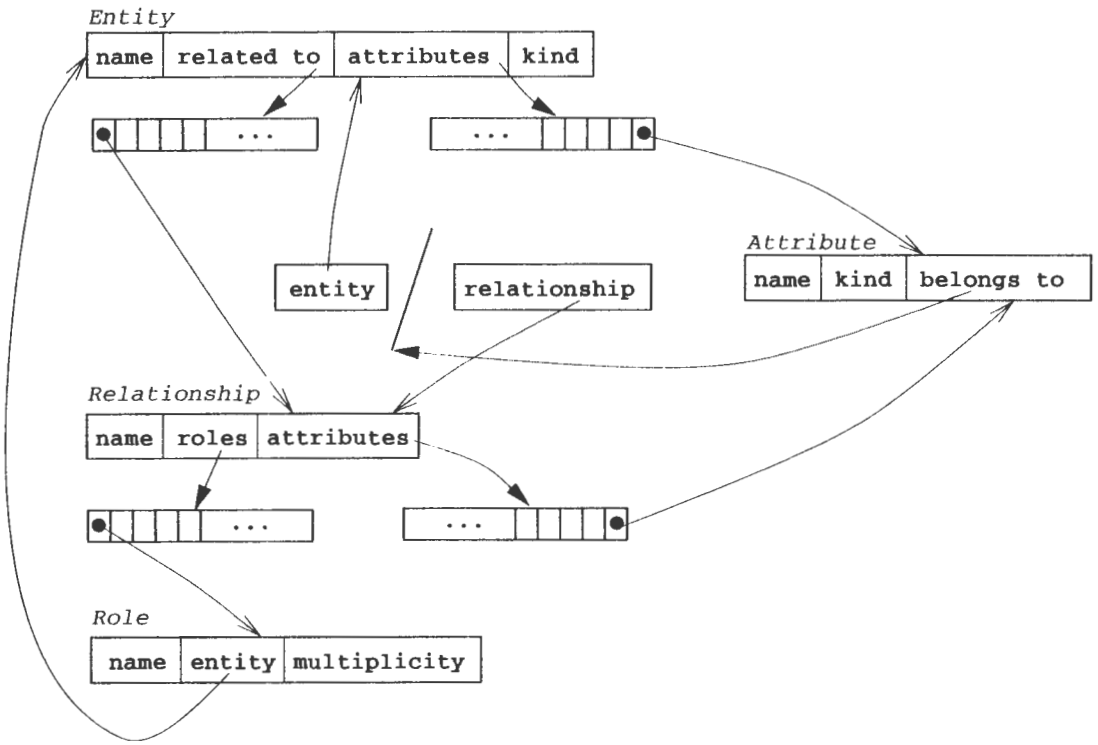


Figure 4.1: Original (ER-based) repository data structure

In contrast, the declaration "**Result(Student,courses(Student)) -> INT**" can be converted to the ER model by adding the attribute "**Result**" to an existing relationship "**courses(Student)**". This would have been declared as "**courses(Student) ->> Course**".

Alternatively, if an expression *expr* in "*function-id*(*expr*) **->** *id*" is a relationship and *id* is an entity, then the entity *id* is added to the relationship implied by *expr* and is given the role name *function-id*. For example, the declaration "**takes(offers(Department)) ->> Student**" augments the existing relationship between "**Course**" and "**Department**" declared by "**offers(Department) ->> Course**" with the entity "**Student**".

The DR does not require relationships to be "two-way" (ie. does not require inverses for all relationships): a relationship (**Relationship**) contains references to entities (*Entity*'s); if an entity does not reference a relationship it is involved in, this can be considered a one-way relationship. Figure 4.3 shows the implied relationship for the FDM function "**hasCourses(Student) ->> Course**". In this figure, the dotted lines show the representation if there were an inverse function for this example.

Procedures that manipulate the repository (such as those responsible for type generation) are unaffected when inverses are not specified since the unconnected entity ("**Course**" in this example) has no reference to the relationship.

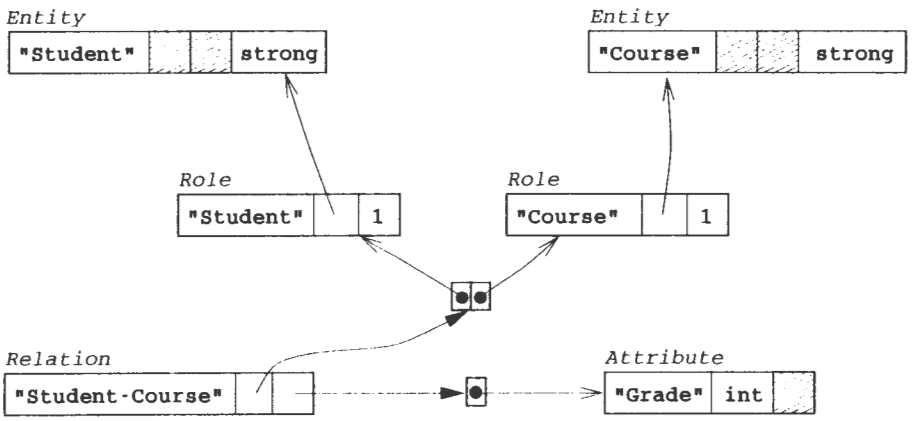


Figure 4.2: Data repository representation of **Grade**

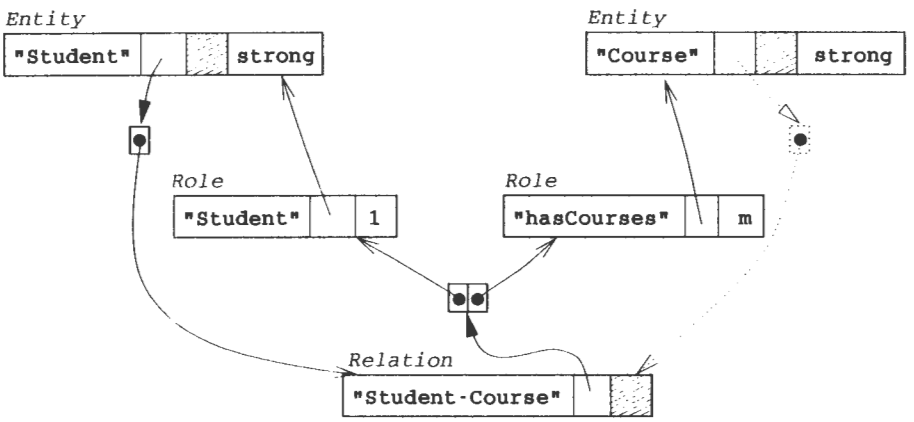


Figure 4.3: Data repository representation of **hasCourses**

4.2.4 Conclusion

FDM specifications can be input into the ER-based data repository. A subset of the FDM model was implemented: data manipulation statements (like **FOR**, as in "**FOR EACH Student SUCH THAT Grade(Student) > 50 PRINT Name(Student)**") were not relevant from a modelling viewpoint, and triggers were omitted because they can be handled in a similar way to constraints.

Although the FDM was not fully implemented, the subset that was handled required minor additions to the data repository and the procedures that accessed it, apart from the addition of constraints as described in the next section. The fact that a new model could be supported by the repository designed for ER modelling was very encouraging – it showed that sufficiently general and powerful data structures were being employed, and made the potential of catering for a wide variety of conceptual models seem highly likely.

4.3 Constraints in an SDE

4.3.1 Introduction

General integrity constraints are not supported by most versions of the ER model. However, in the FDM, constraints can be placed on entities. An SDE handling constraints needs to be able to represent them internally, to convert them to the programming language equivalent and to facilitate their enforcement in target system code (in particular, the handlers).

A constraint is a user-specified predicate on data that needs to be satisfied in order for the data to be valid. All insertions or updates that would violate the constraint must be aborted. An example of a constraint is the following, which requires that the head of a department is also a member of the department:

```
DEFINE CONSTRAINT NativeHead(Department) ->
  Dept(Head(Department)) = Department
```

This constraint is said to be "limited to" `Department` entities.

4.3.2 Constraint representation in the repository

A constraint is represented by an operator-operand tree for its body, and the list of entities it is limited to. The Napier88 type declaration is:

```
type Constraint is structure(
   name : string;
   limited_to : Optional[*Entity];
   body : BinaryTree[exprInfo] )
```

where `BinaryTree[exprInfo]` is a binary tree of `exprInfo` containing information on the type of node (operator or operand) and, if it is a literal, its value; or if it is a function, its parameter and result types. Figure 4.4 shows the representation of the constraint `NativeHead`.

Nodes of the tree are either operators (simple or aggregate functions or a special node indicating function application) or operands (either literals or identifiers). Simple functions include `*`, `+`, `div`, `and` and `rem`. Aggregate functions operate on collections of data such as `maps` or `lists`. These include `UNION`, `DIFFERENCE` and `COUNT`. User-specified and aggregate (built-in) functions are stored in the same way in the tree using **apply** and **next-op** nodes: the **apply** node's left child is the function to be applied to the right child; its right child gives the argument(s). When there is more than one

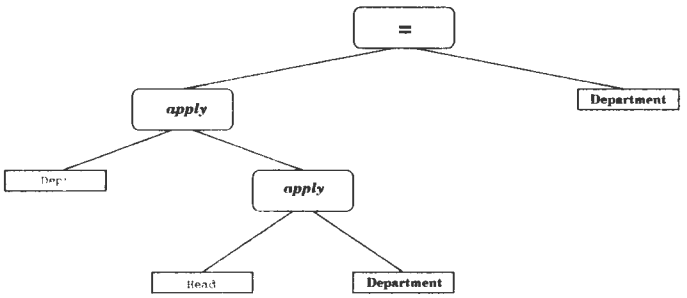


Figure 4.4: Representation of constraint `NativeHead`

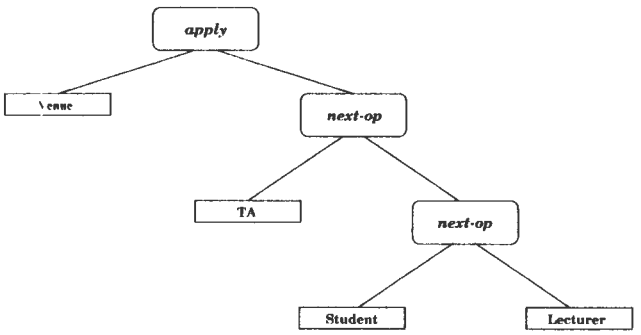


Figure 4.5: `Venue(TA,Student,Lecturer)` is represented using **next-op** nodes

argument these are represented by a chain of one or more **next-op** nodes, as shown in Figure 4.5.

Syntactically, user-specified functions (like `Manager`) and built-in functions (like `UNION`) appear the same. Once the constraint is organised as a tree, the tree is traversed to find the user-specified functions by querying the DR. The subtree for such a function is then transformed to a compressed version where that subtree is replaced by a single node. Type checking is performed at the same time, and is used by the constraint code generator (to produce correct code) and can be used by a change management system (providing additional dependency information). Below is the representation of the constraint `PaidEnough` before (Figure 4.6) and after (Figure 4.7) the user-specified functions are identified and transformed.

```
DEFINE CONSTRAINT PaidEnough ->
       Salary(Employee) * 2.4 < Salary(Manager(Manager(Employee)))
```

In these figures, the FDM functions `Salary(Manager(Manager(Employee)))` and `Salary(Employee)` are identified by querying the data repository. Each of these functions is then stored in a compressed form together with references to its associated data repository object (an entity, a relationship or an attribute).

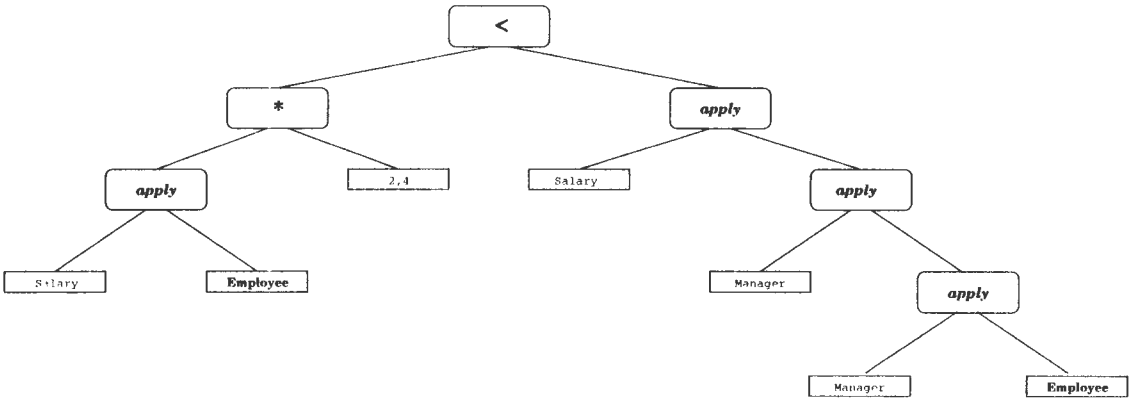


Figure 4.6: Constraint before FDM functions identified

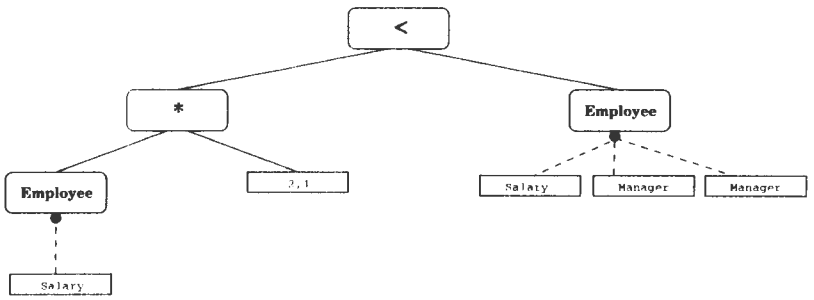


Figure 4.7: Constraint after FDM functions identified

Type checking needs to be performed on the operator-operand tree to identify user-specified and built-in functions and verify the result and parameter types of these functions. The tree is compressed when the type checking is done so that the constraint code generator can traverse the tree more quickly.

Information is collected together in two maps within the target system's data model: `constraints`, a `map` holding all the constraints and `code`, which maps entities onto the set of constraints the entity is limited by.

Once the constraint is parsed and stored in the data repository, a Napier88 procedure can be generated to check the constraint.

4.3.3 Generation of Napier88 code

The generated constraint code is stored as a `Process` in the process model part of the design. The parameters are the types of the entities the constraint is limited to, and its result is `bool`.

For the constraint **PaidEnough**, the following Napier88 code is generated if the Reference approach was used in the Type Generator:

```
PaidEnough := proc( employee : Employee -> bool );
  employee(Salary) * 2.4 < employee(Manager)(Manager)(Salary)
```

The body of the constraint is obtained from an in-order traversal of the operator-operand tree, with the following operation at each node:

- if the node is a literal it is output; or
- if the node is a user-specified function and it is one of the parameters of the constraint, the name of the parameter (e.g. **employee** above) is output; and
- the user function is output (as Napier88 code) in the reverse order to the way it was specified. This is due to the way most programming languages specify fields, for example: the Daplex specification **Salary(Manager(Employee))** is in the reverse order of the Napier88 specification of the field **employee(Manager)(Salary)** for variable **employee** of type **Employee**.
- aggregate functions are translated to library calls as described below.

Aggregate functions, such as **COUNT**, **DIFFERENCE**, **UNION** and **INTERSECTION** can be implemented using the corresponding Napier88 Glasgow Libraries[ABC+93] functions **m_length**, **m_diff**, **m_join** and **m_intersection**, or **l_length**, **l_diff**, **l_join** and **l_intersection**. Here, we give a description of how **COUNT** was implemented, and outline how to implement other aggregate functions.

Aggregate functions take as arguments bulk types (Maps, Vectors or Lists) since the Type Generator creates Maps to represent collections, Lists to represent to-many relationships and Vectors to represent to-N relationships in the target system. To distinguish between these, and for type-checking purposes, type information on the parameters and the result of each function is stored in the operator-operand tree. This type information also differentiates between entities, relationships, attributes and literal values.

For example, the aggregate call **COUNT(Employee)** would be converted to **m_length[string,tEmployee](mEmployee)** and the return type **int** would be stored.

Expressions, such as **COUNT Student SUCH THAT Mark(Student) > 50** can be evaluated using the **m_filter** or **l_filter** library routines, so that the following would be generated:

```
m_length[int,tStudent](m_filter[int,tStudent](mStudent,
 proc(studentNo:int; student:Student -> bool); student(Mark) > 50 ))
```

4.3.4 Using constraints

Once stored in the repository and incorporated in the process model, the constraint can be used by procedures that update data that may violate the constraint. In particular, each generated read handler (which creates a new instance of the type associated with an entity) calls the procedures to check all the constraints the entity is involved in (if any). If the constraint is not satisfied, the instance of the entity input by the handler is not stored in the entity's associated (extent) map. That is, if any constraint is not satisfied, the entity is not inserted in the store.

For example, suppose the **Employee** entity is involved in two constraints: **PaidEnough** and **ValidAge**. Its read handler (using the Reference Approach) would be:

```
readEmployee:=proc( -> tEmployee)
begin
   ...

 if ~PaidEnough(employee) then
   writeString("constraint PaidEnough failed'n")
 else if ~ValidAge(employee) then
   writeString("constraint ValidAge failed'n")
 else  m_isu_insert[string,tEmployee](mEmployee,employee(name),employee)

 employee
end
```

As with the handlers, the constraints are stored in the process hierarchy of the target system's process model.

4.3.5 Conclusion

Constraints are represented as operator-operand trees, with nodes that differentiate between aggregate, user-specified and simple functions. The Napier88 code for the constraint is generated and stored as part of the process model for the target system. Programs can use the constraint procedure (especially when updating data); in particular, the read handler for each entity calls the constraints it is involved in and aborts the insertion of the entity if any constraint fails.

4.4 Integration of OMT

OMT [RBP+91] comprises three separate modelling mechanisms: an object model, a dynamic model and a functional model, each of which is outlined in this section. The extension of the repository structures to cater for OMT is also discussed.

4.4.1 The Object Model

The object model of OMT can be considered an extension of the Entity-Relationship model. Classes are analogous to entity sets and links are analogous to relationship sets. However, classes and links can have methods and classes have part-of / made-of hierarchies. Although some extended ER implementations do have support such aggregation, the DEN prototype did not. The data repository was extended by incorporating this additional information.

The changes to the repository for entities (classes) included adding methods, differentiating between abstract and concrete entities (classes), allowing generalization with multiple inheritance, and adding part-of and made-of hierarchies. Entities (classes) may also have reference to the process (dynamic) model part of the design.

Relationships (links) were extended to include methods, and entities involved in a relationship (roles) could be ordered.

4.4.2 The Dynamic Model and the Functional Model

Dynamic modelling involves sequencing of operations. Its main concepts are events, states and state transitions; a state diagram can be associated with each dynamic model state and with each object model class.

Functional modelling deals with what the operations do; its main concepts are processes, actors, data stores and data flows.

In the representation of OMT's Dynamic Model, `Process` has a dual function: it represents a state diagram (if it has states or events) or it represents a process (in the structure chart sense – the design object representing an implemented procedure). To handle this duality, it was recognised that process modelling involves two activities: decomposition (structuring and subdividing a solution) and specification (describing the function of a solution). In the DR, decomposition is described through parent and children lists associated with `Process` objects. A leaf process (ie. it has no children) has a procedure specification associated with it.

4.4.3 Changes to the repository

The data repository structures `Entity` and `Relationship` were renamed to `Class` and `Link` respectively because these OMT constructs are more general versions of the respective ER constructs. Extending the repository to cater for OMT provided an opportunity to add other general information such as management data (author, date and description fields) and expansion fields (of type any) to each structure (DataModel, Class, Link, Role, Attribute, Type, ProcessModel, Process, Arc, Node and Procedure). Figure 4.8 shows the repository after the OMT additions (management information is

not shown). ProcessModel and DataModel are not shown in the figure. Code (procedure, handler and type) generation procedures as well as the repository query subsystem were updated for the new repository to handle the extra information.

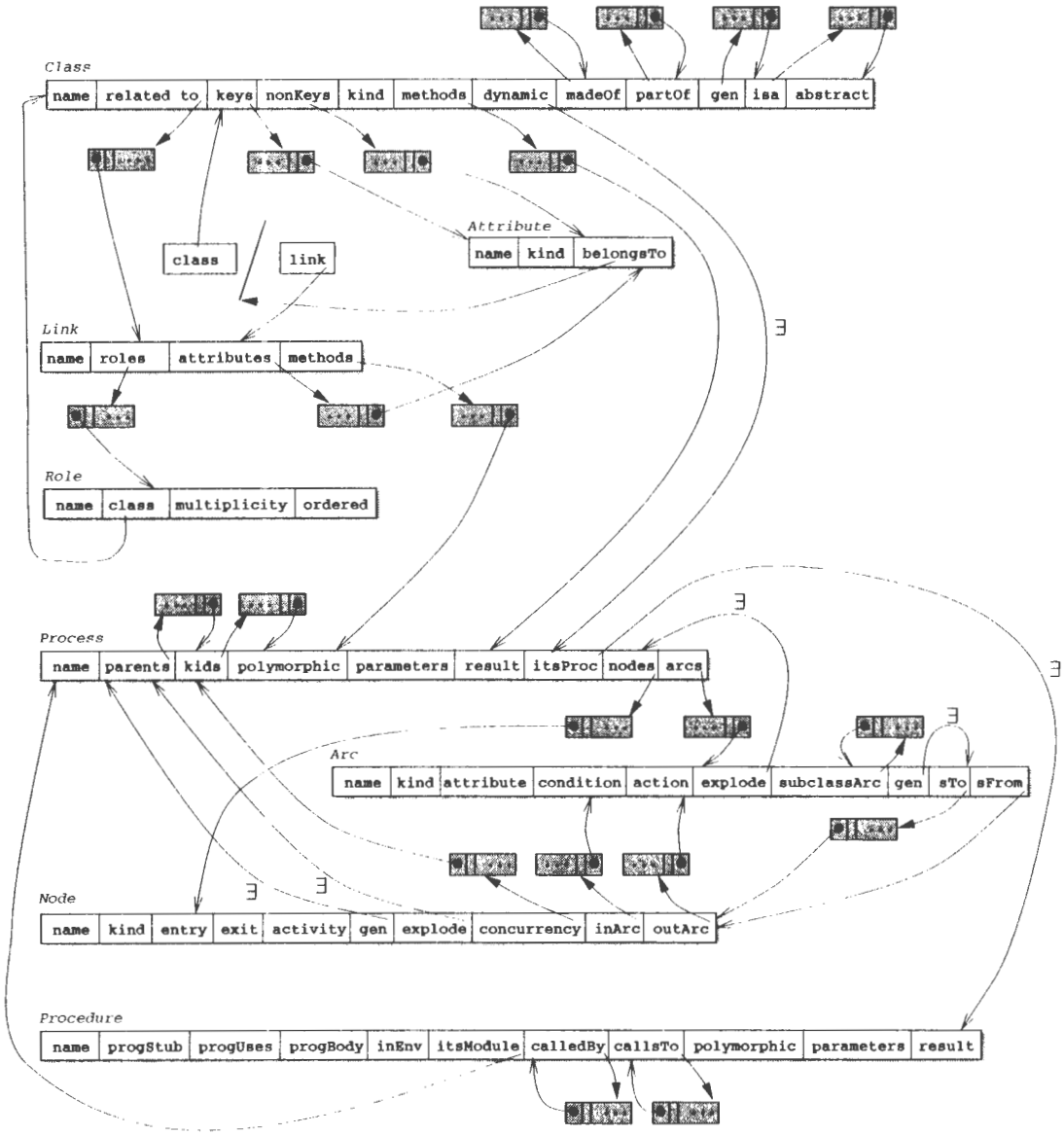


Figure 4.8: Representation of the OMT-based repository

For type generation, additions to the repository are handled in the following ways:

- **multiple inheritance**: A field is generated for each superclass. This is of the form:

  isa*Superclass* : *SuperclassType*

The superclass could alternatively be embedded in the type declaration by copying its attributes and methods. The Handler Generator, however, has not currently been extended to provide this alternative.

- **methods**: For each class, an additional type is generated that encapsulates all the methods of the class. The type declaration generated for the class has a field called `operations` that references an instance of this type. The field is of the form:

  `operations :` *Type*`_methods`

  The *Type*`_methods` has fields of the form:

  methodName : `proc(` *methodParameters* `->` *methodReturnType* `)`

  An alternative approach is to have all methods directly in the type representing that class. This approach uses more space (since there is a reference to each method for each instance of a class) but requires fewer dereferences.

  Methods are not generated for links. Rumbaugh [RBP+91] suggests that links with methods should rather be converted to classes.

- **aggregation hierarchies**: Part-of and made-of hierarchies are generated in a similar way.

  A field is generated for each (aggregate) class the class is "part of". The type of this field is a variant that implements optionality. It is of the form:

  `partof`*AggregateClass* : `Optional[`*AggregateClassType*`]`

  For the aggregate class, a field is generated for each component of the aggregate. The type of the field is a list of the component objects. The field is of the form:

  `madeof`*ComponentClass* : `List[`*ComponentClassType*`]`

  For example, if wheel is part of car, the following types are generated:

```
type Wheel is structure(···partofcar: Optional[Car];···)
type Car is structure(···madeofwheel: List[Wheel];···)
```

The generated handlers were changed to cater for the above fields using the same techniques used for associations. These include calling the handler for the aggregate, component and specialised class when instances of the class are input, output and initialised and using keys when they are input or output.

4.5 Co-existing data models in a general data repository

The building of the DR made evident that more than one model could be supported using a single repository. This can be seen when integrating a new model: the DR

caters for all the constructs in this model, and the DR does not need to be changed in order to accommodate the new model.

This section describes how new data and process models can be incorporated in the system by taking advantage of properties of the data repository. It concludes with an example showing how a design can be viewed from different modelling perspectives.

4.5.1 Model Integration

The first step in integrating a new model in the DR is to find a mapping from the model's conceptual constructs to the structures in the DR (see Section 4.2.3).

At the same time, a reverse mapping needs to be found so that the DR can be correctly represented using the constructs of that model. This is an interpretation of the data repository for this model.

These mappings allow many models to be supported by a single DR where each conceptual construct (modelling primitive) is uniquely represented.

There are a number of conceptual data modelling abstractions: aggregation, generalization, classification, association, derivation and constraints. Since all current models use some subset of these abstractions, a repository that can represent all the above constructs should be able to represent any conceptual model. While building the repository, it became evident that more than one model could be accommodated by this single complex object. This was seen when integrating FDM into the system: the repository catered for all its constructs except constraints and did not need changing to accommodate the model.

In future, new modelling constructs may be invented. A method is needed to represent such constructs in the repository. One possibility is to assume that any new modelling primitive can be represented as some combination of existing constructs. An alternative approach is to provide a minimal set of primitives, and define all existing and future models in terms of this set. A third alternative is to provide a component in each construct (eg. the `extra:any` field in our system) where new primitives or properties can be accommodated. In chapter 7 we present some experiments on the use of multiple models in our repository where the last alternative was user to cater for some features unique to SDM [HM81].

4.5.2 Changing Model Perspective – Example

This section illustrates the viewing of an ER design as an FDM model.

The ER model is stored in the data repository. For each entity (object) in the repository, the Daplex declaration (`DECLARE` *Entity*`()` `->` `ENTITY`) is output. For subtypes,

the declaration (**DECLARE** *SubType*() -> *SuperType*) is output. Similarly, all the attributes of the entities and the relationships (functions) between entities are also output.

This process is the reverse of the one to input an FDM model into the DR, which is the mapping of FDM to DR constructs.

For the university example (see Section 2.3.3), Figure 4.9 is the Daplex output of the ER-specified data model. In this example, the **Employee** entity is a subtype of the

```
DECLARE Person() -> ENTITY
DECLARE Employee() -> Person

 ! Employee attributes
DECLARE EmployeeNo(Employee) -> int
DEFINE Employee(int) -> INVERSE OF EmployeeNo(Employee)
DECLARE ClockOut(Employee) -> string
DECLARE ClockIn(Employee) -> string
DECLARE Status(Employee) -> string
DECLARE HourlyPay(Employee) -> real
DECLARE hasAddress(Employee) -> Address
DECLARE hasEmployees(Employee) ->> Employee
DEFINE hasBoss(Employee) -> INVERSE OF hasEmployees(Employee)

 ! Person attributes
DECLARE Name(Person) -> string
DEFINE Person(string) -> INVERSE OF Name(Person)
DECLARE IDnum(Person) -> string
DECLARE DOB(Person) -> string
```

Figure 4.9: Part of FDM output of an ER model

Person entity and **Employee** has **EmployeeNo** as a key.

Unfortunately, the differences between conceptual data models implies that conversion is not always seamless. Models differ in expressive power, since not all constructs are represented in every model. The following ER constructs cannot be represented in Daplex:

- multiple (alternate) candidate keys

  In the ER model, an entity **Employee** can have multiple candidate keys **EmployeeNo** and **SocSecNo**. That is, it has two integer keys.

Keys can be specified in Daplex as (for the **Employee** example): `DEFINE Employee(int) -> INVERSE OF EmployeeNo(Employee)`. However, `SocSecNo` is also a key of **Employee** and so its definition in Daplex would be: `DEFINE Employee(int) -> INVERSE OF SocSecNo(Employee)`.

- multiple-attribute relationships

  The FDM data model does not have explicit association; so a FDM view of an ER model will not be able to represent relationships with more than one attribute properly:

  > The relationship between **Employee** and **Project** has two attributes: **Task** and **Hours**. In the FDM model, there can be only one attribute per relationship between two or more entities, and so the above two-attribute association cannot be specified. The specification as two separate functions:
  >
  > ```
  > DECLARE Task(Employee,Project) -> STRING
  > DECLARE Hours(Employee,Project) -> INT
  > ```
  >
  > does not capture the fact that the two functions refer to the same association. In order to represent this using Daplex, an **Assignment** entity needs to be created.
  >
  > ```
  > DECLARE Assignment() -> ENTITY
  > DECLARE Task(Assignment) -> STRING
  > DECLARE Hours(Assignment) -> INT
  >
  > DECLARE WorksOn(Employee,Project) -> Assignment
  > ```

4.5.3 Conclusion

Most data models have the same expressive power, and integrating several data models in a single design system is possible by mapping the constructs of each onto a general framework such as that of our Data Repository. The same appears to be true of process models, although we have not concentrated on this aspect. The inclusion of the ER, FDM and OMT models served to illustrate that multiple models can be integrated in this way and to indicate that our repository is sufficiently flexible to support an even wider range of models. We return to this aspect in Chapter 6.

Chapter 5

The Persistent Workshop

The Workshop [WPA+97] is a software architecture for building persistent systems. It uses an abstraction that sees a persistent programmer as a carpenter having tools and items that the tools work on. Each workbench is designed to support a particular activity, with tools and items moving between workbenches over time. The Workshop developers built a single workbench, the "Programming Workbench", that helps in editing and managing persistent application systems. Its tools include editors, compilers, store visualisers, source code completers, pretty-printers and an application builder [SPWW97]. A set of persistent system design tools needs to be integrated with the tools that support the subsequent implementation phase of the software lifecycle. The Workshop was an obvious candidate, and so the system was incorporated in this workshop as a Design Workbench. It also served as a vehicle for studying the interaction of multiple workbenches sharing tools and workitems, and the ease with which new systems can be incorporated into the Workshop. This chapter describes the workshop architecture, outlines the steps involved in adding the new workbench and concludes with the benefits derived from this.

5.1 Workshop abstraction

The Workshop consists of a number of *workbenches*. Each workbench contains a set of *work tools* and a number of *work items*. Tools interact through an established *communication space* on each workbench. They are the routines that alter the state of the workbench, by modifying, querying or creating work items. A set of Workshop rules specify the type, organization and behaviour of tools and work items. These rules include updating management information, the type of interface a tool must have and the structure of work items and tools.

In the "Programming Workbench", work items include Program, Declaration Set and PAS work items. The first two correspond to programs and types and the latter are

collections of the three work items. Typical work tools are a compiler or a pretty printer.

5.2 Integrating the new workbench

5.2.1 Updating program code

As the Design Environment was written in Napier88 release 1.0 and the Workshop in release 2.0, all source code was replaced by the new release equivalent using an `awk` shell script. At the same time, the program files were sorted into a Unix directory hierarchy isomorphic to the environment hierarchy on the store. This makes it easier to locate procedures on the store and the source files on the file system. The Glasgow Libraries naming convention for program files [ABPW94] was also used.

Prior to integration, source files contained code for one or more procedures. The procedures in a single file shared variables and called each other. These files were stored in a similar, but less rigid, hierarchy. Changing the code of a procedure was considerably slower since the other procedures in the source file would also be compiled. To move a procedure to another position in the hierarchy would require uncoupling the dependencies on the other procedures and values in the file where it had originally been stored. In contrast, the new organization requires this independence by forcing a separation of procedures into different files. This is consistent with the library organization prescribed by Atkinson *et al* [ABPW94]. This forced reorganization was very worthwhile since it replaced an *ad hoc* program file structure with one that made it easier to maintain the development environment.

5.2.2 Integration of the development environment

The first step in integrating the system as a second workbench was to identify its tools and items. The individual components of the four main subsystems became the tools:

- data model input
- process model input
- type generation (from data model)
- handler generation (procedures to manipulate instances of these types)
- program generation (environments, data structures, procedure stubs and skeleton code)
- view data model

- view process model
- data repository querying
- share tool (copies a work item to another workbench)

More tools were added later when the advanced features were implemented, as described in the next chapter.

Different representations of programs etc. had been used by the two workbenches. To permit migration of items without loss of information, the data repository types had to be changed: data found in the other workbench (but stored differently) was replaced by an object having the type used in the Programming Workbench. This typically meant that the revised type now contained values not used within our system; we benefitted from this additional information and the associated functionality provided by the other workbench.

Workbenches, tools, items and communication spaces have a defined structure that a workbench designer must conform to. Rewriting the code to fit the abstraction involved two types of transformation. The first involved coding an additional layer of procedures that take work items from the workbench and use them as input to the original procedures for each tool. This was necessary since the original system manipulated all design objects through the repository. The new procedures fetch, insert, replace and remove work items on the workbench. The code for the additional layer procedures was extensively reused since the operations on design objects for each tool was essentially the same (fetching, inserting, deleting or altering work items).

The second transformation changed the tools to fit the Workshop abstraction by passing work items as parameters. Procedures that queried or altered one of the data repository maps are now passed a `DesignWorkItem` object which has the maps as fields. This `DesignWorkItem` type encapsulates all the parts of a system design: data and process models, types, handlers, bulk variables, constraints and programs:

```
type DesignWorkItem is structure (
        constraints : Map[string,Constraint];
        data : Map[string,Data];
        types : Map[string,Type];
        variables : Map[string,Variable];
        handlers : Process;
        processes : Map[string,Process];
        root : Process;
        datamodel : DataModel;
        storeByReference : bool
)
```

The **handlers** field is the process hierarchy containing all the handlers generated from a data model. The **root** is the default process from which program generation and process model viewing start.

The **DesignWorkItem** did not exist in the original system, which used the repository comprising separate maps for data models, process models, types, constraints and variables. At first the individual design components (types, bulk variables, etc.) were used as separate work items, but it became difficult to determine which component came from which design with this approach.

5.3 Interaction between workbenches

The two workbenches, one dedicated to programming, the other to design, can be used together to take advantage of each other's features.

This can be seen in the following example where a program is generated in the design workbench and is edited and compiled in the programming workbench:

> The university data model used in Chapter 2.3 is input to the design workbench as an ER specification. Types are generated from the data model, followed by handler code. A PAS work item is then generated for the entire handler code hierarchy. The share tool (described in section 5.2.2) is called to copy the reference to the PAS work item to the programming workbench.
>
> On the programming workbench, the PAS work item is selected, and a particular handler is chosen from it. The program editor is called, and the user can make changes to the handler. From the programming workbench, the user may compile the new handler.

Even though the handler source code exists on both workbenches, changes to the handler are not lost since the share tool copies the reference to the handler.

5.4 Benefits

Integration in the Workshop produced a more structured implementation of the system; and having the tools conform to Workshop conventions made them independent of other programs; so the resulting system is easier to maintain. Integration amounts to changing the interface between system and user, and between tools. The new DesignWorkItem type developed in this way made it easier to manipulate a design as a whole, and to keep different target systems separate. The user could only work on one design at a time and all tools would operate on that design.

The Workshop provides a standard, consistent graphical user interface that consists of work item and tool menus for each workbench. A workbench also has dedicated input and output windows available to workbench tools. The user interface of the original system was significantly improved (from a textual one) by using the provided interface.

Integration of the Design Workbench provided the first opportunity to study the interaction between workbenches in the Workshop. The communication between workbenches includes the sharing of work items and tools and the interaction between tools on different workbenches. Tools, such as the "Share" tool that copies references to work items to another workbench, are shared between workbenches.

The Workshop, with the two workbenches (Design and Programming), is a more usable product since it covers more of the the software development life cycle. An end-user is able to work with the two systems easily in a single cohesive environment although the systems were created completely independently.

The procedures and types generated by the Design Workbench were of greater utility since these could be used by the Programming Workbench to edit, compile and run the source code to produce the target system.

The code generated from a design can be edited outwith the control of the Design Workbench. This means that subsequent changes to the design that affect the generated code could replace these user edits. Change management is needed to allow design changes without losing user edits to generated code. This is discussed further in chapter 7.

5.5 Conclusion

The system was successfully integrated as a "Design Workbench" in the Glasgow Workshop. This required updating the source code for release 2.0 of Napier88, modularising the source code, introducing a new type of work item and adding a tool to aid workbench interaction. The extended workshop facilitated a more structured set of design tools, provided the design tools with a consistent GUI, demonstrated the interaction between workbenches and illustrated the relative advantages of the integrated workshop.

Chapter 6

The Metamodeller

Support for multiple co-existing models can be taken a step further by allowing arbitrary models to be incorporated in a single data repository. Once any model is fully specified, it can be supported in the same way as the original Design Workbench models.

The Metamodel component of the Design Workbench allows users to specify their own models by supplying the model's input grammar and its mapping onto the repository. A compiler-compiler (similar to Yacc [Joh75]) was written by an Honours student [MO95] and this was adapted and incorporated in the workbench as the first metamodelling component. Textual input by end-users is retained for the sake of speedy design, so a model is specified by giving the grammar of its input language interspersed with actions which arrange for identifiers to be stored in the repository. The compiler produces programs that parse models (such as the ER parts-supplier model in Figure 3.4) and store them in the data repository.

A Repository Library, consisting of procedures that manipulate the repository, is supplied to metamodel users so that they can update the data repository in a safe and consistent way.

Metamodel users can also input a form of SQL `VIEW` definition that specifies the end-user view of a stored model. The compiler-compiler parses model views and produces programs that perform user queries of the data repository given the specified view.

This chapter describes the two subsystems of the metamodeller in turn: model creation and model query. Thereafter, section 3 presents aspects of the metaquery implementation in more detail, and the chapter concludes with a discussion on GUI specification for new models. We use the following terminology to distinguish between the two phases in our extended Design Workbench: the stage at which a new data/process model (eg. IFO [AH87] or DSPD [Han83]) is defined is referred to as *metamodelling* and the person responsible for this is the *metamodel-user*; the actual design where this model is used to describe a target application is called *designing* and is done by *end-users*.

6.1 Model input and edit

The first step in incorporating a new model is to specify how it will be stored in the data repository. The metamodel-user firstly needs to specify the model's representation in the repository and its input language syntax. This is given as a grammar for the input and edits together with actions that insert or modify the model in the data repository.

The metasystem generates a "compiler" for a given grammar (that is, a program to parse models input as text and update the repository accordingly). This generated program is called the Model Builder for this grammar, since it will process subsequent end-user designs formulated using the new model. To facilitate the coding of "actions", procedures for inserting tokens into the repository exist in the store (the Repository Library), so these simply need to be called with the correct parameters. A metamodel-user merely needs to know of these action procedures in order to define her model.

6.1.1 The Repository Library

The Repository Library (RL) is a structured group of procedures that modify the data repository. Calls to these procedures are embedded as actions in the grammar defining a data or process model. For a particular end-user design that is parsed by the generated Model Builder, the relevant RL procedures are called to update the DR accordingly.

The RL is structured using naming and calling conventions. The naming conventions include prefixing the names of RL procedures with "`set_`", "`begin_`" and "`end_`".

The "begin"-"end" library procedures alter the main DR objects. A "begin" library call creates a new instance of the DR object in the repository for the particular design if it does not exist. The "begin" procedures take in the name of the DR object. Each subsequent "set" procedure before an "end" call will operate on this DR object. The "begin" procedures are called first, followed optionally by a number of "set" procedures and completed with an "end" procedure for each complex DR object.

The "set" procedures assign values to components of DR objects (such as the name, kind or multiplicity). Each DR object has a number of "set" procedures that update parts (fields) of that DR object. These are of the form: `set_`{*DRObject*}{*Field*}. For example, the RL procedure to set the "kind" field of an attribute is called `set_AttributeKind`. For each complex DR object there is a procedure that updates the extension field of the object. The name of the procedure is of the form: `set_`{*DRObject*}`Extra`.

The data repository objects that have RL "begin"-"end" procedures are Design, Method, DataModel, Class, Attribute, Role, Link, ProcessModel, Arc, Node, Process and Procedure. An outline of the order in which RL procedures are called is given below (note that within a **`begin-end`** pair, calls can be made in any order):

- `begin_Design:proc( string )`

- `begin_DataModel: proc ( string )`
  - `begin_Class: proc( string )`
  - `set_ClassAbstract: proc( string )`
    set other Class fields ...
    - `begin_Attribute: proc( string )`
    - `set_AttributeAuthor: proc( string )`
    - `set_AttributeDate: proc( string )`
    - `set_AttributeDescription: proc( string )`
    - `set_AttributeKind: proc( string )`
    - `set_AttributeExtra: proc( any )`
    - `end_Attribute: proc( )`
  - `end_Class: proc( )`
  - `begin_Link: proc( string )`
    set Link fields ...
  - `end_Link: proc( )`

  other Classes and Links ..
- `end_DataModel: proc( )`

  similarly `ProcessModel`

- `end_Design( )`

6.1.2 Model grammars

The input specification is in the form of a context-free grammar, similar to the format used by Yacc [Joh75]. The grammar gives the model building language rules together with grammar actions that insert designs into the data repository.

Grammar rules are of the form: *lhs* "->" *rhs*. The left hand side of the rule (*lhs*) must be a nonterminal and the right hand side (*rhs*) is a disjunction of conjunctions of nonterminals and terminals. Disjunctions are separated with a "|". A number of grammar actions can be placed anywhere on the right hand side of the rule. For model input, the grammar actions are usually Repository Library procedure calls.

Figure 6.1.2 is the context-free grammar of a subset of the ER input language in section 3.2.1. Note that this grammar has a number of actions that are not Repository

```
start -> "ER" id                          { begin_DataModel $1 }
         entities                         { end_DataModel }
       ;

entities -> id                            { ERinitialise_EorR $1 }
            entity_or_relation entities
          | "END"
          ;

entity_or_relation -> "(" id ")"          { ERset_RoleFrom $1 }
                      arrow role
                    | "--"                { ERinitialise_Entity }
                      attributes          { end_Class }
                    | arrow role
                    ;

role -> id _role                          { ERset_To $1 }
      ;

_role -> "(" id ")"                       { ERset_RoleTo $1 }
         attributes                       { end_Link }
       | attributes                       { end_Link }
       ;

attributes -> "[" id "]"                  { ERset_Key $1 }
              attributes
            | id                          { ERset_nonKey $1 }
              attributes
            | "."
            ;

arrow -> "<->"                            { ERone_to_one }
       | "<<->"                           { ERmany_to_one }
       | "<->>"                           { ERone_to_many }
       | "<<->>"                          { ERmany_to_many }
       | "<=>"                            { ERweak }
       | "ISA"                            { ERisA }
       ;
```

Figure 6.1: Context-free ER input grammar

Library procedures. These non-RL procedures simply store information that is later passed to RL procedures.

Action procedures prefixed by "**ER**" in this figure are responsible for calling RL procedures appropriately. For example, entity (or relationship) names are temporarily stored by **ER_initialiseEorR**; after the operator (called "**arrow**" in the figure) is parsed, either **begin_Class** or **begin_Link** is called (depending on the operator), using the name stored by **ERinitialise_EorR**.

6.1.3 Model Builder generation

The model input and edit component of the metasystem is made up of a compiler-compiler and Repository Library. The compiler-compiler called **Sacc** (**S**till **a**nother **c**ompiler-**c**ompiler), generates a Model Builder that inserts end-user designs into the data repository. This Model Builder is inserted into a Map of model builders so that, for a given model, the associated builder will be called.

The metamodeller has one restriction on the model grammar: it must start with the name of the model (eg. "ER", "OMT", "DFD", etc). When given an end-user design, the system looks up the associated builder in the Map of model builders by comparing the name of the model with the set of known models.

The Repository Library was adapted from the internal data repository manipulation procedures. This involved reorganising and renaming the procedures in a systematic fashion so that their functionality would be clear to the metamodel-user.

6.2 Model query

As discussed in Chapters 3 and 4, the Design Workbench Query tool accepts SQL-like statements, given in terms of the end-user's conceptual model, to allow for interrogating a design. The metasystem therefore has to include a facility whereby the end-user's view of a stored design can be defined, so that queries will be interpreted correctly. A metamodel-user defines how a stored model can be queried by specifying the relationship between the data repository's representation of the model and the end-user (designer) representation.

The metasystem user specifies a variation of a SQL `VIEW` statement that describes how data repository elements are seen in the model, as show in Figure 6.2.

```
DEFINE VIEW ER
[
 TABLE Entity = (SELECT name,keys,kind,nonkeys FROM class),
 TABLE Relationship = (SELECT * FROM class,link
     WHERE link IN class.relatedTo)
 TABLE Attributes = (SELECT name,kind,belongsTo FROM attribute)
]
```

Figure 6.2: Simplified view definition for an ER model

The keyword "`TABLE`" is used to refer to the main objects in the view. The set of identifiers after the "`SELECT`" keyword define the attributes associated with a table. These attributes are referred to as the columns of the table.

For this, the metamodel-user is given an outline of the relevant parts of the data repository. Effectively she works with a view of the repository, based on which she defines the view to be presented to end-users.

The **base model** is the set of tables (object collections) presented to the metamodel-user as their view of the data repository. This model is a nearly one-to-one mapping from the DR data structure (but with some fields and DR objects omitted). The omitted fields and objects are largely management data.

6.2.1 View definition

The grammar for specifying model views is given in Figure 6.3. A view is specified as a number of tables. Each table is a projection from the cartesian product of base model tables, filtered by a collection of predicates. Table columns can be explicitly named by listing the names of each column after the table name (instead of the default action that uses the base model field names). Since the "tables" in the base model are not normalised – ie. their values need not be atomic – the dot notation is used to reference attributes of attributes. For example, to select the names of all the classes that a class is related to, the specification is: `class.links.class.name`. This notation is also used for variants: the name of the variant tag can be projected by listing it as if it were a column. For example, `belongsTo` in the attribute base table is a variant which is either a class or a link. The projection `nonkeys.belongsTo.class.name` will return a null value for this column if the (nonkey) attribute is an attribute of a link.

view	→	"`DEFINE VIEW`" *id* "`[`" *table-list* "`]`"
table-list	→	*table* ["`,`" *table*]
table	→	"`TABLE`" *table-name* ["`(`" id_1 "`,`" ⋯ "`,`" id_n "`)`"] "`=`" "`(`" "`SELECT`" *column-list* ["`FROM`" *table-name-list*] ["`WHERE`" *where-list*] "`)`"
column-list	→	*column-name* ["`,`" *column-name*]*
	|	"`*`"
column-name	→	*id* ["`.`" *id*]*
table-name-list	→	*id* ["`,`" *id*]*
where-list	→	*where* [["`AND`" | "`OR`"] *where-list*]*
where	→	[*column-name* | *constant*] *operator* [*column-name* | *constant*]
operator	→	"`=`" | "`<`" | "`>`" | "`<=`" | "`>=`" | "`~=`" | "`IN`"

Figure 6.3: Model view specification grammar

6.2.2 Query specification

An end-user first chooses the view with which he wants to query the data repository. The query syntax is identical to the `SELECT-FROM-WHERE` part of the view grammar

given above, ie.

SELECT *column-list*
[**FROM** *table-list*]
[**WHERE** *where-list*]

The following is an example of an end-user query for the ER view in Figure 6.2:

```
SELECT name,kind
FROM Attributes
WHERE belongsTo.class.name = "Course"
```

6.2.3 Query execution

Before describing the metasystem itself, we outline the overall structure of a repository querying program. Its top-level procedures are:

- a **parser**, which reads and checks the end-user query and builds an internal representation of the request.
- a **query** procedure, which iterates through all combinations of DR objects required and, if they satisfy the "**WHERE**" predicate, calls output procedures to present these values to the end-user. The conditions tested are in fact a combination of the view definition and the end-user query criteria. We will refer to this as the merged-predicate in what follows.
- a collection of **output** procedures, which display DR values.
- a **filter** procedure, which tests if the merged-predicate is satisfied.

6.3 Metasystem query implementation

6.3.1 Overall approach

The query program needs to map queries to DR objects in order to access the correct information. This can either be done implicitly (in the code of the query program) or explicitly (where the structure of the base model and view is stored as metadata). Each of these approaches is outlined below.

The query engine could have code to output all the fields of all the DR object structures, and use **if/case** statements to determine which to execute for a given query. They could be grouped together so there is one such procedure per base model table. So for the **Design** structure for example, the procedure could look like:

```
select_Design := proc( column:string; design:Design )
begin
  case column of
   "name"          : writeString(design(name)++"'n")
   "datamodels"    : m_app[string,DataModel](
                                              design(datamodels),
                                              select_DataModel
                                            )
   "processmodels" : {...} !code to select process models
   "handlers"      : {...} !code to select handlers
  default : {}
end
```

This example is presented in a simplified form (to aid clarity): there is no column name checking, and only one column is being selected.

This code outputs those fields of `Design` matching the columns of the query. The disadvantage of this solution is that it requires the query subsystem to "hard code" the statements to output fields of DR structures. Should the base model or the DR structure need to be changed, the subsystem would need to be modified and recompiled.

Alternatively, information on the model and its mapping to the underlying DR can be stored in a data structure which is subsequently used to interpret queries. Changes in the DR can then be absorbed more easily because this data structure is simply regenerated from the new DR declarations. As a result, the first design decision taken when implementing the metaquery system was that is should be data-structure driven.

More importantly however, the implicit approach is problematic for end-user queries that include a `WHERE` clause. The operands of the clause are not known at query subsystem run time, and so in order for the subsystem to interpret a query such as

```
WHERE class.name = nonkeys.belongsTo.class.name
```

all the possible combinations of operands would have to be listed, for each operator. A path through the DR which refers to components of nested structures (eg. `nonkeys.belongsTo.class.name`) is referred to as a *mapping* following the SDM terminology [HM81]. A problem with the implicit approach to queries is that all possible mappings within any possible predicate need to be catered for. Coding this is not a viable option.

The end user *where* clause is only known at run-time (ie. when an end-user invokes the query tool). This means that in order to apply the merged-predicate filter procedure, the query tool must either cover all possible mappings explicitly via different cases in its code, or else it must generate code for a query at run-time. Clearly the former approach is not practical; the need for linguistic reflection in the query program was therefore our second design decision.

Another issue that arose in implementing the metaquery system was whether to create a single universal query program, or to have multiple model-specific Query Engines. The latter seemed preferable since it can accommodate differences in style/interface across models, and permits a query system to be tailored to the preferences of the metamodeller. Query engines are generated at metamodelling time, instead of having a general query interpreter that retrieves view information and then queries the data repository accordingly. The general program would be less efficient than a model-specific one, since it has to interpret the model view for each end-user query; with our approach the model view is only interpreted when the model-specific query engine is generated.

To summarise therefore, the overall approach is to implement a data-driven system which generates a specific Query Engine for each metamodel view, with a generated Engine itself dynamically building (and compiling) a filter procedure for the merged-predicate of each query.

6.3.2 Data structure

Executing an end-user query requires knowing about the base model, such as the names and types of its structures and fields (ie. metadata). As explained in the previous section, this knowledge is provided explicitly (in data structures) rather than implicitly (embedded in the query subsystem code), to make it easier to accommodate change.

This was designed as follows. Each base model object is stored in a "`Representation`" structure, holding its name (ie. of its type, eg. `Datamodel`, `Arc`, `Link`) and the fields of the object's structure. "`Field`" information gives the name of the field and information on its type. Simple and complex field types are differentiated. Complex fields are references to other DR structures and for these the data structure includes the kind of bulk structure (if any) used to represent n-ary relationships (ie. Maps, Lists, Vectors and an indicator if the value is optional) in addition to a reference to the corresponding "`Representation`". For simple types, the name of the type is stored.

The end-user queries her design using what we call a View of the DR, where the DR is seen from the perspective of her particular model. Views are seen as tables and columns, similar to relational databases. A data structure defining the Model View and its mapping onto the DR is derived from that which encodes the base model organisation, as shown in Figure 6.4. A more compact organization for View representation is used internally that includes "subtables". The subtables hold all the columns of a single complex object in the query together. For example, the view table for

```
SELECT keys.name, keys.kind, name FROM class
```

will have two subtables: `class` and `attributes` (keys). Subtables reference the base model object and the selected columns of this object. Columns reference the field in the base model's representation and (if the field is complex) the subtable. These base model items are of course obtained from the persistent store.

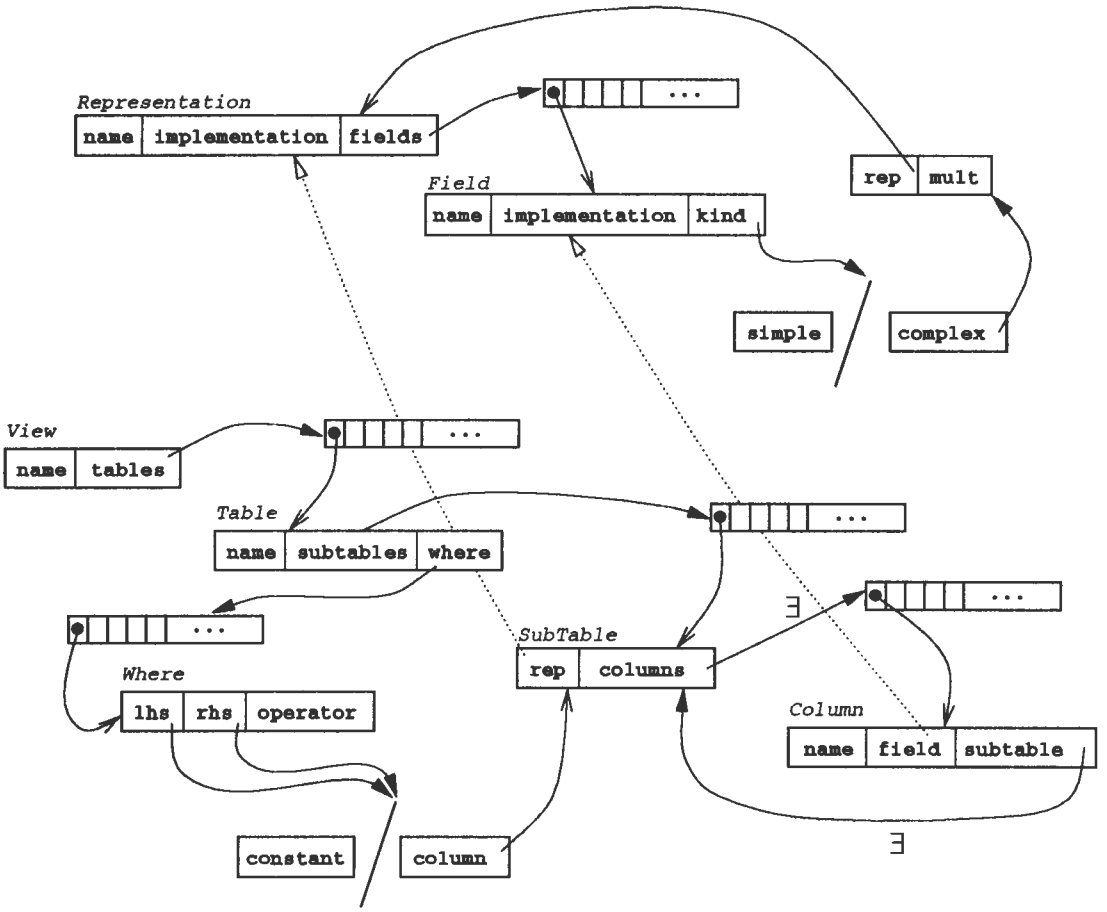


Figure 6.4: Diagram of query subsystem data structures

6.3.3 Code generation

The metaquery system is made up of the View Compiler (parsing metamodel views) and a Query Generator (that produces programs that perform end-user queries given the view). The View Compiler was generated using a context-free version of the meta query grammar given in section 6.2.1 as input to Sacc. Metamodel-user views are parsed and stored in the data structure described earlier.

The Query Generator is responsible for creating the **parser**, **query**, **filter** and **output** procedures that constitute a query engine, and for storing it in the map of Query Engines (similar to the Model Builder map). It recursively traverses the view data structure, generating code to output the corresponding data repository objects and field values.

Output procedure generation

A two-dimensional string vector is used to store the "**SELECT-FROM**" mappings in a query. It contains a string vector for each mapping in the list. The first string in each of these vectors is the table name, the last is the field to be output. For example, for the query "**SELECT name,keys.name FROM Entity**", the mapping vector consists of the mappings ["Entity", "name"] and ["Entity","keys","name"].

The top-level **output** procedure examines the first string in a mapping vector and, according to the table name appearing there, calls the corresponding `output_table` procedure. A simplified example is given in Figure 6.5. The manipulation of the mapping vector **userQuery** is not shown in the example for the sake of clarity. This involves extracting the first string to test in the "case" statement and then removing it from the mapping vector.

```
output_ER := proc(userQuery:**string;
                          class:Class; attribute:Attribute; link:Link)
begin
case userQuery of
   "entity":   output_entity(userQuery, class)
   - - - -
   "attribute":   output_attribute(userQuery, attribute)
   default:{}
end
```

Figure 6.5: A simplified **output**_modelName procedure

Each `output_table` procedure takes the next string from the mapping vector and displays that value (if it is printable). If it is an object (ie. a subtable of the view), the procedure that outputs this data repository object is invoked. For Lists, Maps and Vectors, a call is made to the respective iterator library procedure for that bulk type (ie. **l_app**, **m_app** or **v_app** respectively). A conditional statement is generated for variant fields; each variant branch is output in a separate part of the conditional statement.

In this way the `output_table` procedures call each other as reference paths are traversed through the DR, with each procedure removing the next string in the mapping from the mapping vector. A simplified example of an `output_table` procedure is given in Figure 6.6. Again, the manipulation of **userQuery** is not shown in the figure. For example, for the mapping "**keys.belongsTo.class.name**", the mapping "**belongsTo.class.name**" is passed to `output_Attribute`.

```
output_entity := proc(userQuery:**string; class:Class)
begin
case userQuery of
    "name" : writeString( class( name )++"'n")
    "keys" : l_app[Attribute]( class(keys), proc(attribute:Attribute);
                output_attribute( userQuery, attribute))
     - - - -
    "nonkeys" : l_app[Attribute]( class(nonkeys), proc(attribute:Attribute);
                    output_attribute( userQuery, attribute))
    default:{}
end
```

Figure 6.6: A simplified example of an output_table procedure

Filter procedure generation

An end-user **WHERE** clause is stored in the same data structure as the view's **WHERE** clause. The merged-predicate is then converted into a string representing a boolean procedure that has as its parameters the tables of the view. Each clause in the **WHERE** statement is converted to the corresponding code that compares the clause's operands. If an operand in the clause is a column, code is generated to reference the column's associated field in the data repository. The field may need to be referenced by accessing the fields of bulk data types; the generated code calls the corresponding bulk library procedures for Maps, Lists or Vectors.

As an example, consider the model view in Figure 6.2 :-

```
DEFINE VIEW ER[
TABLE entity = (SELECT name,keys,kind,nonkeys FROM class),
TABLE relationship = (SELECT * FROM class,link WHERE
                        link IN class.relatedTo)
TABLE attributes = (SELECT name,kind,belongsTo FROM attribute) ]
```

and the end-user query

```
SELECT classname, linkname FROM relationship
        WHERE classname = "Part" OR classname = "Supplier"
```

Note that since Class and Link have fields that have the same name (such as the field "**name**"), the corresponding columns in the joined table `class,link` are called

class*fieldName* and **link***fieldName* respectively (in the example, **classname** and **linkname**).

The query engine parses the **WHERE** clause of the query, combines it with those of the model View, and translates the merged-predicate into the following string:

```
filter := proc( link:Link; class:Class -> bool )
begin
      l_some[Link]( class(relatedTo),
          proc( link1:Link );
            link1 = link
      )
      and
      ( class(name) = "Part" or
        class(name) = "Supplier" )
end
```

This string is converted into an executable procedure using the callable compiler, and this is then employed to apply the predicate before displaying class - link combinations.

6.4 Conclusion

A metamodeller has been built that allows arbitrary conceptual models to be specified. These models can be fully supported by the Design Workbench once their mapping to the DR is given. The model interface and semantics are given by defining a means for the input, storage and querying of such models. A compiler-compiler called Sacc was built, and the Metamodeller then based on this. Conceptual models that have been specified via the Metamodeller are ER, Daplex, OMT [RBP+91], DFD [YC75], SDM [HM81] and structure chart [Tri88] models; these have all been used in subsequent system designs.

The Model View tool is the only remaining Design Workbench component that is model-specific. Future work is required to extend the metasystem to permit mappings between DR and Graphical User Interface (GUI) display objects. As an initial step in this direction, we have implemented a Display Library similar to the RL, with a structured organisation of persistent display procedures that draw DR objects using specified (procedure) parameters. To specify a particular convention required by a new model, specific procedure parameters must be identified to replace the defaults, eg.

```
DEFINE DRAW ER
[
```

```
  DrawClass = DrawBox,
  DrawLink = DrawDiamond,
  DrawAttribute = DrawOval
]
```

Graphical display of data models has only been implemented for the ER model. More work on a generalised metasystem for GUIs is required.

Chapter 7

Experimentation

During the development of the Design Workbench a number of experiments were conducted, over and above ongoing validation tests, in order to assess the usability of the system. Firstly, we investigated the utility of the overall product to determine whether the benefits of using this automated environment were sufficient to warrant its adoption in practice. Thereafter, some experiments involving the use of several different models to design and view a single target system were undertaken. Finally, we experimented with the kernel of a change management system to support design evolution. This chapter discusses the three types of experiment in turn.

7.1 Testing System Utility

To assess the benefits of the system and the likelihood of designers using it in practice, software that was required for its own sake (a "real system") had to be developed using the automated environment. The "real system" chosen was the Design Workbench itself.

Its construction can be broadly divided into three phases. In the first, the prototype [FIJK94] was replaced by a toolkit with equivalent functionality, but able to support multiple co-existing models. Phase two integrated this within the Persistent Workshop, and the final phase introduced the metasystem. Naturally the utility of the system needed to be established as soon as possible; since phase one involved reorganising the repository it made a good candidate for such an experiment. A specification of the entire design system was written in the DAPLEX modelling language and used as input to itself, at a time when constraints and OMT were being added to its modelling capabilities.

The first effect was to force a model of the system to be fully specified. Beforehand this had been cursorily done; the discipline and precision required to construct a complete model provided greater clarity, particularly in terms of deciding when items were the

same and when not. Textual input proved very convenient, both in terms of initially creating the model and also for editing it.

The query subsystem was not used as much as expected, but this was probably because the target system (and its model) were too well known already, being a successor of existing software, and developed by an individual rather than a team. On the other hand, when others involved in a similar project [MO95] were asked to comment on the model and the target system it generated, they found it easier to learn and understand the design by using the query system rather than by studying the documentation/output. This suggests that the query facilities will be particularly useful to new members joining a design team, or to anyone needing to maintain the target system.

The code generation facility was extremely useful and saved a great deal of time because so much of the generated code was retained unchanged. In particular, it was very encouraging to see the high degree with which the generated types matched the types that had been planned independently of the SDE. Where there were differences, they were minor ones; the automated suggestions were sometimes replaced, but in other instances were adopted instead of our original ("manually designed") structures. The many **make** handlers were kept in the final system, and their existence greatly facilitated repository changes during subsequent phases of the Workbench development, since the places where code needed to be changed were localised. The other handlers were very useful in validating the system; this required less effort because of their availability.

In general, it was clear that the effort expended to input models was trifling in comparison with the effort saved by having code produced automatically. For this simple reason alone, the system is well worth using. In addition, the advantage of having the system suggest types and procedures that agree closely with her personal ideas, gives a designer much greater confidence in her system. The utility of the query subsystem was not fully evaluated; a team of designers working together on a new project would give greater insight into its benefits.

7.2 Supporting multiple models

Several experiments were carried out in order to study how multiple co-existing models could be used, and to show the advantages of such a system. The first such series involved creating entire designs using a single data model and then studying how the model, once in the data repository, can be viewed through other modelling languages. A second series of experiments involved constructing parts of a single design using different data models to specify the parts. The first two subsections describe the addition of another data model (SDM [HM81]) in the repository and how it is viewed in other modelling languages. The third subsection describes an experiment where parts of the design were specified in a number of modelling languages.

Some of the results of viewing a designed data model in other modelling languages have already been presented at the end of chapter 4. The most comprehensive example submitted to the Design Workbench was the tanker monitoring application environment from Hammer and McLeod [HM81]. This SDM application (shown in Appendix A.1) includes grouping classes, subclasses and class- and attribute-level aggregation. Once stored, the design was viewed via a number of models: ER, OMT, DAPLEX and SDM itself. As an illustration, Appendix A.1 contains the original SDM model specification and the resulting textual OMT and DAPLEX views of the repository representations; simple classes such as `DATE` and `PORT_NAMES` are not shown.

The first task required for this experiment was to define SDM using the metasystem. This meant that mappings between SDM and the Data Repository had to be identified. In the process, the notion of a property had to be considered. The properties of an object are the information on that real-world entity that are of interest in the computer model. These come in two forms: they can be printable values or they can be references to related objects. The former kind are represented as Attributes in the Data Repository and the latter as Links. In SDM there is no notion of relationship, only classes and attributes; and there is no distinction between the two kinds of property. When SDM was defined in the metasystem, printable attributes were mapped onto DR Attributes while non-printable (class-valued) attributes were mapped onto DR Links. An alternative approach would be to use the part-of/made-of facilities of the data repository, but then the attribute name would be lost. This could be problematic particularly if there was more than one attribute in a class having the same complex (class-valued) type.

7.2.1 Accommodating SDM in the data repository

SDM has a number of primitives which are not directly supported by the Data Repository. Some of these were catered for using existing DR fields; others were incorporated using the "`extra`" fields of DR structures. In the case of classes, these primitives are: class attributes (their value is a property of the class as a whole rather than individual instances), class-valued identifiers (eg one of the keys of Inspections is the Tanker attribute, which has Oil_Tankers as its value-class), groupings defined by predicate and subclasses defined by predicate or by union/intersection/difference. Attributes can be complex (ie. objects, not printable values), mandatory, not changeable, exhaustive of value class, nonoverlapping and/or derived.

The approach in accommodating these unsupported primitives is given below.

Class attributes (eg. `absolute_top_legal_speed`) have a single value for the entire class. These are incorporated in a new class comprising all such attributes for this class, named `classAttributes_`*className*. In the example presented here, the class attribute is a member of the class `classAttributes_SHIPS`.

Identifiers with non-printable components (eg. Tanker is part of the key for Inspections) do not appear as keys in the Data Repository, since such attributes are represented as Links. The "**extra**" field of the attribute's class has references to such non-printable identifiers.

Although the DR already has structures suitable for representing **subclass** and **grouping** definitions and attribute derivations by virtue of its support for constraints, in this experiment this was not exploited because the handlers do not take such features into account. It would be worthwhile to extend them in this way in a future version of the workbench. The remaining SDM-specific features were stored in the repository using the "**Extra**" Repository Library procedures. It would be easy to introduce new fields rather than use extensions fields; however the code generation facilities have not been extended to cater for these at present.

7.2.2 Viewing SDM-input designs in other models

When viewing the tanker specification from the viewpoint of other models, SDM-specific constructs incorporated as "**extra**" information can be lost; however, when model output is textual (eg DAPLEX and OMT) this is given as comments. Other differences that arise when viewing an SDM model through ER, DAPLEX or OMT reflect the different models' ways of seeing the same concept. For example, object aggregations appear as "separate" associations in an ER view of a design.

The major contribution of semantic data models like SDM is that they provide a variety of constructs for capturing the same information in different ways; their purpose is to make it as easy as possible to describe the real world and as difficult as possible for relevant details to be omitted or misrepresented [HM81]. Choosing a particular conceptual model to build/edit a developing system is of primary significance because it determines the versatility afforded the designer in this important task.

To illustrate, consider a many:many Employee:Employee relationship in an ER model representing co-workers in a Laboratory. In SDM, this could be given as a class Lab_Groups defined **either** as a grouping of Employee on common value of Assigned_Lab, **or** as a base class having a multivalued Employee attribute called eg co-workers (and there are still other possibilities!) As the Workbench Repository only supports the latter viewpoint, it will always present the information in this way, irrespective of how the designer specifies it. Using ER, Daplex or OMT models the former approach is not possible; the advantage of SDM is that it enables designers to describe Laboratory co-workers in whichever of these two ways is most natural or intuitive to them.

As another example, values which apply system-wide rather than to individual objects, are conveniently expressed by a designer working with SDM using the concept of a class attribute. Using a model where this notion is lacking (eg ER, OMT or Daplex) one or more additional classes/entity sets will have to be introduced so as to record

these properties. It would be up to the end-user to choose how this is done - they could use a single "system" class to store all these attributes, or could use the approach adopted in our experiment, or some other mechanism. The choice of model through which a design is viewed can aid or hamper understanding a design, depending on which constructs are directly represented in that model. Thus, class attributes are conveniently and simply dealt with in SDM, and are more intuitive than a separate classAttribute_Ships class.

Experiments such as that based on the tanker monitoring system showed that the view of a model can be changed once stored, but also that certain constructs are model-specific and cannot be cleanly represented in other modelling languages. This is generally a consequence of the power of the chosen model and not of some shortcoming in the Data Repository design. As a result of our experiments, the Design Workbench is now equipped to handle ER, Daplex, OMT, SDM, DFD and Structure Chart models. When additional models are included, the onus for ensuring smooth transitions from one model to another will rest with the metamodeller, who is responsible for specifying their mapping to the DR.

Figure 7.1 summarises the similarities of the data models implemented to date.

OMT	ER	FDM	SDM
class	entity	entity	class
link	relationship	function†	attribute•
attribute	attribute	function%	attribute#

% Daplex functions that return printable types are stored as attributes.
† Daplex functions that have more than one parameter are stored as links.
• Printable SDM attributes are stored as attributes.

SDM attributes that are classes are stored as links.

Figure 7.1: Similarities of data models implemented

7.2.3 Designing with different models

A second series of experiments was carried out to demonstrate the utility of multiple co-existing models. These investigated how different parts of a single design can be specified using different models. One such experiment used a parts-supplier schema with suppliers, retailers, and parts, and was designed using two models: Entity-Relationship and Functional (DAPLEX). The initial design was input using DAPLEX; this was subsequently extended using firstly an ER specification and then DAPLEX again.

Appendix A.2 contains the original textual DAPLEX (Figure A.4) and ER (Figure A.5) specifications and the resulting DAPLEX representation (Figure A.6). Figure A.7 is the subsequent DAPLEX specification and Figure A.8 is the DAPLEX representation of the final model in the data repository.

The original DAPLEX specification declares Part, Order, Supplier, Person, Employee and Project entities and a number of functions on these. The ER specification defines Subpart (using the DAPLEX-specified part Entity), Company, Manufacturer, Retailer and Car. The subsequent DAPLEX specification refers to objects defined in both previous specifications (eg. the Car entity from the ER design and the Project entity from the DAPLEX design).

7.2.4 Conclusion

In general the first set of experiments showed that, when a design is seen via a different model from that with which it was created, the result does not appear contrived, unnatural or complicated. Moreover, team members who prefer working with a particular model can benefit by seeing, in automatically added comments, information that would not otherwise have been available using their model.

From the second set of experiments we conclude that different parts of a single data/process design can be created and modified using different models over time, to the benefit of the development team. It permits all team members to design with their individual models of preference, rather than being forced to adopt a single standard model; while at the same time being able to use additional constructs/models if and when necessary, simply by switching to an appropriate model at that point. This flexibility makes for a more helpful and more usable Design Workbench.

7.3 Change Propagation

While experimenting with the Design Workbench it became apparent that support for design evolution would be highly beneficial. This would ideally enable users to alternate between changing their models and altering the generated target system, without concerning themselves with change propagation. In the absence of such a facility, the user who has edited generated code and now wishes to modify the system design must accept that doing this will cause all existing manual edits to be lost (overwritten).

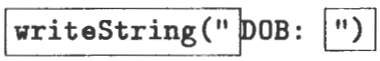
Some initial work was therefore undertaken to investigate the feasibility of an automated mechanism for propagating changes to affected objects. The modification of a data model object, for example, could affect documentation, other parts of the data model, processes (and their associated procedures) in the process model, type declarations and type handlers. As a first step in understanding the role of a change management system, it was decided to investigate preserving user edits to generated handlers.

The simplest solution is to replace all design objects affected by a change. However, since certain design objects are *generated* and *then edited* by users, any manual code changes would be lost. Below is an outline of some of the issues in designing a change absorption system.

As an illustration, the following change may be made to part of a handler that outputs student information:

The code unit: `writeString("DOB: ")` could be changed to
`writeString("Date of Birth: ")`

If any change to student causes straight-forward *re-generation* of its handlers, the above edit, which does not affect other parts of the system, would be lost. Change absorption mechanisms can be used to control and propagate changes through a design (both models and code). Such a system needs to distinguish between those parts of the design that cannot be changed and those that can. One solution is to divide the handler programs into cells, and distinguish user-modifiable cells. For the example code unit above, the following decomposition is created:



where only unboxed code is user-modifiable. The collection of cells still needs to be associated with the "DOB" attribute so that, if the attribute is deleted, the entire code unit disappears. We represent generated code by separate code units, differentiating system-created from user-created units; the latter represents text inserted between generated statements. A system unit is a collection of code cells, each flagged as user-modifiable or not. A prototype change absorption system for the **make**, **read** and **write** handlers has been implemented using this approach, and is intended to serve as a basis for change absorption throughout the system.

Chapter 8

Conclusion

This chapter summarises the thesis, discusses the contribution of persistence for CASE technology, answers the questions posed in the Introduction and presents some ideas for future work.

8.1 Thesis Summary

This thesis described a Software Development Environment for Napier88 which inputs data and process models and automatically generates the kernel of the target software system. All designed and implemented components are kept together on the persistent store, and can be queried or graphically viewed. The data repository was extended to handle EER, then FDM and finally OMT models and the system integrated in the Persistent Workshop as the Design Workbench. The workbench supports multiple co-existing models and a metasystem allows user-defined models to be specified. Thus programmers can use their preferred models, even when working in teams on a single design. The product permits applications to be developed more easily and quickly, and produces better systems, because code and documentation are generated automatically and according to prescribed conventions [ABPW94]. Its immediate benefits relieve programmers of tedious software construction tasks; its long-term advantages are improved quality, increased reliability and greater software reuse.

A metamodeller, consisting of two subsystems, was created to support the incorporation of new models in the repository by automating model input and model querying. A model compiler is used to generate a parser for the model; the metamodel-user specifies the mapping from model primitives to repository constructs and the generated parser checks and stores the designs accordingly. A powerful interpreter is generated from a metamodel allowing designs to be queried according to that model's view of repository constructs.

Multiple co-existing models in a unified data repository allow each member of a team of designers to use the modelling language they prefer, as well as other constructs explicitly available in other models, and to swop between model views of the repository.

The benefits for programmers include the provision of a CASE environment aimed at Napier88 programmers (helping with Napier88-specific programming tasks, such as preamble generation and stub creation). The generation of types, handlers and procedure stub hierarchies was found to be particularly useful in producing a prototype system quickly. Querying the repository provided a means for new programmers to discover the models designed and the systems implemented. The dependency query can show the possible impact of user changes to the system. Some initial work has also been done on a change absorption facility that propagates design / code changes through the models and target system implementation.

8.2 Discussion

The primary goal of the thesis was to explore the utility of orthogonal persistence for Computer-Aided Software Engineering (CASE). The product was also intended to provide useful technology for POS system designers who currently do not use automated design aids.

This thesis has demonstrated the utility of orthogonally persistent systems for CASE, by exploiting the benefits they provide to build a sophisticated systems design workbench. The workbench implementation was enormously facilitated because POS simplifies the programming model for persistent applications and provides constructs that are highly suited to complex data and transactions.

Orthogonal persistence simplifies the programming task for persistent application systems such as CASE by not distinguishing between code that uses persistent data and code that does not, by allowing data of any type to be made persistent (such as procedures, design models and graphical objects) and by ensuring that the persistence property of an object does not depend on the type of the object. Referential integrity is automatically preserved, ensuring that these application systems are more reliable. A persistent CASE system can store the entire range of system construction data from designs to the generated system, in the same stable store. The CASE data structures include small-sized data, such as the multiplicity of a design object, and large-sized data, such as graphical images and procedure hierarchies. Persistent programming languages support atomic updates to data; this is essential since we must ensure that the valuable software design data is safe in the event of a system crash.

The orthogonally persistent programming language Napier88 has a rich type set that supports the complex data types required by an SDE (such as design data and graphical data types for model viewing), linguistic reflection allows generated types and procedures to be compiled and executed at run-time, the union type **any** allows data

stored by the SDE to be extended (without changing existing data) and, together with linguistic reflection, allows the functionality of the SDE to be extended (such as with the incorporation of query engines in the metamodeller). Maps provided an efficient way to store model objects in the Data Repository and within the systems developed using the SDE.

We end this discussion by returning to the questions posed in the Introduction of this thesis:

- *are persistent languages convenient for building CASE Environments? If so, in what ways?*

  Orthogonal persistence and persistent programming languages in particular provide a programming model, a transaction model and a type system that makes implementing CASE systems easier. A CASE developer does not have to code data transfers between a database and memory, or maintain different versions of data (one for persistent and another for transient data) or be restricted by the type of data that may be made persistent.

- *what features of persistent languages are useful for CASE and what problems exist?*

  Napier88 provides a rich type system and linguistic reflection that make it easier to build a CASE system. The type system is sufficiently rich to represent all CASE data, and the callable compiler allows the types and programs to be interpreted and then stored in the repository. Linguistic reflection was also used to interpret end-user where-clauses in the metamodeller.

- *what trade-offs exist in persistent SE?*

  Orthogonally persistent object systems conform to a closed world model to ensure that referential integrity is enforced. For a CASE-builder, this means that tools available in conventional systems (such as Unix tools like yacc) need to be ported to the orthogonally persistent world. A CASE end-user also cannot use common editing tools, configuration tools, etc. available to other programming languages.

  Napier88 programs run far slower than, for example, equivalent C programs; this can be discouraging for a user of a design application such as CASE where reaction time is important.

- *can a persistent CASE system extend the boundaries of CASE technology, and if so, how?*

  The support of multiple co-existing models by the Design Workbench presents a unique environment where parts of a design can be described in a number of modelling languages within a single Data Repository. The metamodeller provides a mechanism through which new models can be integrated into the Design

Workbench; its implementation has shown how persistence opens up opportunities for building extensible systems.

– *will persistent programmers use CASE products and what tools would they find most beneficial?*

  Rapid prototyping of target systems and the automation of programming language tasks (such as environment hierarchy and stub generation) were identified as most useful to persistent programmers. Programmers new to the designed system found the query system useful in understanding the design. Experiments performed on the system confirmed that these were beneficial to persistent programmers.

8.3 Future work

The two main aspects of the Design Workbench that require further work are the change absorption facility and the specification of graphical display of new models in the metasystem. The latter in particular is little more than an idea at present; its feasibility needs to be demonstrated by constructing a fully operational "metamodel-display" system. On the other hand sufficient change management code has been written to show that some form of support for evolution is definitely possible; this needs to be expanded considerably to test the limits of our approach.

The specification of SDM via the metasystem demonstrated that there were constructs lacking in the data repository. Some of these could be added to the Data Repository; the Type and Handler Generators would need to be extended in line with this. A mechanism for specifying the mapping from a new modelling primitive to its associated type and handler generation activities is also worth investigating.

The Design Workbench generates types, handlers, stubs and environment hierarchies for Napier88 programs. This could be extended to produce code for other persistent programming languages such as DBPL [SM91] and Fibonacci [AGO95], but the system would not be able to store the object code for these languages since Napier88 only supports a Napier88 compiler.

Finally, there is scope for extending the system in new directions, particularly in the areas of software visualisation and version management. Some work in this area is being done by [Lav94].

8.4 Conclusion

We believe that this project has clearly demonstrated the benefits of persistent object systems for computer-aided software design. It has demonstrated the utility and the

need for persistent programming languages such as Napier88 and shown how persistent systems can be exploited to provide better software engineering environments than are supported by conventional databases.

Appendix A

Metasystem experiments

A.1 Viewing in a different model

This section lists a tanker application that was input to the data repository using the SDM model given in Figure A.1. The OMT and Daplex representations of the resulting data repository are shown Figure A.2 and Figure A.3 respectively.

Figure A.1: The tanker monitoring application SDM model

```
SDM Tankers

!--forward references
ASSIGNMENTS
END

ENGINES
END

INCIDENTS
END

OFFICERS
END

OIL_TANKERS
END
!--

SHIPS
 description: "all ships with potentially hazardous cargoes that may enter the U.S. coas
tal waters"
 duplicates not allowed
 member attributes:
  Name                    value class: SHIP_NAMES
  Hull_number             value class: HULL_NUMBERS
                          may not be null
                          not changeable
  Type                    description: "the kind of ship, for example, merchant or fishing
"
                          value class: SHIP_TYPE_NAMES
  Country_of_registry     value class: COUNTRIES
                          inverse: Ships_registered_here
  Name_of_home_port       value class: PORT_NAMES
  Cargo_types             description: "the type(s) of cargo the ship can carry"
                          value class: CARGO_TYPE_NAMES
                          multivalued
  Captain                 description: "the current captain of the ship"
                          value class: OFFICERS
                          match: Officer of ASSIGNMENTS on Ship
  Engines                 value class: ENGINES
                          multivalued with size between 0 and 10
                          exhausts value class
                          no overlap in values
  Incidents_involved_in   value class: INCIDENTS
                          inverse: Involved_ship
                          multivalued

 identifiers:             Name Hull_number
END

INSPECTIONS
 description: "inspections of oil tankers"
 member attributes:
  Tanker                  description: "the tanker inspected"
                          value class: OIL_TANKERS
                          inverse: Inspections
  Date                    value class: DATES
  Order_for_tanker        description: "the ordering of the inspections for a tanker with
the most recent inspection having_value 1"
                          value class: INTEGERS
                          derivation: order by decreasing Date within Tanker

 class attributes:
  Number                  description: "the number of inspections in the database"
                          value class: INTEGERS
                          derivation: number of members in this class

 identifiers:             Tanker + Date
END

OFFICERS
 description: "all certified officers of ships"
 member attributes:
  Name                    value class: PERSON_NAMES
  Country_of_licence      value class: COUNTRIES
  Date_commissioned       value class: DATES
  Seniority               value class: INTEGERS
                          derivation: order by Date_commissioned
  Commander               description: "the officer in direct command of this officer"
                          value class: OFFICERS
  Superiors               value class: OFFICERS
                          derivation: all levels of values of Commander
                          inverse: Subordinates
                          multivalued
  Subordinates            value class: OFFICERS
                          inverse: Superiors
                          multivalued
  Contacts                value class: OFFICERS
                          derivation: where is in Superiors or is in Subordinates

 identifiers:             Name
END

ENGINES
 description: "ship engines"
 member attributes:
  Serial_number           value class: ENGINE_SERIAL_NUMBERS
  Kind_of_engine          value class: ENGINE_TYPE_NAMES

 identifiers:             Serial_number
END

INCIDENTS
 description: "accidents involving ships"
 member attributes:
  Involved_ship           value class: SHIPS
                          inverse: Incidents_involved_in
  Date                    value class: DATES
  Description             description: "textual explanation of the accident"
                          value class: INCIDENT_DESCRIPTIONS
  Involved_captain        value class: OFFICERS

 identifiers:             Involved_ship + Date + Description
END

ASSIGNMENTS
 description: "assignments of captains to ships"
 member attributes:
  Officer                 value class: OFFICERS
  Ship                    value class: SHIPS

 identifiers:             Officer + Ship
END

OIL_TANKERS
 description: "oil-carrying ships"
 interclass connection: subclass of SHIPS where Cargo_types contains "oil"
 member attributes:
  Hull_type               description: "specification of single or double hull"
                          value class: HULL_TYPE_NAMES
  Is_tanker_banned        value class: bool
                          derivation: if in BANNED_SHIPS
  Inspections             value class: INSPECTIONS
                          inverse: Tanker
                          multivalued
```

Figure A.1: The tanker monitoring application SDM model

```
  Number_of_times_inspected
                        value class: int
                        derivation: number of unique members in Inspections
  Last_inspection       value class: MOST_RECENT_INSPECTIONS
                        inverse: Tanker
  Last_two_inspections  value class: INSPECTIONS
                        derivation: subvalue of Inspections where Order_for_tankers <= 2
                        multivalued
  Date_last_examined    value class: DATES
                        derivation: same as Last_inspection.Date
  Oil_spills_involved_in
                        value class: INCIDENTS
                        derivation: subvalue of Incidents_involved_in where is in OIL_SP
ILLS
                        multivalued

 class attributes:
  Absolute_top_legal_speed
                        value class: KNOTS
  Top_legal_speed_in_miles_per_hour
                        derivation: = Absolute_top_legal_speed / 1.1
                        value class: MILES_PER_HOUR
END

RURITANIAN_SHIPS
 description: ""
 interclass connection: subclass of SHIPS where Country.Name = "Ruritania"
END

RURITANIAN_OIL_TANKERS
 description: ""
 interclass connection: subclass of OIL_TANKERS where Country.Name = "Ruritania"
END

MERCHANT_SHIPS
 description: ""
 interclass connection: subclass of SHIPS where Type = "merchant"
 member attributes:
  Cargo_types:          value class: MERCHANT_CARGO_TYPE_NAMES
END

OIL_SPILLS
 description: ""
 interclass connection: subclass of INCIDENTS where Description = "oil spill"
 member attributes:
  Amount_spilled        value class: GALLONS
  Severity              value class: real
                        derivation: = Amout_spilled/100000
 class attributes:
  Total_spilled         value class: GALLONS
                        derivation: sum of Amount_spilled over members of this class
END

MOST_RECENT_INSPECTIONS
 description: ""
 interclass connection: subclass of INSPECTIONS where Order_for_tanker = 1
END

DANGEROUS_CAPTAINS
 description: "captains who have been involved in an accident"
 interclass connection: subclass of OFFICERS where is a value of Involved_captain of INC
IDENTS
END

SHIP_TYPES
 description: "types of ships"
 interclass connection: grouping of SHIPS on common value of Type
  groups defined as classes are MERCHANT_SHIPS
 member attributes:
  Instances             description: "the instances of the type of ship"
                        value class: SHIPS
                        derivation: same as Contents
                        multivalued
  Number_of_ships_of_this_type
                        value class: int
                        derivation: number of members in Contents
END

CARGO_TYPE_GROUPS
 description: ""
 interclass connection: grouping of SHIPS on common value of Cargo_types
END

TYPES_OF_HAZARDOUS_SHIPS
 description: ""
 interclass connection: grouping of SHIPS consisting of classes BANNED_SHIPS, BANNED_OIL
_TANKERS, SHIPS_TO_BE_MONITORED
END

CONVOYS
 description: ""
 interclass connection: groups of SHIPS as specified
 member attributes:
  Oil_tanker_constituents
                        description: "the oil tankers that are in the convoy (if any)"
                        value class: SHIPS
                        derivation: subvalue of Contents where is in OIL_TANKERS
                        multivalued
END
```

Figure A.2: The tanker monitoring application OMT model

```
CLASS ASSIGNMENTS
 DESCRIPTION "assignments of captains to ships"
 !composite key Officer and Ship
END CLASS

CLASS BANNED_OIL_TANKERS
END CLASS

CLASS CARGO_TYPE_GROUPS MADE OF SHIPS !on common value of Cargo_types
END CLASS

CLASS CONVOYS MADE OF SHIPS !as specified
END CLASS

CLASS DANGEROUS_CAPTAINS ISA OFFICERS !where is a value of Involved_captain of INCIDENTS
 DESCRIPTION "captains who have been involved in an accident"
END CLASS

CLASS ENGINES
 DESCRIPTION "ship engines"
 ATTRIBUTES Serial_number:ENGINE_SERIAL_NUMBERS  KEY
            Kind_of_engine:ENGINE_TYPE_NAMES
END CLASS

CLASS INCIDENTS
 DESCRIPTION "accidents involving ships"
 ATTRIBUTES Description:INCIDENT_DESCRIPTIONS !composite KEY with Date and Involved_ship
             DESCRIPTION "textual explanation of the accident"
            Date:DATES !composite KEY with Description and Involved_ship
END CLASS

CLASS ClassAttributes_INSPECTIONS
 ATTRIBUTES Number:INTEGERS
             DESCRIPTION "the number of inspections in the database"
END CLASS

CLASS INSPECTIONS
 DESCRIPTION "inspections of oil tankers"
 ATTRIBUTES Date:DATES !composite KEY with Tanker
            Order_for_tanker:INTEGERS
             DESCRIPTION "the ordering of the inspections for a tanker with the most rece
nt inspection having_value 1"
END CLASS

CLASS MERCHANT_SHIPS ISA SHIPS !where Type = "merchant"
 ATTRIBUTES Cargo_types:MERCHANT_CARGO_TYPE_NAMES
END CLASS

CLASS MOST_RECENT_INSPECTIONS ISA INSPECTIONS !where Order_for_tanker = 1
END CLASS

CLASS OFFICERS
 DESCRIPTION "all certified officers of ships"
 ATTRIBUTES Name:PERSON_NAMES  KEY
            Date_commissioned:DATES
            Seniority:INTEGERS
END CLASS

CLASS ClassAttributes_OIL_SPILLS
 ATTRIBUTES Total_spilled:GALLONS
END CLASS

CLASS OIL_SPILLS ISA INCIDENTS !where Description = "oil spill"
 ATTRIBUTES Amount_spilled:GALLONS
            Severity:real
END CLASS

CLASS ClassAttributes_OIL_TANKERS
 ATTRIBUTES Absolute_top_legal_speed:KNOTS
            Top_legal_speed_in_miles_per_hour:MILES_PER_HOUR
END CLASS

CLASS OIL_TANKERS ISA SHIPS !where Cargo_types contains "oil"
 DESCRIPTION "oil-carrying ships"
 ATTRIBUTES Hull_type:HULL_TYPE_NAMES
             DESCRIPTION "specification of single or double hull"
            Is_tanker_banned:bool
            Number_of_times_inspected:int
            Last_inspection:MOST_RECENT_INSPECTIONS
            Date_last_examined:DATES
END CLASS

CLASS RURITANIAN_OIL_TANKERS ISA OIL_TANKERS !where Country.Name = "Ruritania"
END CLASS

CLASS RURITANIAN_SHIPS ISA SHIPS !where Country.Name = "Ruritania"
END CLASS

CLASS SHIPS PART OF SHIP_TYPES
            PART OF TYPES_OF_HAZARDOUS_SHIPS
            PART OF CONVOYS
            PART OF CARGO_TYPE_GROUPS
 DESCRIPTION "all ships with potentially hazardous cargoes that may enter the U.S. coast
al waters"
 ATTRIBUTES Hull_number:HULL_NUMBERS  KEY
            Name:SHIP_NAMES  KEY
            Type:SHIP_TYPE_NAMES
             DESCRIPTION "the kind of ship, for example, merchant or fishing"
            Name_of_home_port:PORT_NAMES
            Cargo_types:*CARGO_TYPE_NAMES
             DESCRIPTION "the type(s) of cargo the ship can carry"
END CLASS

CLASS SHIPS_TO_BE_MONITORED
END CLASS

CLASS SHIP_TYPES MADE OF SHIPS !on common value of Type groups defined as classes are ME
RCHANT_SHIPS
 DESCRIPTION: "types of ships"
 ATTRIBUTES Number_of_ships_of_this_type:int
END CLASS

CLASS TYPES_OF_HAZARDOUS_SHIPS MADE OF SHIPS !consisting of classes BANNED_SHIPS, BANNED
_OIL_TANKERS, SHIPS_TO_BE_MONITORED
END CLASS

LINK ASSIGNMENTS-OFFICERS
 ASSIGNMENTS:M TO OFFICERS:1 ROLE Officer
END LINK

LINK ASSIGNMENTS-SHIPS
 ASSIGNMENTS:M TO SHIPS:1 ROLE Ship
END LINK

LINK INCIDENTS-OFFICERS
 INCIDENTS:M TO OFFICERS:1 ROLE Involved_captain
END LINK

LINK INCIDENTS-SHIPS
 INCIDENTS:M TO SHIPS:1 ROLE Involved_ship !composite KEY of INCIDENTS with Date and Des
cription
END LINK

LINK INSPECTIONS-OIL_TANKERS
        INSPECTIONS:M ROLE Inspections TO
        OIL_TANKERS:1 ROLE Tanker DESCRIPTION "the tanker inspected"
```

Figure A.2: The tanker monitoring application OMT model

```
END LINK

LINK OFFICERS-COUNTRIES
        OFFICERS:M TO
        COUNTRIES:1 ROLE Country_of_licence
END LINK

LINK OFFICERS-OFFICERS1
        OFFICERS:M TO
        OFFICERS:1 ROLE Commander DESCRIPTION "the officer in direct command of this of
ficer"
END LINK

LINK OFFICERS-OFFICERS2
        OFFICERS:M ROLE Subordinates
        TO OFFICERS:M ROLE Superior
END LINK

LINK OFFICERS-OFFICERS3
        OFFICERS:M TO
        OFFICERS:1 ROLE Contacts
END LINK

LINK OIL_TANKERS-INCIDENTS
        OIL_TANKERS:M TO
        INCIDENTS:M ROLE Oil_spills_involved_in
END LINK

LINK OIL_TANKERS-INSPECTIONS1
        OIL_TANKERS:M TO
        INSPECTIONS:M ROLE Last_two_inspections
END LINK

LINK SHIPS-COUNTRIES
        SHIPS:M ROLE Ships_registered_here TO
        COUNTRIES:1 ROLE Country_of_registry
END LINK

LINK SHIPS-ENGINES
        SHIPS:M TO
        ENGINES:M ROLE Engines
END LINK

LINK SHIPS-INCIDENTS
        SHIPS:M ROLE Involved_ship TO
        INCIDENTS:M ROLE Incidents_involved_in
END LINK

LINK SHIPS-OFFICERS
        SHIPS:M TO
        OFFICERS:1 ROLE Captain DESCRIPTION "the current captain of the ship"
END LINK

LINK SHIPS-SHIPS
        SHIP_TYPES:M TO
        SHIPS:M ROLE Instances DESCRIPTION "the instances of the type of ship"
END LINK

LINK CONVOY-SHIPS
        CONVOY:M TO
        SHIPS:M ROLE Oil_tanker_constituents DESCRIPTION "the oil tankers that are in th
e convoy (if any)"
END LINK
```

Figure A.3: The tanker monitoring application FDM model

```
DECLARE ASSIGNMENTS() -> ENTITY
!composite KEY Officer and Ship
DECLARE BANNED_OIL_TANKERS() -> ENTITY
DECLARE CARGO_TYPE_GROUPS() -> ENTITY
!made of Ships on common value of Cargo_types
DECLARE CONVOYS() -> ENTITY
!made of Ships as specified
DECLARE COUNTRIES() -> ENTITY
DECLARE DANGEROUS_CAPTAINS() -> OFFICERS !where is a value of Involved_captain of INCIDE
NTS
DECLARE ENGINES() -> ENTITY
DECLARE INCIDENTS() -> ENTITY
DECLARE ClassAttributes_INSPECTIONS -> ENTITY
DECLARE INSPECTIONS() -> ENTITY
DECLARE MERCHANT_SHIPS() -> SHIPS !where Type = "merchant"
DECLARE MOST_RECENT_INSPECTIONS() -> INSPECTIONS !where Order_for_tanker = 1
DECLARE OFFICERS() -> ENTITY
DECLARE ClassAttributes_OIL_SPILLS -> ENTITY
DECLARE OIL_SPILLS() -> INCIDENTS !where Description = "oil spill"
DECLARE ClassAttributes_OIL_TANKERS -> ENTITY
DECLARE OIL_TANKERS() -> SHIPS !where Cargo_types contains "oil"
DECLARE RURITANIAN_OIL_TANKERS() -> OIL_TANKERS !where Country.Name = "Ruritania"
DECLARE RURITANIAN_SHIPS() -> SHIPS !where Country.Name = "Ruritania"
DECLARE SHIPS() -> ENTITY
!part of SHIP_TYPES
!part of TYPES_OF_HAZARDOUS_SHIPS
!part of CONVOYS
!part of CARGO_TYPE_GROUPS
DECLARE SHIPS_TO_BE_MONITORED() -> ENTITY
DECLARE SHIP_TYPES() -> ENTITY
!made of SHIPS on common value of Type groups defined as classes are MERCHANT_SHIPS
DECLARE TYPES_OF_HAZARDOUS_SHIPS() -> ENTITY
!made of SHIPS consisting of classes BANNED_SHIPS, BANNED_OIL_TANKERS, SHIPS_TO_BE_MONIT
ORED

! ENGINES attributes
DECLARE Serial_number(ENGINES) -> ENGINE_SERIAL_NUMBERS
DEFINE ENGINES(ENGINE_SERIAL_NUMBERS) -> INVERSE OF Serial_number(ENGINES)
DECLARE Kind_of_engine(ENGINES) -> ENGINE_TYPE_NAMES

! INCIDENTS attributes
DECLARE Date(INCIDENTS) -> DATES
!composite KEY with Description and Involved_ship
DECLARE Description(INCIDENTS) -> INCIDENT_DESCRIPTIONS
!composite KEY with Date and Involved_ship

! ClassAttributes_INSPECTIONS attributes
DECLARE Number(ClassAttributes_INSPECTIONS) -> INTEGERS

! INSPECTIONS attributes
DECLARE Date(INSPECTIONS) -> DATES
!composite KEY with Tanker
DECLARE Order_for_tanker(INSPECTIONS) -> INTEGERS

! MERCHANT_SHIPS attributes
DECLARE Cargo_types(MERCHANT_SHIPS) -> MERCHANT_CARGO_TYPE_NAMES

! OFFICERS attributes
DECLARE Name(OFFICERS) -> PERSON_NAMES
DEFINE OFFICERS(PERSON_NAMES) -> INVERSE OF Name(OFFICERS)
DECLARE Date_commissioned(OFFICERS) -> DATES
DECLARE Seniority(OFFICERS) -> INTEGERS

! ClassAttributes_OIL_SPILLS attributes
DECLARE Total_spilled(ClassAttributes_OIL_SPILLS) -> GALLONS

! OIL_SPILLS attributes
DECLARE Amount_spilled(OIL_SPILLS) -> GALLONS
DECLARE Severity(OIL_SPILLS) -> real

! ClassAttributes_OIL_TANKERS attributes
DECLARE Absolute_top_legal_speed(ClassAttributes_OIL_TANKERS) -> KNOTS
DECLARE Top_legal_speed_in_miles_per_hour(ClassAttributes_OIL_TANKERS) -> MILES_PER_HOUR

! OIL_TANKERS attributes
DECLARE Hull_type(OIL_TANKERS) -> HULL_TYPE_NAMES
DECLARE Is_tanker_banned(OIL_TANKERS) -> bool
DECLARE Number_of_times_inspected(OIL_TANKERS) -> int
DECLARE Last_inspection(OIL_TANKERS) -> MOST_RECENT_INSPECTIONS
DECLARE Date_last_examined(OIL_TANKERS) -> DATES

! SHIPS attributes
DECLARE Name(SHIPS) -> SHIP_NAMES
DEFINE SHIPS(SHIP_NAMES) -> INVERSE OF Name(SHIPS)
DECLARE Hull_number(SHIPS) -> HULL_NUMBERS
DEFINE SHIPS(HULL_NUMBERS) -> INVERSE OF Hull_number(SHIPS)
DECLARE Type(SHIPS) -> SHIP_TYPE_NAMES
DECLARE Name_of_home_port(SHIPS) -> PORT_NAMES
DECLARE Cargo_types(SHIPS) ->> CARGO_TYPE_NAMES

! SHIP_TYPES attributes
DECLARE Number_of_ships_of_this_type(SHIP_TYPES) -> int

DECLARE Officer(ASSIGNMENTS) -> OFFICERS
DECLARE Ship(ASSIGNMENTS) -> SHIPS

DECLARE Involved_captian(INCIDENTS) -> OFFICERS
DECLARE Involved_ship(INCIDENTS) -> SHIPS
!composite KEY of INCIDENTS with Date and Description

DECLARE Tanker(INSPECTIONS) -> OIL_TANKERS
DEFINE Inspections(OIL_TANKERS) ->> INVERSE OF Tanker(INSPECTIONS)

DECLARE Country_of_licence(OFFICERS) -> COUNTRIES
DECLARE Commander(OFFICERS) -> OFFICERS
DECLARE Superior(OFFICERS) -> OFFICERS
DEFINE Subordinates(OFFICERS) ->> INVERSE OF Superior(OFFICERS)
DEFINE Contacts(OFFICERS) -> OFFICERS

DECLARE Oil_spills_involved_in(OIL_TANKERS) ->> INCIDENTS
DECLARE Last_two_inspections(OIL_TANKERS) ->> INSPECTIONS

DECLARE Country_of_registry(SHIPS) -> COUNTRIES
DEFINE Ships_registered_here(COUNTRIES) -> INVERSE OF Country_of_registry(SHIPS)
DECLARE Engines(SHIPS) ->> ENGINES
DECLARE Incidents_involved_in(SHIPS) ->> INCIDENTS
DEFINE Involved_ship(INCIDENTS) ->> INVERSE OF SHIP
DECLARE Captain(SHIPS) -> OFFICERS

DECLARE Instances(SHIP_TYPES) ->> SHIPS

DECLARE Oil_tanker_constituents(CONVOY) ->> SHIPS
```

A.2 Using alternative models to specify a model

A set of experiments was performed to show how parts of a single design can be specified in different modelling languages. The experiment listed in this section is a part-supplier schema; the initial modelling language was Daplex. A subsequent addition to the design was specified using the ER model, and the final specification was in Daplex. Figures A.4, A.5 and A.7 list these three specifications respectively. Figures A.6 and A.8 show a Daplex view of the repository after each stage.

Figure A.4 Initial DAPLEX specification of part of part-supplier schema

```
FDM part-supp

DECLARE Part() -> ENTITY
DECLARE name(Part) -> STRING
DECLARE category(Part) -> STRING
DECLARE quantity(Part) -> STRING

DECLARE Order() -> ENTITY
DECLARE number(Order) -> STRING
DECLARE date(Order) -> STRING
DECLARE quantity(Order) -> STRING

DECLARE Supplier() -> ENTITY

DECLARE Person() -> ENTITY
DECLARE name(Person) -> STRING
DECLARE address(Person) -> STRING

DECLARE Employee() -> Person
DECLARE salary(Employee) -> STRING
DECLARE position(Employee) -> STRING
DECLARE id_number(Employee) -> STRING
DECLARE joined(Employee) -> STRING

DECLARE Project() -> ENTITY
DECLARE aim(Project) -> STRING
DECLARE number(Project) -> STRING

DECLARE suppliedBy(Part) ->> Supplier
DECLARE units(Part,Supplier) -> STRING
DECLARE minimum_Order(Part,Supplier) -> STRING
DEFINE hasParts(Supplier) ->> INVERSE OF suppliedBy(Part)

DECLARE orderedFrom(Order) -> Supplier
DECLARE partsOrdered(Order) ->> Part
DECLARE quantity(Part,Order) -> STRING

DECLARE manages(Employee) ->> Employee
DEFINE managedBy(Employee) -> INVERSE OF manages(Employee)

DECLARE child(Employee) ->> Person
DEFINE children(Person) ->> INVERSE OF child(Employee)

DECLARE members(Project) ->> Employee
DEFINE worksOn(Employee) ->> INVERSE OF members(Project)

DECLARE leader(Project) -> Employee
DEFINE leads(Employee) -> INVERSE OF leader(Project)
END
```

Figure A.5 ER specification of part of part-supplier schema

```
ER part-supp

Company -- [name] address.
Manufacturer --.
Retailer --.
Car --.
SubPart --.

Manufacturer ISA Company.
Retailer ISA Company.
SubPart ISA Part.

END
```

Figure A.6 DAPLEX output of combined DAPLEX and ER part-supplier schema

```
DECLARE Car() -> ENTITY
DECLARE Company() -> ENTITY
DECLARE Employee() -> Person
DECLARE Manufacturer() -> Company
DECLARE Order() -> ENTITY
DECLARE Part() -> ENTITY
DECLARE Person() -> ENTITY
DECLARE Project() -> ENTITY
DECLARE Retailer() -> Company
DECLARE SubPart() -> Part
DECLARE Supplier() -> ENTITY

! Company attributes
DECLARE name(Company) -> string
DEFINE Company(string) -> INVERSE OF name(Company)
DECLARE address(Company) -> string
! Employee attributes
DECLARE salary(Employee) -> STRING
DECLARE position(Employee) -> STRING
DECLARE id_number(Employee) -> STRING
DECLARE joined(Employee) -> STRING
! Order attributes
DECLARE number(Order) -> STRING
DECLARE date(Order) -> STRING
DECLARE quantity(Order) -> STRING
! Part attributes
DECLARE name(Part) -> STRING
DECLARE category(Part) -> STRING
DECLARE quantity(Part) -> STRING
! Person attributes
DECLARE name(Person) -> STRING
DECLARE address(Person) -> STRING
! Project attributes
DECLARE aim(Project) -> STRING
DECLARE number(Project) -> STRING

DECLARE children(Person) ->> Employee
DEFINE child(Employee) ->> INVERSE OF children(Person)
DECLARE managedBy(Employee) ->> Employee
DEFINE manages(Employee) ->> INVERSE OF managedBy(Employee)
DECLARE orderedFrom(Order) -> Supplier
DECLARE partsOrdered(Order) ->> Part
DECLARE hasParts(Supplier) ->> Part
DEFINE suppliedBy(Part) ->> INVERSE OF hasParts(Supplier)
DECLARE leads(Employee) ->> Project
DEFINE leader(Project) ->> INVERSE OF leads(Employee)
DECLARE worksOn(Employee) ->> Project
DEFINE members(Project) ->> INVERSE OF worksOn(Employee)
DECLARE minimum_Order(Supplier,Part) -> STRING
DECLARE quanity(Order,Part) -> STRING
DECLARE units(Supplier,Part) -> STRING
```

Figure A.7 Subsequent DAPLEX specification of part of schema

```
FDM Parts_Supplier

DECLARE Alloc(Car) -> Project
DECLARE Client(Project) -> Company
DECLARE Date(Car,Project) -> string
DECLARE Fee(Project, Client(Project) ) -> string
END
```

Figure A.8 DAPLEX output of combined DAPLEX, ER and DAPLEX schema

```
DECLARE Car() -> ENTITY
DECLARE Company() -> ENTITY
DECLARE Employee() -> Person
DECLARE Manufacturer() -> Company
DECLARE Order() -> ENTITY
DECLARE Part() -> ENTITY
DECLARE Person() -> ENTITY
DECLARE Project() -> ENTITY
DECLARE Retailer() -> Company
DECLARE SubPart() -> Part
DECLARE Supplier() -> ENTITY

! Company attributes
DECLARE name(Company) -> string
DEFINE Company(string) -> INVERSE OF name(Company)
DECLARE address(Company) -> string
! Employee attributes
DECLARE salary(Employee) -> STRING
DECLARE position(Employee) -> STRING
DECLARE id_number(Employee) -> STRING
DECLARE joined(Employee) -> STRING
! Order attributes
DECLARE number(Order) -> STRING
DECLARE date(Order) -> STRING
DECLARE quantity(Order) -> STRING
! Part attributes
DECLARE name(Part) -> STRING
DECLARE category(Part) -> STRING
DECLARE quantity(Part) -> STRING
! Person attributes
DECLARE name(Person) -> STRING
DECLARE address(Person) -> STRING
! Project attributes
DECLARE aim(Project) -> STRING
DECLARE number(Project) -> STRING

DECLARE Alloc(Car) ->> Project
DECLARE Date(Car,Project) -> string
DECLARE children(Person) ->> Employee
DEFINE child(Employee) ->> INVERSE OF children(Person)
DECLARE managedBy(Employee) ->> Employee
DEFINE manages(Employee) ->> INVERSE OF managedBy(Employee)
DECLARE orderedFrom(Order) -> Supplier
DECLARE partsOrdered(Order) ->> Part
DECLARE hasParts(Supplier) ->> Part
DEFINE suppliedBy(Part) ->> INVERSE OF hasParts(Supplier)
DECLARE Client(Project) ->> Company
DEFINE Fee(Project,Client(Project)) -> string
DECLARE leads(Employee) ->> Project
DEFINE leader(Project) ->> INVERSE OF leads(Employee)
DECLARE worksOn(Employee) ->> Project
DEFINE members(Project) ->> INVERSE OF worksOn(Employee)
DECLARE minimum_Order(Supplier,Part) -> STRING
DECLARE quanity(Order,Part) -> STRING
DECLARE units(Supplier,Part) -> STRING
```

Bibliography

[ABC+93] M. Atkinson, P. Bailey, C. Christie, K. Cropper, and P. Philbrow. Towards Bulk Type Libraries for Napier88. Technical report, Dept. of Computer Science, University of Glasgow, 1993.

[ABPW94] Malcolm Atkinson, Pete Bailey, Paul Philbrow, and Ray Welland. *An Organization for Napier88 Libraries*. Technical Report 94-77, FIDE, 1994.

[Abr74] J. R. Abrial. Data Semantics. *Data Base Management*, pages 1–59, 1974.

[AGO95] A. Albano, G. Ghelli, and R. Orsini. An Introduction to Fibonacci: A Programming Language for Object Databases. Technical Report FIDE/95/120, FIDE, 1995.

[AH87] S. Abiteboul and R. Hull. IFO: A Formal Semantic Database Model. *ACM Transactions on Database Systems*, 12(4):525–565, December 1987.

[ALPR91] M. Atkinson, C. Lécluse, P. Philbrow, and P. Richard. Maps as Bulk Types for Data Base Programming Languages. Technical Report FIDE/91/24, FIDE ESPRIT BRA Project 3070, 1991.

[AM95] Malcolm Atkinson and Ronald Morrison. Orthogonal Persistent Object Systems. *VLDB Journal*, 4(3), August 1995.

[Atk93] Malcom Atkinson. Lecture Notes on Programming in Napier88, Part 1, April 1993. University of Glasgow, Department of Computer Science.

[BF96] Sonia Berman and Ricardo Figueira. Automating Software Design in a Persistent Environment. *Seventh International Workshop on Persistent Object Systems*, 1996.

[Boe88] B.W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21:61–72, May 1988. (Chapter 3).

[Bro87] F.P. Brookes. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, 20:10–19, April 1987.

[Che76] P.P.S. Chen. The entity-relationship model - Toward a unified view of data. *ACM Transactions on Database Systems 1*, 1:9–36, March 1976.

[Coo90] R. Cooper. Configurable data modelling systems. In *Proceedings of the 9th International Conference on the Entity Relationship Approach*, pages 35–52, Lausanne, Switzerland, October 1990.

[CQ91a] R. Cooper and Z. Qin. An Implementation of the IFO Data Model. Technical Report CSC 91/R14, Department of Computer Science, University of Glasgow, August 1991.

[CQ91b] R. Cooper and Z. Qin. Constraint Management in a Configurable Data Modelling System. Technical Report CSC 91/R14, Department of Computer Science, University of Glasgow, August 1991.

[CR92] J.L Cybulski and K. Reed. A Hypertext Based Software Engineering Environment. *IEEE Software*, pages 62–68, March 1992.

[CT92] R.L. Cooper and I. Tabkha. A Semantic Framework for the Design of Data Intensive Applications in a Persistent Programming Language. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, volume II of *Software Technology*, pages 799–809, 1992.

[DD95] C.W. Dawson and R.J. Dawson. Towards more flexible management of software systems development using meta-models. *Software Engineering Journal*, 10(3):79–88, May 1995.

[DEFH87] Susan A. Dart, Robert J. Ellison, Peter H. Feiler, and A. Nico Habermann. Software Development Environments. *Computer*, 20:18–28, November 1987.

[DM78] T. De Marco. *Structure analysis and system specification*. Yourdon, 1978.

[Dow87] Mark Dowson. Integrated Project Support with IStar. *IEEE Software*, 4:6–15, November 1987.

[FIJK94] R. Figueira, G. Ilsley, N. Jefthas, and C. Katts. DEN: A Development Environment in Napier88. Honours project, University of Cape Town, 1994.

[Gri94] Gary Griffiths. CASE in the third generation. *Software Engineering Journal*, 9(4):159–166, July 1994.

[GS77] C. Gane and T. Sarson. *Structured systems analysis*. IST, 1977.

[Han83] K. Hanse. *Data Structured Program Design*. Ken Orr and Associates, Topeka, KS, 1983.

[HK87] Richard Hull and Roger King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computer Surveys*, 19(3):201–260, September 1987.

[HM81] Michael Hammer and Dennis McLeod. Database Description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.

[HS90] Neil Haddley and Ian Sommerville. Integrated support for systems design. *Software Engineering Journal*, 5(6):331–338, November 1990.

[IBM78] on the contents of a sample of programs surveyed. Technical report, IBM, San Jose, California, 1978.

[Jac83] M.A. Jackson. *System Development*, chapter 7,10. Prentice-Hall, Englewood Cliffs, NJ, 1983.

[Joh75] S. C. Johnson. Yacc - yet another compiler compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[KBC+94] G. Kirby, F. Brown, R.C.H. Connor, Q. Cutts, A. Dearle, R. Morrison, and D. Munro. The Napier88 Standard Library Reference Manual (Version 2.2). Technical Report CS/94/7, University of St Andrews, 1994.

[KCCM92] G. Kirby, R. Connor, Q. Cutts, and R. Morrison. Persistent hyperprograms. *Fifth International Workshop on Persistent Object Systems*, pages 86–106, 1992.

[KS86] Henry F. Korth and Abraham Silberschatz. *Database system concepts*. McGraw-Hill, 1986.

[Lav94] Darryn Lavery. The Design of Effective Software Visualizations for Persistent Programming Environments. Technical Report FIDE/95/116, Dept. of Computer Science, University of Glasgow, 1994.

[LW94] M. Lloyd-Williams. Knowledge-based CASE tools: improving performance using domain-specific knowledge. *Software Engineeing Journal*, 9(4):167–173, July 1994.

[Mar83] James Martin. *Managing the Data Base Environment*. Prentice-Hall, 1983.

[Mar88] C.F. Martin. Second-Generation CASE Tools: A Challenge to Vendors. *IEEE Software*, pages 44–49, March 1988.

[Mat95] F. Matthes. Higher-Order Persistent Polymorphic Programming in Tycoon. In M.P. Atkinson, editor, *FIDE: Fully Integrated Data Environments*. Springer-Verlag, 1995.

[MBC+] Ron Morrison, Fred Brown, Richard Connor, Quintin Cutts, Al Dearle, Graham Kirby, and Dave Munro. *The Napier88 Reference Manual (Release 2.0)*. University of St Andrews. CS/93/15.

[MBC+94] R. Morrison, F. Brown, R.C.H. Connor, Q. Cutts, A. Dearle, G. Kirby, and D. Munro. The Napier88 Reference Manual (Release 2.0). Technical Report CS/93/15, University of St Andrews, 1994.

[MDB+85] R. Morrison, A. Dearle, P.J. Bailey, A.L. Brown, and M.P. Atkinson. The Persistent Store as an Enabling Technology for Integrated Project Support Environments. In *Proc. 8th IEEE International Conference on Software Engineering*, pages 166–172, London, 1985.

[MO95] Y. Mallum and R. Oozeer. Object Modelling Techniques in Napier88. Honours project, University of Cape Town, 1995.

[NC92] J. Norman, Ronald and Minder Chen. Working Together to Integrate CASE. *IEEE Computer*, pages 12–17, March 1992.

[O2 93] O2 Technology, ruc du Parc de Clagny, 78000 Versailles, France. *The O_2 Programmer's Manual – Version 4.3*, 1993.

[Pen87] Maria H. Penedo. Prototyping a Project Master Data Base for Software Engineering Environments. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, volume 22 of *SIGPLAN Notices*, pages 1–11. ACM, Association of Computing Machinery, Inc., January 1987.

[Pet77] J.L. Peterson. Petri Nets. *Computer Surveys*, pages 223–253, September 1977.

[PM88] Joan Peckham and Fred Maryanski. Semantic Data Models. *ACM Computer Surveys*, 20(3):153–189, September 1988.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall International, 1991.

[Rei87] Steven P. Reiss. Working in the Garden Environment for Conceptual Programming. *IEEE Software*, pages 16–27, November 1987.

[Ris85] N Rishe. Semantic modeling of data using binary schemata. Technical Report TRCS85-06, University of California, Santa Babara, California, 1985.

[Ris86] N Rishe. On representation of medical knowledge by a binary data model. In X.J.R. Avula, G. Leitman, C. D. Mote Jr., and E. Y. Robin, editors, *Proceedings of the 5th International Conference on Mathematical Modelling*, Elmsford, N.Y., 1986. Pergamon Press.

[Roy70] W.W. Royce. Managing the Development of Large Software Systems: Concepts and Techinques. Technical report, WESCON, Western Electronic Show and Convention, Los Angeles, August 1970. Reprinted in: *Proceedings of the 11th International Conference on Software Engineering*, Pittsburg, May 1989, pp 328-338. (Chapter 3).

[Shi81] David W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.

[Sjø91] D. Sjøberg. The thesaurus – a tool for meta data management. Technical Report FIDE/92/37, FIDE, 1991. ESPIRIT Base Research Action, Project Number 3070.

[Sjø92] D. Sjøberg. Measuring name and identifier usage in Napier88 applications. Technical Report FIDE/92/37, FIDE, 1992. ESPIRIT Basic Research Action, Project Number 3070.

[SM91] J.W. Schmidt and F. Matthes. Modular and Rule-Based Database Programming in DBPL. Technical Report FIDE/91/16, FIDE, 1991.

[SPWW97] D.I.K Sjøberg, P.C Philbrow, C. Waite, and R. Welland. Build Management in Database Programming Language Environments. In to appear in *Software Practice and Experience*, 1997.

[SS77] J. M. Smith and D. C. P. Smith. Database abstractions: Aggregation and generalization. *ACM Transaction on Database Systems*, 2(2):105–133, March 1977.

[Tea93] The GoodStep Team. General Object-Oriented Database for Software Engineering Processes. Technical Report 1, ESPIRIT-III Project, November 1993. GOODSTEP (6115).

[TH77] D. Teichroew and III Hersey, E.A. PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems. *IEEE Transactions on Software Engineering*, pages 41–48, January 1977.

[Tri88] L.L. Tripp. A survey of graphical notations for program design - an update. *ACM SigSoft Software Engineering Notes*, 12(4):39–44, 1988.

[Wet94] 1994 Wetzel, I. *Programming with STYLE: On the Systematic Development of Programming Environments*. PhD thesis, University of Hamburg, 1994.

[WMS95] I. Wetzel, F. Matthes, and J.W. Schmidt. The STYLE Workbench: Systematics of Typed Language Environments. Technical Report FIDE/95/139, FIDE, 1995.

[WMWM87] J.H. Walker, D.A. Moon, D.L Weinreb, and M. McMahon. The Symbolics Genera Programming Environment. *IEEE Software*, 4:36–45, November 1987.

[WP87] Anthony I. Wasserman and Peter A. Pircher. A Graphical, Extensible Integrated Environment for Software Development. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, volume 22 of *SIGPLAN Notices*, pages 131–142. ACM, Association of Computing Machinery, Inc., January 1987.

[WPA+97] C.A. Waite, P.C Philbrow, M.P. Atkinson, R.C. Welland, D.O. Lavery, S.D. Macneill, T. Printezis, and R.C. Cooper. *to appear in* The Nordic Journal of Computing. Technical report, 1997.

[YC75] E. Yourdon and L.L. Constantine. *Structured Design*. Yourdon Press, 1975.