On The Utilisation of Persistent Programming Environments

Richard Cooper

A thesis submitted to the Faculty of Science,
University of Glasgow
For the degree of Doctor of Philosophy
September, 1989

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

Declaration

The material presented in this thesis, except where stated below, is the product of my own independent research carried out at the Department of Computing Science, University of Glasgow under the supervision of Professor Malcolm Atkinson. Sections 4.2, 4.4, 6.1, 7.1 and some of section 4.1 represent work carried out by other workers in the persistent programming projects, which is used in this thesis to elicit a common methodology. Any published or unpublished material used by me has been given full acknowledgement in the text.

Acknowledgements

I would like first of all to thank ICL Ltd. for their financial support for the major part of this work.

I would also like to acknowledge the assistance of my colleagues in the PISA project at Glasgow who proved stimulating co-workers: Jack Campin, who so often finds a simple way to put a complex idea, Paul Philbrow, who was always eager to fix problems and facilitate other people's work, and Francis Wai, a much-interrupted room-mate. My indebtedness extends to our colleagues at the University of St. Andrews for many useful conversations, for their excellent hospitality and for providing PS-algol systems which while being experimental exhibited a remarkable degree of robustness and efficiency. In particular, I would like to acknowledge the example of Al Dearle, both for introducing me to diving and for the title for the thesis borrowed from him (an indication of the quality I would like to find in my own work), and of Professor Ron Morrison for illuminating shafts of sanity when the murky waters of confusion thickened.

I would also like to acknowledge my other colleagues in Glasgow too numerous to mention in total, but in particular David Harper, Phil Gray, Kevin Waite, Kieran Clenaghan, Phil Trinder and Ray Welland, from each of whom I gained insights into some part of the work presented here. Some of the work was performed collaboratively with Djamel Abderrahme and Zhenzhou Qin, whose collaboration I acknowledge with gratitude. The work in Chapter 8 benefited both from the advice of Dr. Chris Marlin and from many discussions with David Kerr, while Prof. John McLeish gave me a great deal of confidence boosting assistance at the beginning of my thesis.

My principal thanks however are due to two people. My supervisor, Malcolm Atkinson, showed support for me and a commitment to my work far exceeding all reasonable expectation. Every meeting with him has resulted in some growth in my research work and my working life. My wife, Rosemary McLeish, has also shown a level of love and support which is exceptional and without which this thesis would not have come to be.

Abstract

There is a growing gap between the supply and demand of good quality software, which is primarily due to the difficulty of the programming task and the poor level of support for programmers. Programming is carried out using software tools which do not match very well either real world understanding of a problem or even the other tools which need to be used. In every phase of software production, the programmer must master new tools which function in a different way from each other.

The Persistent Programming Paradigm attempts to reduce these problems by providing a programming environment which gives consistent methods of accessing program values of various kinds. Long-term and short-term data are treated in the same way. Numbers, text, graphical values and even program objects are all referred to in the same consistent way. Languages which support persistence provide considerable power within a simple environment, so that programmers can perform most if not all parts of the programming task in a coherent and uniform manner.

This thesis tests the hypothesis that programmers do in fact derive some benefit from this - the simplification of the program and faster implementation of complex programs. The persistent language PS-algol is introduced and used to build: user-interface and compiler tools; a database application; some data modelling tools, both relational and semantic; a rapid prototyping system; an object-oriented language; and software support systems. In doing so, the thesis demonstrates the breadth of work which can be achieved using a Persistent Programming Language, and the ease with which these various projects can be implemented.

Further, the thesis derives the beginnings of a methodology for using such a language and analyses how PS-algol could be improved. In doing so, the work aims to put the Persistent Programming Paradigm on a firm basis following significant use and experimentation.

Table of Contents

List of Figures.

Chapter 1. Introduction.

This thesis examines the claim that persistent programming languages facilitate the production of software. The need for better tools for software production is discussed, the claim of persistent systems in this regard is presented and then this claim is investigated with reference to the first significant persistent language, PS-algol. Several experiments using PS-algol are described, from which a methodology for using the language is derived. The thesis ends with some conclusions on the overall effectiveness of the language and the approach.

A programming paradigm comprises a computational model, languages that realise this model and a culture and conventions for the use of those languages. The persistent paradigm is developed in this work. The model and languages are the result of earlier work, but this thesis aims to provide a major contribution to the culture and conventions.

1.1 The Software Engineering Crisis.

It is a recurrent challenge to the Computer Scientist that the demand for software exceeds the ability to produce it [ACARD, 1986, Warren, 1988]. Computers are used for a widening range of increasingly complex tasks. The growth is due largely to the provision of cheaper and more powerful hardware, but also to social factors such as the greater tolerance for computers among the public and rising expectations from all users of information systems. This, in turn, is due to the improved quality of software produced. This is a measure of the success of software developers and results in a growth of demand for software which outstrips the rate at which programmers are trained.

There are a number of possible approaches to this problem. For instance, the problem could be accepted as insoluble and the cost of software production be allowed to increase and in this way demand would be controlled. Alternatively, even more human resources could be diverted into software production although this loses sight of the secondary nature of most of the computer industry. It is there to produce tools to support other tasks, not to drain manpower away from them. The most satisfactory approaches, however, include making programming easier to do, making it easier to prove programs correct and reducing the amount of code required for a given task.

The intrinsic reason why programming is difficult is that computers operate in a formally defined symbol space. They can only do what is told to them in a way which can be represented formally and precisely - and people are not particularly skilled at the precise specification of tasks. The fact that the representation is formal is a necessary limitation of the use of computer systems - even apparently informal, mouse-driven interactive systems operate according to formal rules. However, the nature of the formalism can be controlled. In the earliest computers, the formalism consisted of bit-strings, which only the arithmetically skilled could manipulate. Programming languages were then developed which at least began to overcome the memory load imposed by bit-strings. The computer took the program and produced the bit-strings itself. However, the languages have traditionally tended to reflect to a greater rather than a lesser degree the structure of the computational model.

The central task of the programmer, then, is to take an *ad hoc* informal description of a problem and transform it into a description in a formal language. For a given problem, the difficulty of this process is dependent on the nature of the formal language. The more the language reflects the structure of the computational model, the easier the task for the computer to turn the formal description into bit-strings and the more difficult the task for the programmer to produce the formal description in the first place. One way to make programming easier is, then, to make the formal languages more closely resemble the languages with which people are used to describing the world. This can now be done, because as computers become more powerful, more and more of the translation task can be thrown onto the computer and the programmer can be freed from the worst of the task - the identification of programming constructs which correspond to real-world intuitions and the routine repetition of low-level tasks. Note that this does not reduce the requirement for precise formulation of the problem, but rather factors out recurrent detail and provides a more direct mapping between the objects in the real application domain and their computer representations.

The second approach is to make programs easier to prove correct. If it is possible to know that a program (or even a part of it) is definitely in accordance with its specification, then there will be a great saving in the debugging time which dominates software development. Two approaches which lie outside of the scope of this thesis are the development of functional programming languages [Glaser *et al.*, 1984, Bird and Wadler, 1988] and formal specification techniques [Gehani and McGettrick, 1986, Bjørner and Jones, 1982]. One problem with these approaches is that they do not yet appear to cover many of the aspects of long-lived systems, which are the particular concern of the persistent programming paradigm. One aspect of proving correctness which is discussed here is the use of strongly typed languages. It has been estimated that 70% of all programming errors are type-mismatch errors [Buneman, 1988]. The use of a strongly typed language gives the earliest possible detection of such errors and thus saves debugging.

The third approach is to cut down on the amount of programming there is to do. Two themes appear here. Database systems are examples of programs in which code is provided to give a range of facilities to a number of applications without those facilities having to be reprogrammed. The database system is programmed in a "machine-oriented" language and provides an interface for the application designer to pick and choose the facilities required in simpler or more "human-oriented" languages. Applications of considerable power can be produced quickly using such systems. The DBMS designer made a once and for all *a priori* choice of the functions to be factored out. One of the particular benefits of the persistent programming paradigm is that this factoring can be postponed and used incrementally - thus providing the ability to produce personalised versions of the application development environment.

The other theme is that of software re-use. Code is stored in libraries accessible to programmers, thus obviating the need to code facilities more than once. This activity may be viewed as a kind of database application in which the data being stored are code fragments. Support for such a library is one of the experiments described in this thesis (Chapter 8). Improvements in Software Development Environments facilitate access to such code, whilst the Object-Oriented paradigm brings facilities for the re-use of code into the language. This is one example of the use of polymorphism to permit the same piece of code to be usable over a variety of types.

The two principal elements of application development are the storage of data and the specification of the code manipulating the data. In an application of any magnitude, both of these will be complex and interacting tasks. Most applications which are primarily concerned with data storage and retrieval will eventually develop a need for some computation. Similarly, most computationally intensive programs will want to store partial results. However, the approaches above make little attempt to integrate these two tasks. Traditionally programming languages are targeted at the data manipulation part, while database systems, file managers, etc. deal with the data storage part. To do so, each takes a somewhat different view of the structure of data. The development of database programming languages is intended to bring these two parts into a common framework, but has to contend with the discrepancy between these views (called the "impedance mismatch" problem in the literature).

Moreover, the approaches above take the view that one or other of program and data dominates. In traditional languages the application is developed first and then data are added. In Object-Oriented systems the data structure dominates, with program being added in the context of the data structure. In reality, application development needs to occur incrementally, with data and program being added independently in whichever order is required. The **Persistent Programming** paradigm takes the view that program and data have equal status in the application space.

Another tension which will be produced on trying to benefit from the approaches above is that between strong typing and polymorphism. Specification of polymorphic code may save on coding, while strong typing may save on debugging, but these two techniques may conflict. The design of languages which provide a good compromise is an active research area. Persistent Programming resolves this tension by permitting a mixture of static and dynamic type checking. Therefore static checking occurs whenever possible, but the checking of polymorphic code is deferred for as long as possible.

This thesis concentrates on the claims of **Persistent Programming Systems**. The term Persistent Programming System arises since they provide "orthogonal persistence" for data. This means that every piece of data in a program has the same rights to outlast the program (or be transient) as any other - no matter what type it is. This principle sits inside a general concept that a Persistent Programming System should not present the programmer with arbitrary distinctions between the way in which different types of data are manipulated. Thus it is satisfactory to insist that strings and integers can be distinguished because strings cannot be multiplied together, but not because strings can be stored but integers cannot - that would be an arbitrary restriction. Such arbitrary restrictions litter programming languages - consider what can and cannot be done with procedures in Pascal. A Persistent Programming System, then, is one in which all such arbitrary restrictions are removed and the claim of such systems to provide better programming environments will be discussed next and examined in the rest of the thesis.

1.2 The Claim of Persistent Programming.

The **persistence** of a value is the length of time for which it may be used by a program. A given value can exist for very short time (until the end of the block); for the length of the program run; for the lifetime of the database; or even longer if the data are switched to a new database when the initial one is replaced.

The concept of a Persistent Programming System embraces two principles. Firstly, that **any value can have any degree of persistence**. That is, all values of any type have the same rights to be short-lived or long-lived. Secondly, that **the way an object is referred to should not depend on its persistence**. That is, there is not one way of referring to values which cease to exist when the program terminates (these are called **transient** values) and another for referring to long-lived (or **persistent**) values. What emerges from these principles is the notion that to make values persist beyond the end of the program should take little effort.

One example of explicitly making values persist is the use of file systems (for instance within Pascal programs). In such programs, the piece of code implementing the algorithm on which the program is based must be augmented by two further pieces of code. The first starts the program by reading data out of files, while the second finishes the program by writing it back to files. It has been estimated that the code involved in these two pieces takes 30% of the programming effort on average [IBM, 1978]. These two pieces of code are not central to the programmer's intention - they are extra tasks to be done, in order to get the main section to work. Worse than that, they require a completely new way of conceptualising the data, which complicates the program in the following way.

When the central part of the program is being written, a structure is imposed on the data which is suitable for the algorithm being programmed. The programmer thus must have two views of the world: the external reality; and this "algorithmic model"; and must keep in mind two translation processes between these two views. The use of a filing system imposes a third, wholly unnecessary view of the world: the structure the data has in the filing system. Now the programmer has three views instead of two and six translation processes instead of two - a significant increase in the complexity of the system and to achieve what? Putting away the data, so that it can be re-accessed another time.

Persistent Programming Systems avoid this extra complexity by using the algorithmic model to store the data. Take for example a program that requires a tree-structure for its data. A file-based approach means that the nodes of the tree must be flattened into some arbitrary but important order at the end of the program and then retrieved in the same order at the beginning of the next run. A superior approach would seem to be just to state "store the tree". More specifically "store the root of the tree" and, as a consequence of this being a tree, all of the other nodes would be stored as well. The structure within which it is stored is of no concern to the programmer - the system should decide upon the most efficient structure. The only important aspect is that the programmer can retrieve the tree, again by specifying the retrieval of the root node. This technique, which consists of storing one object with the consequence that any object it refers to is also stored, is referred to as **persistence by reachability**.

Another technique used to provide a form of persistence is employed by languages such as Prolog and ML. In these languages, at the end of a program run, the whole program space can be saved so that, when a program is restarted, its data space will be exactly as it was when the program was last quit. This is a very coarse way of providing persistence. Firstly, it is all or nothing - everything gets saved, whether it is useful or not. Secondly, it seems to preclude the possibility of two programs sharing data - each program lives in its own little world - and hence inhibits concurrency and incremental system development.

In fact, Persistent Programming Systems generalise this notion of removing the arbitrary restriction on the persistence of objects in most programming languages to a general programme for removing any such discontinuities. That they can do this is due to the increase in hardware performance. In the previous paragraph, it was blithely stated that storing the root node causes the rest of the tree to follow. The construction of a system which will do this is an extremely complex programming task [Cockshott, 1983, Brown and Cockshott, 1985, Dearle, 1988, Brown, 1989] and relies for even reasonable performance on good hardware.

Discontinuities in computing systems arise for a number of reasons:

i) historical reasons - for instance, independently developed technology being made to interwork;

ii) engineering limitations - some of which have ceased to be significant, while others can be hidden by automatic means;

iii) different human perspectives dictating the design of different parts of the system;

iv) systems being partitioned during design in order to achieve large-scale and complex systems;

v) a lack of fundamental understanding of the total computational process.

With increased understanding of the nature of the underlying problems, Computer Science is now in a better position to begin to remove these discontinuities, replacing *ad hoc* and badly matched components with single coherent systems. In doing so, more of the programming tasks will be brought into a common framework thus making them simpler and more manageable.

Some approaches to the removal of these discontinuities include:

- the use of any kind of value in expressions, as variables or as the parameters of procedures (as compared with Pascal procedures in which the results may not have compound types);

- the provision of consistent mechanisms for introducing objects into programs;

- the inclusion of richer types, including multimedia types, thus extending the simplicity with which different values may be expressed in the language;

and

- the extension of the scope of the language to include more of the programmer's activity, for instance the HCI and persistent parts of the program.

In conceiving the program, a programmer is likely to view as values: simple things such as strings and integers; complex objects, such as ships; and processes or activities involving those objects. The programming language should reflect this, by permitting the programmer to refer to all of these same values in the same way. In particular, the facilities most languages provide for manipulating procedures are restricted to defining them and applying them. A language which allows procedures

to be first-class objects provides significantly greater modelling power than one in which only static objects can be described. One which supports processes would probably be a further advance [Morrison *et al.*, 1989].

These requirements can be wrapped up into one over-riding principle - that the language be **data-type complete** (there are no arbitrary distinctions between the way values of different types can be manipulated) [Morrison, 1982]. There is an additional, implicit expectation of such a system that this completeness will be provided by leveling-up. If type *X* can do operation *O*, but type *Y* cannot, the system will be changed so that both can do *O*, and not so that neither can. The system therefore becomes more powerful as well as simpler. A PPS is, therefore, one in which the programmer is provided with a programming environment with as few distracting distinctions as possible. The claim is that such a system will greatly increase programming efficiency, because:

a) the programmer is not distracted by discontinuities;

b) the language is easier to learn;

c) the language provides a single model of all the data held or processed by the computer;

d) it is easier to argue about program correctness if the code is more transparent and there are fewer effects taking place outside the formally defined system;

and e) the code is more succinct.

1.3 The Programming System Used.

The experimentation has been carried out in the context of the programming language, PS-algol. This member of the algol family provides persistence by reachability, is data-type complete, has graphical data types and conveys simplicity of expression combined with power. The language was developed at the Universities of Edinburgh, St. Andrews and Glasgow, as described in section 2.3, and is essentially a research vehicle for experimenting with the ideas presented here. The constructs provided by the language will be described in more depth in Chapters 3 and 4.

The language has been implemented on a variety of hardware - DEC VAX 11 series under VMS and UNIX, ICL series 39 under VME, ICL Perq workstation under PNX, Sun workstations under UNIX and the Apple MacIntosh. PS-algol has been designed to be machine independent in the sense that if two machines have the same facilities they will be made available in the same way. The bulk of the work described here was carried out on the Perq and Sun workstations. These both provide large high resolution screens, a mouse driven input device and a UNIX-like operating system. As such the two implementations were identical (apart from performance) from the viewpoint of the programmer.

A PS-algol system consists of a compiler, an interpreter, a persistent store and a number of systems programs. The compiler takes PS-algol source code and produces an abstract machine code, which is then executed by using the interpreter. The persistent store consists of a set of PS-algol "databases" (described in section 3.2.5) and

the system is initiated to contain databases containing system code and fonts. Using the system consists of writing programs which manipulate this persistent store - adding, modifying and removing objects in it. There is one system program provided which allows the user to browse the persistent store, navigating through the store by following object references. This browser and its implementation is described in more detail in section 4.2.

1.4 The Need for a Culture, Methodology and Support Environment.

The last section describes the supplied PS-algol system and, when this work began, this was a Spartan system within which to start writing programs. A new user also has access to the user manual [PS-algol, 1987], a few tools and maybe the source code of those parts of the system written in PS-algol itself. To tackle programming using a new paradigm with such small assistance would seem a daunting task. Since then more tools, such as the browser, and an introductory tutorial to PS-algol [Carrick *et al.*, 1987] have been added. However, it has been one of the principal aims of this research to improve this situation by the addition of fresh components and the development of methodological guidelines. The author had the advantage that the work was carried out in the context of a group of PS-algol programming enthusiasts, who had, at least implicitly, developed an initial culture and an initial understanding of such languages.

Programming in any conventional language is facilitated by the considerable experience available from those who have used it in the past. There is a reasonably clear idea of how best to program any kind of problem in Pascal, for instance. There is normally a considerable amount of code available for inspection, modification and re-use. Similarly, every UNIX system comes with a significant library of C routines and these can be easily added to from any of a number of sources depending upon the kind of application that is required.

A new language in a new paradigm provides no such support. The number of PS-algol experts was restricted to a small number of people in Scottish Universities and one department of STC Technology Ltd., with a scattered following in Australia. No matter how good the language might be, there will reasonably be a considerable reluctance to use it if there is a significant unsupported learning task to be performed. Persistent programming is a new paradigm and will benefit from different ways of performing familiar tasks.

A central aim of this research was to develop a systematic way of using the language, together with supporting software tools and a library of re-usable modules. The development of some re-usable modules are described in Chapter 4 and their organisation into a library in Chapter 8. Chapter 9 then provides a methodology which may be followed when writing programs in the language. Chapter 10 concludes with some overall impressions of persistent programming in general.

1.5 The Thesis Statement.

For a new programming paradigm which purports to improve the economy of programming, several questions need to be asked to substantiate this claim:

What programming tasks can the language be used for? There is a need to verify that languages like PS-algol are sufficient for a range of programming problems.

How do the novel aspects of such a language simplify programming? Do persistence, data-type completeness and the other features simplify programming?

Is there any cost in using such languages? Are they inevitably slower? Is expressive power reduced?

What are the implications of any weaknesses? Are they due to a failure of the paradigm or are they a consequence of experimenting with an early prototype?

This research attempts to find answers to these questions by making use of PS-algol to implement sophisticated, large-scale programs. The experiments include the implementations of: a traditional data-intensive application; higher-level data models; and systems tools such as compilers and software support environments. In doing so, the limits of complexity, power, speed or functionality available in the language will be tested.

The research depends on the assumption that PS-algol is a useful representative of the potential of Persistent Programming. Programs written in the language by others were analysed to determine the influence of the paradigm on program structure. Further programs were written both to develop the paradigm and to carry out further evaluation. One difficulty this method encounters is that the evaluation is influenced by the development of the paradigm. From the observations it is necessary to extrapolate in order to judge the overall effectiveness of the paradigm.

The statements examined in this research are:

> **Persistent programming, as exemplified by PS-algol, is a sufficient and effective foundation for the development of large, complex and long-lived systems.**
>
> **The paradigm beneficially influences the style of programming carried out.**
>
> **A methodology can be developed which facilitates this style of programming.**

1.6 An Outline of the Experiments and the Structure of the Thesis.

Figure 1.1 shows an overall map of the structure of the thesis. After the introductory chapter, there is a chapter surveying a variety of approaches to the problem of improving the economy of software production from the perspectives of language design, database systems and software engineering. This chapter also introduces the Persistent Programming Paradigm in more detail. Chapter 3 then introduces the principle features of PS-algol.

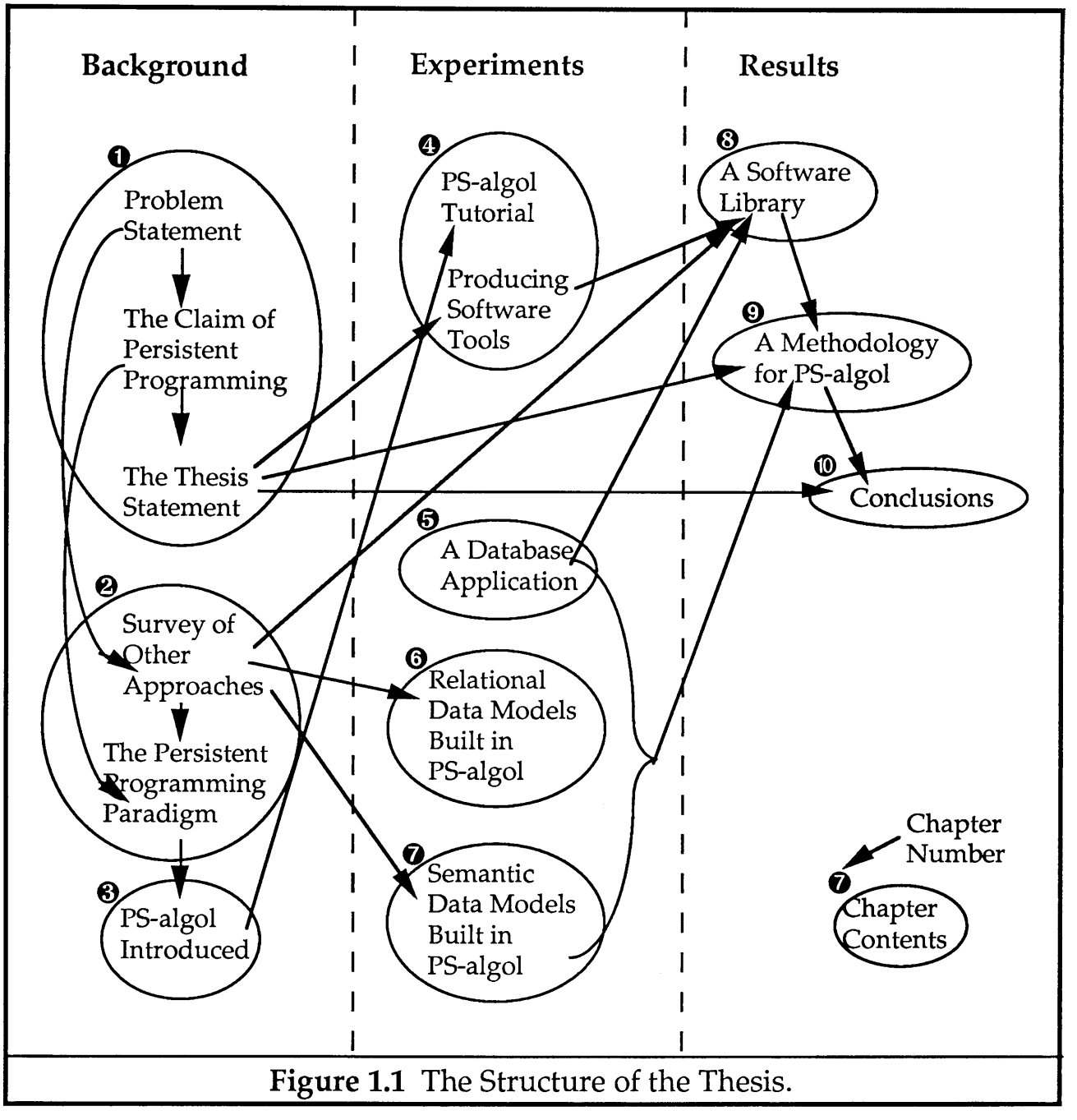


Figure 1.1 The Structure of the Thesis.

Chapter 4 is intended as a tutorial in the language, but is also used to describe an experiment in providing re-usable software components. This demonstrates the ease with which a software library to perform some basic functions can be built up in PS-algol.

Chapter 5 describes the implementation of a data-intensive application - that of maintaining a database of bibliographic references. This is a demonstration that PS-algol is sufficient for a task of this sort.

Chapter 6 describes two experiments in providing a classical database environment for developing applications. These both implement relational databases and demonstrate PS-algol power as a meta modelling tool for database design.

Chapter 7 takes this idea somewhat further and demonstrates how even higher-level data modelling systems can be built in PS-algol. These comprise three semantic data modelling systems and an object-oriented programming language.

Chapter 8 describes experiments for the manipulation of software itself. The persistent store as a repository for software components is at once extremely powerful and without any kind of supporting structure. Experiments here show how such supporting structures can be built, including notions of version control and configuration management.

These experiments, taken together, will then be held to demonstrate that PS-algol has sufficient expressive power to perform a great number of programming tasks. Chapter 9 then takes the experience of implementing these programs and derives from it a methodology for programming in PS-algol. The chapter also describes certain deficiencies in the language, due principally to its prototypical nature.

Finally, the concluding chapter re-states the evidence for the overall thesis that persistent programming is an effective paradigm and that a methodology for the paradigm can be produced. Necessary prerequisites are identified and either produced or shown to be feasible via prototypes. The thesis concludes with recommendations for future work to develop the paradigm and to use it for further complex programming tasks.

Chapter 2. A Survey of Approaches.

This chapter surveys a variety of approaches to the problems of constructing software on a large scale. This is carried out in the context of three main lines of approach - providing increasingly effective programming languages; providing increasingly effective database systems; and providing increasingly sympathetic environments within which to develop software. Much current work, including that reported in this thesis, attempts to combine these lines of approach in order to gain the benefits of all worlds.

In particular, the effective provision of usable software environments is shown to be eased by using database technology to store the code. It is also shown that integrating a programming language with a database as a single system gives the benefits of computational completeness and data access efficiency without the need to switch continually between two programming worlds. This last point is part of the underlying concept behind all of this work. Efficient programming depends on having a programming environment which is at once simple and powerful. This means that the programmer should be given as few tools as necessary; a uniform programming environment; as few exceptions to overcome as possible; and should be enabled to express requirements as naturally as possible.

This thesis synthesises ideas from these three domains, in order to support persistent and large scale programming. Consequently, it is necessary to review each area in turn.

2.1 Three Approaches.

2.1.1 The Development of Better Programming Languages.

The history of the development of programming languages has shown that poor design is due to a number of factors. Firstly, the constituents of a good language are difficult to determine. Secondly, compromises have been necessary to produce languages which could be implemented efficiently. Thirdly, language design has been influenced to a great degree by the architecture of computer systems. Considering the software crisis described in section 1.1, it may now be time to tailor the computer architecture to good language design, rather than vice-versa.

It is now possible to make some general statements on the constituents of a good programming language. The language should be designed to produce simple and natural descriptions of applications. The correctness of programs should be determined as soon as possible. The language should be simple without forfeiting power. There should not be too many constructs and there should be no arbitrary exceptions to the semantic and syntactic rules of the language.

Programming languages were originally designed to be appropriate for the specification of arithmetical algorithms. This was seen as the prime use for computers and therefore languages such as algol and Fortran proliferated. They concentrated on a store model of computation and were sufficient for small scale programming. These languages as well as contemporaneous languages for other application areas such as COBOL and LISP all took a microscopic view of the programming task. The

programmer had to do the bulk of the translation work from the task into the small impoverished world of the programming language.

The experience with these languages led to a number of developments. Firstly, superior "traditional" languages such as Pascal, algol-68 and PL/1 appeared with an increasing number of features and overall increase in complexity which offset their gain in expressive power. This line of development culminated in the language ADA, which strives for maximal power by providing a considerable number of programming constructs, supplementary to the basic computational model of algol and Pascal. There also appeared the notion of structured programming as a conceptually simplifying framework within which to produce programs. A top-down development of programs was recommended, which resulted in cleaner programs. The drawback of this approach is that it may be alien to the way in which many people work. An incremental approach, specifying parts of a system in no particular order, fits better with the way many people conceptualise very large tasks.

A second line of development is functional programming [Glaser *et al.*, 1984; Bird and Wadler, 1988]. This imposes an even more stringent structure on a program. All computation is specified in the form of functions which use no variables to store partial results. No "side effects" are permitted and an exact correspondence between a program and its specification can be demonstrated. Once again, however, there is a cost to the programmer. Where it feels natural to store information for later use, other, less direct, mechanisms must be used. Furthermore, the basic problem of storage and access to large amounts of long-lived data has not been effectively resolved within the functional paradigm.

The most recent line of development has been that of object-oriented programming languages [Dahl and Nygard, 1966; Goldberg and Robson, 1983; Meyer 1988]. These argue strongly for a natural way of describing tasks to be given to the computer, but also impose a rigid structure upon the program. An object-oriented program consists of the descriptions of a number of kinds (or classes) of object that are to be manipulated. A **class** describes the passive features or attributes and the active feature or operations that are common to all objects of the class. This bringing together of the passive and active descriptions certainly produces a conceptual simplification for the programmer, but has two drawbacks.

Firstly, the common way that the two sets of features are specified has an unfortunate implication which causes confusion. A passive feature describes a component of an object of the class. Each object of the class will have a component or attribute of the specified type whose value will vary from object to object. Conversely an active feature is an operation which is common to all objects of the class. That is, a class attribute describes a set of things, while a class operation describes only one thing. This violates a basic principle of the natural representation of objects in a language - the same sort of description is used for things of greatly differing nature.

In fact this violation is due to the non-orthogonal provision of two features. There is no intrinsic reason why object-oriented languages should not provide the ability to specify attributes and operations which are specific to instances and attributes and operations which are general to the class. This ability to factor out common descriptions is a major benefit of the object-oriented approach, which is lost if this is not provided orthogonally to other facilities.

The second problem with object-orientation is considerably more serious. The only place where program code may appear is as the operations associated with a class. This constraint on where code can appear forces the program into severe contortions when trying to achieve fairly straightforward tasks. Mention is made here of the problems of representing code that is not easily viewed as being associated with some class; code that represents dyadic operations; and code that is essentially a component of some object. Section 2.2.2 goes in much more detail into these problems and on object-oriented systems in general.

Two other sorts of language have appeared recently: specification languages and rapid prototyping languages. The former provide the ability to make precise statements at a very high level about selected aspects of a program's functionality. This is intended as an aid for conceptualising the design of the program and for verifying the design and is independent of the computational paradigm. As these languages are currently provided, such specifications are not usually executable, although if the specifications really represent the functionality required this should be possible without the need to respecify in another language. One of the goals of this research is to show how, given a persistent environment, it is possible to produce implementations which are executable. However, if specifications are made executable, the differences between them and programming languages have been reduced to nothing, thus rendering them redundant.

Rapid Prototyping Languages, on the other hand, give the appearance of being executable. They resemble the stage set of a film or theatre, providing the appearance of something that works, but not the reality. These languages are used to provide a system quickly according to a specification for experimentation purposes. Used extensively in the context of user-interface design, prototypes can, for instance, allow users to test how a program will feel, before it is actually built. The user interface can then be modified in the light of any criticisms. Rapid prototyping can also fulfill a different rôle in the development of very large systems. Each component can be prototyped so that inter-component interfaces can be verified by the prototype, before the actual components are implemented [Cooper *et al.*, 1989].

One issue which cuts across the different paradigms is the kind of type checking provided. The type system of a language is a framework within which data can be structured. To indicate that a piece of data is of a given type means that its structure and what can be done with it are completely defined. Therefore, if an attempt is made to use it in different ways or as if it had a different structure, this will result in a reported program error and not in the corruption of the data. Buneman claims that such attempts to misuse data account for 70% of all programming errors [Buneman, 1988].

Therefore two conclusions may be drawn. A program should provide a type system, which cannot be violated - otherwise this kind of error will give rise to obscure and potentially catastrophic errors. Secondly, the sooner the programmer is made aware of these errors, the less costs will be incurred. Languages in which all use of data and program must comply with their type specifications are said to be **strongly** typed. If these languages perform all the type checking as the program is being compiled, they are said to be **statically type checked**, while if all these checks are deferred until the data are used at run-time, they are said to be **dynamically type checked**. The persistent paradigm provides languages which use a mixture of both.

Clearly strong typing is essential and static type checking is desirable since the errors will be detected before the program is run. However, static type checking requires that all code is written to run against explicitly stated types and this means that no code can be written to run over a range of types. In order to write such **polymorphic** code, the decision on which data types the code uses in a particular run must be deferred until run-time. The consequence is that if the savings of code re-use supplied by polymorphism are required, a degree of dynamic type checking is required. However, polymorphism is not required everywhere in a program. There are some parts of the program which can be written to run against some explicitly stated types. What the language should provide, then, is a judicious mixture of static and dynamic type checking, so that the compiler checks any part of the program which it can, while the polymorphic types are left unresolved until run-time.

All of these paradigms exhibit two chief failings. Firstly, in order to provide a good mechanism to achieve one purpose, other purposes are ignored. Secondly, languages tend to be either insufficiently powerful or too complex. The paradigm of Persistent Programming attempts to tackle these issues. The underlying philosophy of the approach is to provide coherence, firstly in that the same mechanism is not used for doing two different things, and secondly that two mechanisms are not used for doing similar things.

For a language to be persistent means that the mechanism for handling long-term and short-term data is unified. Such languages do away with all of the baggage other languages require to store and retrieve data explicitly. A similar unification is also achieved by making a language data-type complete [Morrison, 1982; Tennent, 1981]. This means that all kinds of data can be handled in the same way. Making procedures values and giving them first-class status in this "data type complete" domain allows similar manipulation of program. Such unification of mechanisms and removal of arbitrary exceptions should have a greatly simplifying effect on the task of programming. It is the aim of this research to test whether or not this is so.

2.1.2 The Development of Better Database Systems.

Databases grew out of an attempt to reduce the amount of software that had to be written by factoring out the common elements of data intensive applications. Such facilities as security, concurrent access to data, distribution of data and so on were provided by a program called a Database Management System. The user would then interact with the DBMS via an interface that was easy to use (relative to using a full programming language). The interface was either via a small set of simple high-level languages (the Data Definition Language, the Data Manipulation Language and the Query Language) or, latterly, via graphical tools [Zloof, 1977; Odesta, 1984].

To accommodate a variety of applications being built on top of a single program, this program had to supply a model for the structure to which the data it could handle must conform. Early experience with these "data models" led to the development of the Classical Data Models: the Network, the Hierarchical and the Relational Data Models. The Relational Model [Codd 1970] proved to be a particularly elegant model within which to structure all kinds of data. It took the view that all data could be represented in the form of rectangular tables, with columns that were named and typed and rows which were undistinguished. The mathematical properties of this model led to a great deal of research on how to optimise data storage and retrieval.

Not only was the production of the facilities of the application factored out, but also research into how to optimise them.

Database systems based on the Relational Model began to proliferate [Stonebraker *et al.*, 1976; Oracle, 1983] and standards began to be set. For instance a common query language (SQL) is a provisional standard. Many business applications can be framed in terms of relations and so Relational Databases have become effective and enduring products for reducing the coding effort for simple data intensive applications.

However, many new application areas have opened up for which the Relational Model seems somewhat inadequate. These include Computer-Aided Design and Manufacturer (CAD/CAM), Computer-Aided Software Engineering (CASE), Computer-Aided Engineering (CAE) and Office Automation (OA). These all require the manipulation of objects with complex structures, which may be forced into the relational mould only with difficulty. Manipulating these kinds of objects is intrinsically so conceptually complex that the software producer requires all the help available in conveying the complexity to the computer. An underlying data model which is as simple as the Relational Model enforces a translation process from the real world which imposes an insupportable cognitive load on the software engineer.

[Kent, 1979] provides an excellent analysis of the limitations of the RM from the point of view of someone wishing to represent complex objects. Essentially, he argues that the simple nature of the RM provides two mechanisms for relating two pieces of data: either they are in different fields of the same tuple; or they are in two tuples with a common field. These two mechanisms are each used for a variety of purposes, thus causing semantic overloading. The function of each part of a Relational Schema may not be immediately obvious to someone brought to examine a database set up by someone else. The other limitation of the RM is that it lacks the ability to provide a consist way of describing single objects. Sometimes they are tuples. Sometimes they are whole relations. Sometimes they are distributed over a number of tuples. Sometimes they are just a part of a tuple. In short, the model falls short in expressing the structure of complex objects in a coherent way. Furthermore, there is no support for object identity and reference. Linking any two objects uses one of the two mechanisms above in an ad-hoc way. The most systematic attempt to extend the RM to deal with these points was given in [Codd, 1979], but the mechanisms proposed for providing automatic support for semantic content seem merely to bring the problems into sharper focus.

Therefore, there began to be proposed a number of models with richer structures, which more closely model relationships found in the real world. These **Semantic Data Models** (SDM's) will be discussed in more detail in section 2.2.1, but the growing expressive power that SDM's provide in a database context is noted here. They take the idea of factoring out common facilities and extend this to factor out more of the problem of translating a real-world application into a computer representation.

Thus, within the database context, improved facilities are emerging for expressing data intensive applications as simply and naturally as possible. The development of SDM's greatly eases the description of the structure of the database. However, another restriction imposed by the classical data models is in the expression of the active aspects of a data application and SDM's do not, in general, contribute a solution to this. Database systems tend not to provide the ability to describe the active

aspects of an application. The languages they provide are very simple, rarely computationally complete and weak, in general, at describing data transformations.

This need for more programming power in this area resulted in another trend starting in the mid-70's - the database programming language (DBPL). The essential feature of the development of DBPL's was an attempt to circumvent the lack of expressive power of database systems, by merging them with full programming languages. Section 2.2.3 briefly describes some work in this area, which is intended to give the program developer more flexibility to express the active aspects of the application. The critical problem here is merging two wholly contrasting views on the manipulation of data. This was a central issue in the development of the Persistent Programming paradigm, and PS-algol in particular.

Database systems were an early attempt to factor out whole areas of programs dealing with data. At first they provided an unnaturally passive view of data, although the need to describe the active aspect has long been recognised. An early example of this was the introduction of database procedures in [CODASYL, 1971]. The technology for their convenient provision has only appeared with the advent of the Persistent Programming Language. More recently, more natural ways of describing data have been included along with facilities for describing the active components of a database. Yet better languages are needed for manipulating data. These will push the trends towards simplicity, power and naturalness still further.

2.1.3 Software Engineering Solutions.

One further trend is that of providing better environments in which to develop software. These environments provide a coherent context in which to produce new software modules, and store, retrieve and link them together. Given support of this kind, the manufacture of large software products becomes a much more tractable proposition, compared with using an unstructured environment. However, such environments are, at present, either restricted in the kinds of software they can manage, restricted in the kinds of language they can use, or restricted in their facilities. Many environments grow from a particular context (for instance, X-windows for graphics [Jones, 1989]). Their semantics is therefore independent of and possibly incompatible with the semantics of other environments which are in use.

The key to producing better environments is to view the software itself as data objects being manipulated within the Software Development Environment (SDE). Once this view has been taken, then two consequences emerge. Firstly, if program is data, database technology can be brought to bear on the problem of managing the software. Secondly, to do this languages are required which manage program as if it were data.

Managing software includes a number of tasks. There is the problem of introducing new software modules into the environment, by means of some source code editor, say. New modules must be inserted in a structured way, so that they can be retrieved for subsequent re-use. The process of finding modules which already exist is a second task. Another task is that of managing versions of modules. These can arise for a number of reasons, each of which might require slightly different handling. Finally, there is a need for a mechanism to configure modules into a final product. These mechanisms are described in more detail in Chapter 8.

These tasks are all database tasks, provided only that program can be viewed as data and the software environment as a database. In order to produce good SDEs, therefore, database languages capable of supporting this view are required.

2.2 Some Relevant Approaches.

2.2.1 The Semantic Data Modelling Approach.

Section 2.1.2 introduced the notion of the data model and the limitations of the classical data models, which led to the development of models which strive to capture more of the meaning of the application. The best summary of work on Semantic Data Models is the survey in [Hull and King, 1987], to which the reader is referred for greater detail. There appear also such terms as "Conceptual Data Models" [Brodie *et al.*, 1984], but there seems to be no difference between conceptual and semantic data models.

In essence, a Semantic Data Model (SDM) includes constructs which mirror different kinds of concepts in the real world. These may include:

entities: the objects which are to be modelled;

identity: these objects will have a fixed representation, which can be referred to from any number of other objects, whilst encompassing the same set of values;

entity types or **classes**: groups of objects with common properties;

attributes: a dependent value of an object;

components: an object which is part of another object;

relationships: a link between two or more entities;

constraints and **assertions**: statements which are invariant about the modelled world;

activities, processes or **events**: descriptions of the way the system reacts to events;

and **exceptions** - descriptions of rare deviations from constraints and processes.

Any given SDM will provide a subset of the above modelling constructs and the application designer frames the design in terms of these. The design is constructed by use of some Data Description Language, which may very well be graphical, and this will be transformed into a lower-level description, such as a relational one. The central idea behind SDM's is that the description will be more naturally related to the real-world application.

Several questions come to mind when surveying SDM's. Most crucially comes the question of how many different kinds of constructs are provided. Some models, for instance the FDM (see below section 2.2.1.4), model everything in terms of one

main construct (in this case the function), while others, such as the Semantic Data Model of Hammer and McLeod (see section 2.2.1.3), provide a great variety and intricacy of constructs. In the one case, simple models may be built, although one mechanism does duty for a number of meanings (i.e. there is semantic overloading). In the other case, a great deal of meaningful detail can be built into a model, but the modelling language may be much less easy to use. The question is whether or not more means better, or do more constructs just mean a confusion of alternatives in the modelling process. Just one illustration of this problem concerns the nature of attributes and components in the list above. Consider an address. It has dependent properties house number, street, etc. Are these attributes or components? Does it matter? Will people be confused if they can represent them as either? Does the choice affect the underlying implementation anyway?

A second question is whether the model is able to discriminate between different kinds of entity type. This question breaks down into a positive and a negative aspect. Can the differences between entity types be specified sufficiently? Are certain kinds of entity unavailable in certain parts of the model? The kind of distinction to be made between entity types separates basic or printable entity types, such as integers or strings, from complex entity types. Then complex entity types may be divided into primary types and subordinate ones. These distinguish the central object types of the system from those which are dependent on them. Thus a university database might include a primary entity type for people and secondary sub-types for staff members and students.

The question then is whether a given model provides these distinctions. The ER model (section 2.2.1.2) has just such distinctions (between strong and weak entity sets), but also specifies the considerable restriction that attributes must be printable. A subsidiary question is whether such a taxonomy of entity types provides any modelling value or is it just a carry over from implementation detail which only serves to confuse?

Another question concerns the nature of relationships between objects. What kinds of relationships are provided? Attribution and aggregation (the component relationship) have already been mentioned. Most models also include what is variously described as subtyping, specialisation, inheritance, "IS-A", etc. The precise notion involved, however, varies from model to model, so that a great number of different concepts may be clustered together under the one umbrella, partly because the same concept is being used for a number of purposes [Atkinson, 1988].

Two other questions concern derived data and meta-data description. Is there the ability to describe inter-component structure so that data does not have to be entered or stored more than once, but derivation rules can be entered instead? Can the model describe itself? If so, meta-data can be modelled in the same way as ordinary data, which means both that the model is conceptually simplified and that the same facilities can do double duty in manipulating data and schema.

Most SDM's concentrate purely on a passive description of the database. Some (such as the Event model described in section 2.2.1.7) also include an active component. These allow a straightforward description of processes with which the database can be changed. Although much of the modelling ideas of SDM's were incorporated into the Object-Oriented approach, the notion of freely describing process objects was avoided, as will be discussed in section 2.2.2.

2.2.1.1 The Semantic Binary Data Model.

The first Semantic Data Model intended for databases was the Semantic Binary Data Model proposed by Abrial [Abrial, 1974]. This was intended as a design tool for relational databases, but introduced constructs for describing entity types and binary relationships between entities. It takes an extensible view of database design, in which statements are made that there are "categories" of object and then that objects in these categories may be inter-related in particular ways. Thus a start of a database design may look like:

PERSON = **CATEGORY**

introduces a new entity type

PERSONNAME = **RELATION**(*PERSON*, *NAME*, *has_name*, *names*)

introduces two relationships: *has_name* from *PERSON* to *NAME* and *names* from *NAME* to *PERSON*.

PERSONADDRESS = **RELATION**(*PERSON*, *ADDRESS*,
lives_at = **AFN**(1, 1), *residents_of* = **AFN**(0, ∞))
introduces **cardinality constraints** on the relationships, in this case all people have exactly one address, while any number from 0 to ∞ may live at a given address.

All database design is then carried out with these two constructs, although there is also a piece of syntax that allows attributes, called properties, to be described which is merely a short-hand for describing an attribution relationship.

There is also a language for performing data manipulation and querying. Data manipulation proceeds by generating instances of categories and then creating relationship links between the categories. Querying is performed by providing conjectures in the form of predicates and validating them against the database. The most significant feature of this model is that programs can be built with these languages and such programs can be used to model constraints or to perform activities. In particular, the SBDM provides the ability to specify an activity which is to occur when an object is created. This idea became the *when_created* operation common to most Object-Oriented data models. The SBDM was, then, a very simple modelling system, but it set the precedent for modelling real-world notions directly.

2.2.1.2 The Entity Relationship Model.

This model was introduced in [Chen, 1976] and was the model which first popularised the Semantic Data Modelling concept. It remains the most popular data model. It too was created to be an off-line graphical design tool for relational systems and is similar to the Bachman diagrams previously proposed for CODASYL databases [Bachman, 1969]. It may be viewed as an extension of the SBDM above in modelling power in that there are now: **entity sets** which are equivalent to categories; **attributes** are allowed on entity sets; and **relationship sets** which interconnect entity sets. There is also the ability to model different kinds of entity type. Primary types are those considered central to the definition of the data, have primary keys and are known as

strong entity sets. Subordinate entity types derive their key from some primary type and are known as **weak entity sets**. All of this is easily transformed into a relational database schema.

Figure 2.1 shows an example (drawn from [Korth and Silberschatz, 1986]), in which there are three strong entity sets, for the *customer*, the *account* and the *branch*, interconnected by the relationship *CAB*. There is also a weak entity set, *transaction*, which depends on *account* connected by the relationship, *log*. All of the entity sets have attributes.

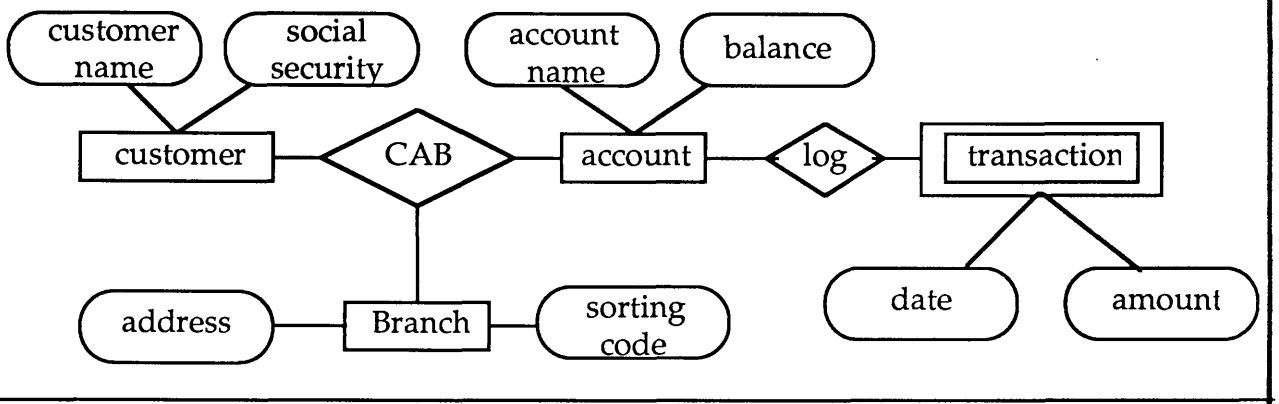


Figure 2.1 An Entity Relationship Diagram.

The ER model thus provides a somewhat richer modelling environment for describing the passive structure of the basic objects in a database. It is easily mastered, but does not add much depth to the description.

2.2.1.3 The Semantic Data Model.

Other passive data models centred around the notions of entity types and relationships were proposed, of which the richest was the Semantic Data Model [Hammer and McLeod, 1981]. Its basic modelling construct for entity types is the **class**, which is defined to have:

a name;
a set of members;
a description for documentation purposes;
a set of **member attributes** (i.e. defined on each member, e.g. *age*);
a set of **class attributes** (i.e. defined on the class as a whole, e.g. *cardinality*);
whether it is a **base-class** (i.e. is it defined independently of others) or not;
if it is a base-class, the set of attributes which form the **key**;
does it contain duplicates or not?

Non- base classes are defined either by **sub-classing**, which is done by one of:

a filtering membership predicate;
the intersection of two classes;
the range of some attribute on another class
e.g. if *PERSON* has an attribute *age*: *integer* then the class of *integers* that are *ages* for some *PERSON* can be specified.

user-defined (the user will explicitly select objects from the super-class to go in the sub-class).

or by **grouping**, in which a partitioning expression splits a class into a group of sub-classes.

Attributes can be of a large number of kinds: single-valued or multi-valued; mandatory or optional; changeable or not; exhaustive (i.e. every value in the attribute range must be the value of at least one object); non-overlapping (i.e.unique); or derived (by some expression from other attributes).

There are many kinds of derived data and constraints and there are inheritance mechanisms between the classes. In short, this is a modelling world which is very rich indeed. The paper ends with a sample database of ships and inspections, which seems to capture a considerable number of the facts one would know about the data structure of the modelled world. The authors claim that the model has been used with success in designing applications. The problem with such data models is defining and understanding their semantics.

2.2.1.4 The Functional Data Model.

The Functional Data Model [Shipman, 1981] goes to the other extreme in providing a single modelling construct for the whole job of data modelling. Using functions, one can model: entity types, attributes, sub-typing, relationships, multi-valued and single-valued data, base- and derived data and meta-data.

Entity types are modelled by functions which have no arguments which return the set of values of that type. There is one top type, called *ENTITY*, and other types are declared as in:

DECLARE *PERSON*() ->> *ENTITY*
DECLARE *STUDENT*() ->> *PERSON*

which creates a new primary type *PERSON* as well as *STUDENT*, a sub-type of *PERSON*. Inheritance from *PERSON* to *STUDENT* then occurs automatically. Note that there is semantic over-loading of the type names - they mean both the type and the function which returns the values of the type. All entity types hold single valued entities and so there is no such thing in the model as multi-valued types.

Attributes are also defined as functions, such as

DECLARE *NAME*(*PERSON*) -> *STRING*
DECLARE *COURSES*(*STUDENT*) ->> *COURSE*
DECLARE *GRADE*(*STUDENT, COURSE*) ->> *INTEGER*

in which a single-valued attribute *NAME* has been defined on *PERSON* and a multi-valued (two-headed arrow) attribute *COURSES* has been defined on *STUDENT*. Points to note include: the existence of predefined types, *STRING*, etc for the printable types; the fact that *COURSE* does not have to be defined before *COURSES*; there is freedom to re-use names so that the same named attribute can be defined on two different types and the system will resolve the overloading by using the type; multi-argument functions are allowed.

Derived attributes can be defined by using constructs which are essentially functionals (functions which map functions into functions) such as *INVERSE OF* and *TRANSITIVE CLOSURE OF*; or by function composition; or by aggregating functions such as *AVERAGE*. **Derived entity types** can also be defined by using functionals which mimic aggregation and set union and intersection.

Shipman also provides DAPLEX, a data manipulation and querying language. The language stands up reasonably well as a query language, with queries such as:

```
FOR EACH STUDENT SUCH THAT FOR SOME COURSES( STUDENT )
    SUCH THAT NAME( LECTURER( COURSES ) ) = "Richard"
        PRINT NAME( STUDENT )
```

which have a fairly natural language feel to them. The DML, although consistent with this, seems overly verbose, on the other hand:

```
FOR A NEW STUDENT
    BEGIN
        LET NAME( STUDENT ) = "Bill"
        LET DEPT( STUDENT ) = THE DEPARTMENT
            SUCH THAT NAME( DEPARTMENT ) = "CS"
    END
```

is too much code for the job. Note one more use for the variable *STUDENT*, which now means all of an entity type, the function which returns the values of that type and a variable that ranges over instances of that type.

One of the strong features of the FDM is the ability to model the meta-data within the model itself. There is an entity type called *FUNCTION* and attributes of this type to hold the name, arguments, result type, etc. of functions. The schema is thus manipulable by the DML and queryable by the QL.

The FDM is the strongest example of a simple modelling system with great power. There is a feeling when trying to use it, however, that a lot of the semantic content of the database is lost. When looking at a schema, it is not always clear what role a given function is playing, nor is it obvious whether the semantic overloading of names in the system is simplifying or complicating. An implementation of this model is described in section 7.1.

A separate functional approach is FQL [Buneman and Nikhil, 1984] in which a purely functional language for the description of database schemas and queries is proposed. It provides **functionals** for describing sets, aggregates and attributes and permits simple queries to be built using them. Intended as a front-end to relational databases, it uses the functional style of lazy evaluation to reduce data access time and presumably will carry with it the formal properties which are the *raison d'être* of the functional paradigm. As such, it is a particular elegant example of the coming together of two approaches. One implementation of FQL was carried out in the context of the work on RAQUEL systems described in section 6.1.

2.2.1.5 TAXIS.

TAXIS is a system which is designed for the creation of interactive information systems [Mylopoulous *et al.*, 1980]. It resembles the FDM in that everything is modelled in terms of one construct, this time the class. Classes are used to model passive objects and active objects, like transactions, constraints, exceptions and even expressions. There is a two-way taxonomy of objects - an inheritance hierarchy, in which any kind of object can take part, and division into **tokens** (or entities) which are grouped into **classes** (or types), which are grouped into **meta-classes** (cf Cardelli's kinds).

Properties are defined on tokens, e.g. (*john_smith, has_name,* "JOHN SMITH"), on classes < *PERSON, has_name, PERSON_NAME* > and on meta-classes <*PERSON_CLASS, average_age, AGE_VALUE* >. The first of these represents a single fact from the database, the second a function from the *PERSON* class to the *PERSON_NAME* class, while the third represents a function from a collection of classes to the *AGE_VALUE* class.

The TAXIS language allows a fairly straightforward description of types and properties, which resembles the SDM. Classes are defined as instances of meta-classes, which must therefore be defined first, as in:

metaclass *PERSON_CLASS* **with**
    attribute_properties
        average_age: *AGE_VALUE*
end

Then a class description can be written:

PERSON_CLASS PERSON **with**
    keys: *person_id*: (*name, address*)
    characteristics:
        name: *PERSON_NAME*
        address: *ADDRESS_VALUE*
        phone#: *PHONE_VALUE*
    attribute_properties:
        age: *AGE_VALUE*
        sex: *SEX_VALUE*
end

which introduces *PERSON*, with five properties, three invariant (characteristics) and two variable.

There are some system-defined metaclasses for particular types of class. These include: *VARIABLE_CLASS*, whose members are classes which support insertion and deletion of members; *FINITELY_DEFINED*, whose members are classes whose instances are explicitly listed; *TEST_DEFINED*, whose members are classes whose instances are determined by a predicate; *AGGREGATE_CLASS*, whose members are classes which are aggregates of other classes; and *FORMATTED_CLASS*, whose members are classes of *STRINGS* with a common format. Using these meta-classes and inheritance, classes can be created with different features. As defined above, *PERSON_CLASS* classes would not support insertion and deletion. To achieve this the following should be specified:

metaclass *PERSON_CLASS* **is-a** *VARIABLE_CLASS* **with ....**

The inheritance hierarchy extends throughout the class and metaclass network and there are specially defined classes *ANY* and *NONE*, such that for any class, *X*, *X* isa *ANY* and *NONE* isa *X*. There are equivalent metaclasses, *ANY_CLASS* and *NO_CLASS*, as well as other classes such as *ANY_VARIABLE*, etc.

There is a special metaclass called *TRANSACTION_CLASS* which contains class objects which are not really sets of tokens, but are essentially program objects. An example of this is

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

which is a full specification of the input parameters and local variables, a precondition for the transaction to execute smoothly and then a list of sub-actions which constitute the transaction's behaviour. These are specified in a QUEL-like language.

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

TAXIS is a very consistent, coherent system which, like FDM, uses one mechanism, this time inheritance over classes, to model all sorts of things. It has some limitations and some things, like exceptions, feel a bit unnatural to use. However, it is a very clear specification language. The designers have been ambivalent about whether or not it should be compilable. An early effort was made to compile it to PASCAL/R. Attempts have been made to provide toolsets for Taxis and its derivatives and to use it in conjunction with other languages [Borgida *et al.*, 1989]. Another drawback may be the three-level world of tokens, classes and meta-classes which seems to violate a basic principle of Computer Science, which is that there should be 0, 1 or an infinite number of anything. It may be that there are systems whose optimal data model includes meta-meta-classes, for instance.

2.2.1.6 The IFO Data Model.

This model proposed in [Abiteboul and Hull, 1988] attempts to bring a common and precisely defined framework to Semantic Data Models. It is also used to place a framework for the analysis of updates, which is beyond the range of this survey. It provides a sophisticated taxonomy of types, which will be discussed with reference to the following diagram given as Figure 2.2.

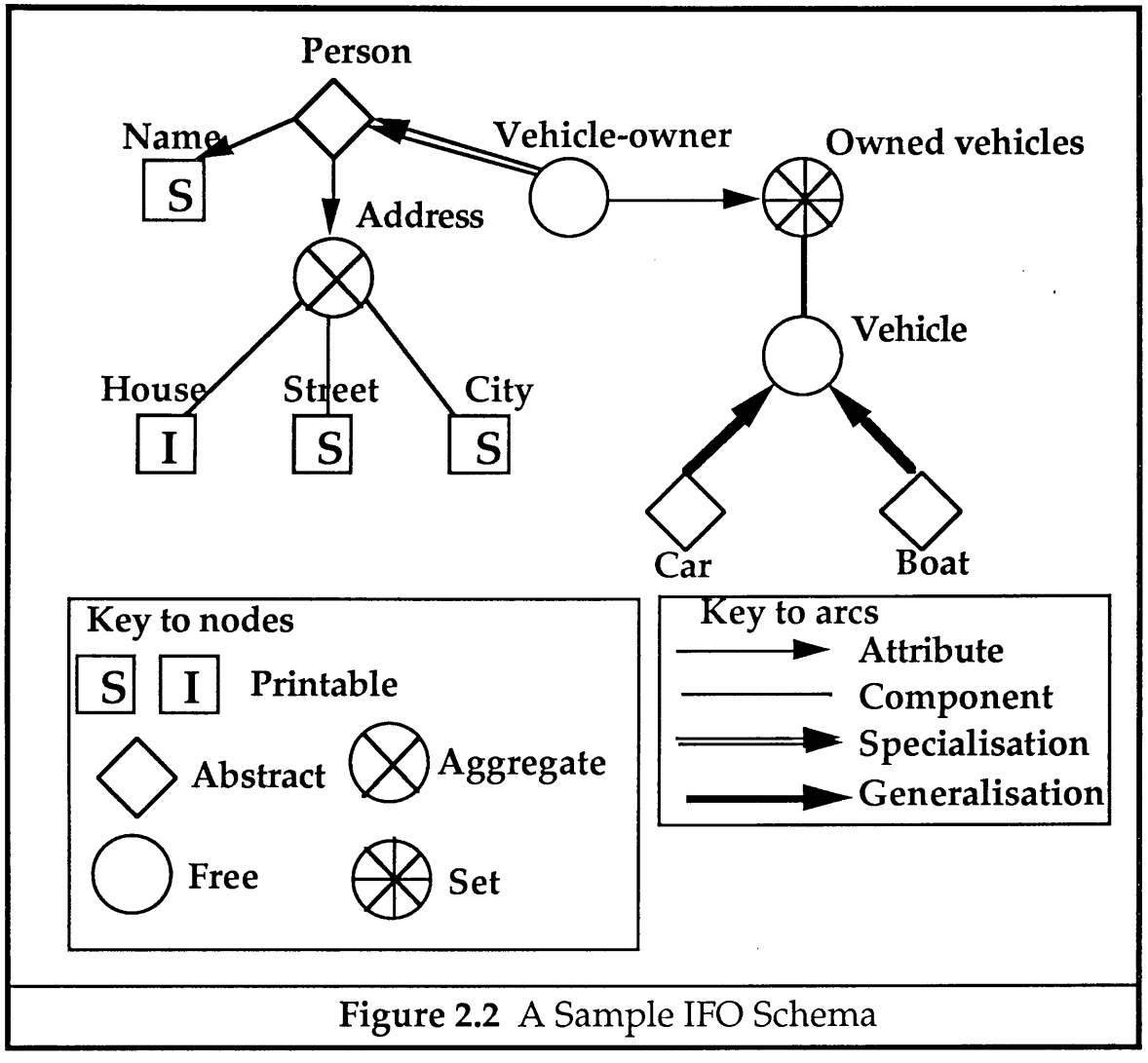


Figure 2.2 A Sample IFO Schema

Firstly, there are three kinds of **atomic** type: **printable** types, which are the usual base types, string, integer, etc.; **abstract** types, which represent the basic entities the schema is modelling, and have "no underlying structure"; and **free** types, which are defined with reference to other types (sub-types are an example of this). There are also the following complex type constructors: **set** or **collection** creates a multi-valued object; while **aggregate** creates single objects out of component parts (address in the example). Using these constructs, the space of objects can be thought of as being partitioned into sets, each of which is controlled by one of the abstract types.

The entity types are connected by a number of different **relationships**. A **fragment** is any graph of types and relationships and a **schema** is the complete graph representing the model. The relationships available in the model are **attribution** (*X* is an attribute of *Y*); **component** (*X* is a **part** of aggregate type *Y*); or the sub-typing relationships **specialisation** and **generalisation**. Specialisation represents the notion that type *X* is a sub-type of *Y* in the senses that *X* inherits the properties of *Y* and all *X*'s

are also *Y*'s. The specialised type, *X*, will be a free type, since it is defined relative to the type being specialised. Generalisation, on the other hand, creates a free type which is the super-type of a set of other types. It encompasses the notion that every instance of this type must also be an instance of one of the sub-types (thus vehicle is constructed out of car and boat).

One bonus of this model is that it allows the definition of relationships between relationships, by nesting. For instance, in the FDM, to represent the grade of a student on a course, a two argument function is needed - *GRADE(COURSE,STUDENT)-> STRING*. In IFO, this can be represented by an attribute of *COURSE*, which contains a set of students, and then an attribute of the members of this set which returns the grade. This better models the notions involved, whereas the FDM representation essentially loses the order between the concepts.

IFO allows some local **constraints** to be specified, such as that a relationship is 1:1, etc. It also allows constraints on specialisation relationships - that the subtypes of *X* must be **disjoint** or that they **cover** *X*. There are also global constraints which are enforced by the system: the sub-type graph must be acyclic; no type can be the specialisation of more than one atomic type; no free type can have been created simultaneously by generalisation and specialisation. The first of these is intuitively obvious - it makes little sense to say *X* is a sub-type of *Y*, which is a sub-type of *Z*, which is a sub-type of *X*. If the sub-typing (or subsetting) relationship adds more information at every stage, *X* now has more information than *X*! The second global constraint means that inheritance from more than one super-type is acceptable as long as these both eventually inherit from the same atomic type. This makes some sense in that whereas a type which specialises both *STUDENT* and *STAFF*, themselves both specialisations of *PERSON*, seems reasonable, a type which specialises both *SHIP* and *PERSON* does not. (Note however that the type of potential vehicle owners may be the sub-type of both *PERSON* and *COMPANY*.) The third constraint is a consequence of the system. Free types are either created by specialisation or by generalisation and to do both at once suggests that a type gets its defining information from two possibly conflicting sources. Thus *VEHICLE* may be created as a specialisation of *MOVING THING* or as a generalisation of *BOAT, PLANE* and *CAR*, but not both at once. If the model must capture all this, either *MOVING THING* is a generalisation of *VEHICLE* (and other types) or *CAR*, etc are specialisations of *VEHICLE*.

IFO places the concepts which were introduced in the preceding models into a relatively simple framework in which the intrinsic nature of the constructs is revealed. This allows the first step to be made towards analysing data models formally [Abiteboul and Hull, 1988]. Section 7.3 describes an implementation of this model.

2.2.1.7 The Event Model.

The next model incorporates active object descriptions into the data model. The Event Model [King and McLeod, 1984] is designed to model a database by making statements about the database which are true for all time. To do so, it needs some notion of active objects and then it can make statements like "Event E modified object O at time T".

There are two types of passive object in the system: **descriptor** objects are strings and hold identifiers and printable values; **abstract** objects are complex objects. The latter consist of **attributes**, of which one (the **primary** attribute) uniquely defines the

object. Attributes are modelled by functions and can be specified to be unique, single-valued, non-null, exhaustive or the inverse of another attribute. Objects may be sub-typed, using restricting predicates or adding more attributes. Some examples of the definition of object types:

Type: *Correspondence*
primary attributes: *ID from Correspondence-ID-#s*
(single-valued, non-null)
dependent attributes: *Kind* **from** *Correspondence-Kinds* **(single-valued)**
subtype: *Bills*
all *Correspondence* **where** *Kind* = "Bill"
subtype: *Requests*
all *Correspondence* **where** *Kind* = "Request"

Events are divided into **application** events (which model transactions) and **perusal** events (which model queries). They may be parameterised and require the specification of objects ("working subtypes") to be used in the event and the sub-actions involved. An example:

Type: *Process-Correspondence*
parameters: *Item* **from** *Correspondence-ID-#s*
working subtypes: *P* **is** *Correspondence* **where** *Correspondence = Item*
actions: **if** *P.Kind* = "Request"
then (*Select-Form*(*P*), *Complete-Form*(*P*))
else if *P.Kind* = "Bill" **then**
(*Perform-Account-Fcns*(*P*), *Write-Cheque*(*P*)),
Transmit-Response(*P*),
Archive-Request / Bill-and-Response

Given these kinds of object, the goal of this work is to provide a graphical tool for producing a "design schema" of the database and to map this automatically into the textual descriptions described above, "the conceptual schema". There will be a set of **modelling** events which do this translation.

2.2.1.8 Summary.

This has been a very quick tour through a number of data modelling systems. There now follows a short discussion on their usefulness.

Models which capture more of the semantic information of the application than do the classical models are essential to facilitate the creation of complex database applications by non-specialist application implementers. There is an array of constructs they may be given to achieve this. From this work should emerge a data model which is capable of expressing entity types and relationships between then which at least include attribution, aggregation, grouping and sub-typing. The system should optionally provide some of the richness in modifying these basic constructs as shown by the SDM. It is not so obvious that there is any **modelling** value in providing a taxonomy of entity types into printable, primary complex and subsidiary complex, although the system might infer these kinds and then implement them in different ways. These constructs should be adequate for specifying the data structure of the database.

However, there is an increasing need to use the same modelling tool to describe the active properties of the database. For instance, the worlds of CAD, CAM, CASE and Office Automation increasingly need the ability to describe "active" objects in the same way as the passive objects. Models, such as the Event Model, may be thought of as providing a step in this direction. They are to be compared with Object-Oriented Database, in that OODBs typically tie the active component to the passive component, which in an overall model of some active system may not be an adequate description of the real world activity.

2.2.2 The Object-Oriented Approach.

The Object-Oriented approach takes the ideas of Semantic Data Modelling and puts them into a structure within which the passive and active components of an application area can be described. The origins of the Object-Oriented approach lie variously in the domains of programming languages [Dahl and Nygard, 1966], database systems [e.g. Smith and Smith, 1977] and artificial intelligence [e.g. Hewitt *et al.*, 1973]. There are a number of good short surveys of the approach (for instance [Stefik and Bobrow, 1985] and [Bancilhon, 1988]). However, the best exposition of the approach is [Meyer, 1988].

Meyer develops the motivation behind the O-O approach and derives the following definition:

> Object-Oriented design is the construction of software systems as structured collections of abstract data type implementations.

This definition demonstrates at once the strengths and weaknesses of the model. There is the modular construction of a system in a highly structured way, but the structure imposes a straight jacket in the form of the abstract data type.

To extend this definition, a system will usually be thought of as Object-Oriented if it exhibits the following features:

classification - the division of all data values into sets, called classes, with common structure and behaviour;

identity - the values associated with a given object will be collected together and manipulated as a single unit;

reference - this unit may be referred to from any other object and any change to the constituents of the unit will be visible to all these references;

controlled naming - all naming will occur relative to the objects;

encapsulation - the grouping together of the descriptions of the data structure and behavioural aspects of a class - this often has a further implication that the data structure is hidden and the only way of using an object is via a set of operations which are made publicly available;

sub-typing - the ability to describe one class as a being a more specialised form of one or more other classes;

inheritance - the automatic availability of the definition of one class, to any of its sub-classes;

overriding - the ability to replace inherited definitions by sub-class specific ones;

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

These will be described in more detail in the following sections.

2.2.2.1 Simula - a first step towards Object-Orientation.

The first language to exhibit properties later associated with Object-Oriented languages was Simula 67, an extension of algol developed by Dahl and Nygard [Dahl and Nygard, 1966], principally as a language for discrete event simulation. The language has many features, but the interest here is primarily in the class constructor, first seen in this language.

All objects in Simula are either of basic algol types, such as real or integer, or are instances of classes. Classes are defined as in:

```
Shape class Polygon( n ); integer n
        virtual:        procedure setVertices;
        begin
                integer indexNumber
                ref (Point ) centroid;
                procedure scale;
                        begin
                                .... appropriate code body
                        end
        end
```

This class definition contains a number of different components. Firstly comes a name for the class, *Polygon*, a class which this is a sub-class of, *Shape*, and a specification of parameter values which must be provided when an instance is created, in this case *n* the order of the polygon. Secondly there are some virtual or deferred procedures. These are specifications only and defer an implementation until a sub-class is defined. In this case the *setVertices* procedure will be specified within the definitions of subclasses of *Polygon*, such as *Triangle*. Thirdly come some local state variables. Finally there are some fully defined procedures. Instances of this class can be produced by the following lines which declare and instantiate a Polygon variable:

```
ref( Polygon ) p;
p :- new Polygon( 5 );
```

Notice the syntax, which distinguishes assignment to a complex object (":-") and to a basic value (":=" as usual). This distinction was dropped in later languages as it was found to be unhelpful.

Object properties and class procedures are accessed by using the dot notation as in:

c :- p.centroid

and *p.scale*

The variable *p* can now be used wherever a *Polygon* is allowed. Moreover, it may also be used wherever a *Shape* is allowed. Thus a degree of polymorphism is introduced through subclassing. Moreover, inheritance also obtains, in that any procedure defined on *Shapes* is also available to *Polygons*. Finally, a notion of deferred binding is available, in that if versions of the *scale* procedure on *Shape* and *Polygon* are defined, then an instance declared on *Polygon* will automatically use the *scale* for polygons. An object which is actually a *Polygon*, but which is declared as *Shape* will usually use the *scale* of *Shape*, but may use the *scale* of *Polygon* by doing:

ref (*Shape*) *s*
(*s* **qua** *Polygon*).*scale*

Simula thus introduces most of the range of Object-Oriented concepts: classes, inheritance, encapsulation and deferred binding.

2.2.2.2 Smalltalk.

Smalltalk [Goldberg and Robson, 1983] was developed at Xerox by a group led by Kay, Goldberg and Ingalls, and was influenced by Simula, but used the same concepts in a dynamically typed system akin to Lisp. The critical difference between this language and other O-O languages is that there is no static type checking. A check is made that a given operation can actually be run against a given object each time the operation is applied. This violates the requirement that type errors, the dominant programming error, should be detected at the earliest possible time. An example of the way in which this hurts the programmer will be given shortly. Another, less significant, drawback to Smalltalk is that the language has freely introduced neologisms for concepts which have already got perfectly acceptable names. Thus there are *methods* instead of procedures or operations; and *sending a message* instead of applying a procedure. These terms will be ignored in this discussion, as they only serve to confuse.

On the other hand, the significant contribution of Smalltalk is to frame everything in terms of objects. Sometimes some sleight of hand has to be performed behind the scenes to achieve this, but this brings considerable conceptual simplicity. It has also been implemented in an interpretive way that throws the class hierarchy open for browsing and modification at run-time. This is a great aid to program debugging. Finally the language adds, in a way that is reminiscent of TAXIS, the notion of a meta-class, within which it is possible to describe class operations in the same way as instance operations.

Classes are defined in Smalltalk in very much the same way as in Simula. However, Smalltalk takes the view that classes themselves are objects and this simplifying concept means that operations can be defined on the class of classes, *Class*, which means that instance-specific and class-specific operations can be described in the

same framework. For instance, there is a class operation, *new*, which creates a new instance of whichever class it is applied to. Instance creation is therefore written:

p -> Polygon new

in which *Polygon* refers to a class object and the *new* means apply its *new* operation.

Further distinctions between Smalltalk and Simula are that Smalltalk takes the view that all state variables are hidden and that only operations may be public and that instance operations are defined relative to a local variable called *self*. Operations to change the state are defined using the *self* variable and then these are the only way of manipulating the state data. Finally Smalltalk only provides single inheritance.

There are a number of inelegancies, which are general to the O-O approach. Firstly the representation of dyadic operations. These appear as in:

2 add: 3

where the operation *add* of the integer is applied. This then picks up the 3, produces the 5 and then returns it. The expression of addition as a property of a single integer, instead of as an operation which takes two integers and produces a third seems extremely cumbersome and unnatural.

Secondly, the unavailability of the local state, whilst often being desirable, seems equally often to be a constraint on programming style which leads to overly verbose and unnatural code. Again, the natural notion is to manipulate an attribute of an object directly and not via a procedure call.

The problem with the lack of typing is illustrated by the following Smalltalk operation which is defined on a class of strings and returns the length of the string.

```
method length
    begin
        length -> 0
        ... iterate through string and add to length
        ↑ length
    end
```

The final clause, with the ↑*length*, returns the value of *length*, an integer. If the programmer omits the ↑, then the operation returns *self*, in this case a string. This has introduced not only a logical error, but also what in most languages would be a type error. Not in Smalltalk, though. This will compile correctly and run, giving a very strange run-time error which, without the excellent debugging tools in the system, would be difficult to track down.

Smalltalk therefore falls down in its lack of typing. This is compensated for to a large degree by a sophisticated software development environment. This provides templates for creating classes and operations. It also provides a debugger for examining the structure of the program and a sophisticated set of system-defined classes which enable the rapid construction of such aspects of the program as the user interface. Therefore the language has been used as a starting point for a range of

remarkable work, including the GemStone O-O Database System (see section 2.2.2.6) and the Alternative Reality Kit [Smith, 1987].

It is interesting to speculate what could have been achieved if the same effort in developing software development environments and methodologies had been expended on other languages. This thesis presents an investigation of these matters for the language PS-algol which is very small in relation to the investment in Smalltalk.

2.2.2.3 C Extensions.

The other language which it is fashionable to extend in an O-O way is C. Three such extensions are briefly mentioned:

C++ was designed by Bjorne Stroustrup of AT&T [Stroustrup 1984]. It introduces the notion of classes into C and cleans up the language somewhat. C++ provides complete encapsulation, although some of the operations in the interface may be declared to be friend operations. This means that they take the object they will operate on as an extra parameter. The language also provides a single inheritance hierarchy and virtual operations, like Simula.

Objective C was produced by Brad Cox [Cox 1986] and is a kind of marriage of C and Smalltalk. It provides the same sort of polymorphism and dynamic binding. The language remains typed, but all complex objects are declared to be of the same type, *ID*. This is similar to the **pntr** type of PS-algol.

E [Richardson and Carey, 1987] is an extension of C++ to assist in the implementation of database systems (not applications). It adds a subsidiary kind of class, called the **dbclass**,with which implementation details, buffering and pointer control can be added. It furthers adds persistence to the language, by a special class kind called a **file**. Finally, it adds a notion of generic classes, which were derived from CLU [Liskov *et al.*, 1977]. This leads to a rich language, with sufficient low level detail to permit the efficient implementation of database systems.

The response to all of these languages depends largely on one's view of C, itself, a language whose usefulness has been largely due to the slowness of hardware on which UNIX systems were originally supplied. Given improved hardware with novel architectures, there is no reason to believe that languages which encourage the specification of low-level detail will survive. The inefficiency which will count increasingly will be that of the production of software and not of its run-time speed.

2.2.2.4 Eiffel.

Eiffel [Meyer 1988] is an attempt to pull together the best features of the above languages with modern concepts of software engineering and as such would seem to be the O-O language of choice. It incorporates the typed class world of Simula within a much simpler architecture similar to Smalltalk. It includes the following features:

Strong static typing - the case for this has been made above and Eiffel demonstrates that strong typing and object-oriented programming can fit well together.

Access to any "exported" **operations and attributes** of a class via the dot notation - note that this means that in requesting a feature of an object, there is no way of telling whether it is an attribute or an operation.

Assertions - any operation may have pre- and post-conditions specified for it, while a class may have invariants specified. The inclusion of assertions in this way greatly enhances the probable correctness of class descriptions.

Exceptions - provided in a slightly different way from CLU or PS-algol. If an operation fails then a **retry** clause is executed to try to patch things up - if this fails or does not appear then the exception is transmitted to the calling operation.

Genericity - classes with type parameters may be specified, such as *STACK OF [T]*, meaning stack of unknown type. Such classes are instantiated by supplying a type in place of the type parameter.

Multiple inheritance with name clashes resolved by renaming.

Dynamic binding and **feature overriding** - so that any feature may be respecified in a sub-class and the implementation of the feature for a given object will be determined at run-time.

Deferred classes - the inheritance mechanism is extended to include a special form of the subtyping relationship. Some of the features of a given class may be specified to be **deferred** - i.e. implementations of this feature will only appear in subclasses. Such classes may not have direct instances - all instances must appear only in a subclass having implementations of any deferred features. Thus there may be a *VEHICLE* class with a deferred operation, *register*, which is implemented in different ways for the sub-classes, *BOAT* and *CAR*. Note that this is a similar notion to the generalisation in IFO and has similar modelling value. The particular value for software engineering is that classes may be described at a high-level as deferred classes, with implementations being left to a later stage.

In short, the language seems to strip away a lot of the surface weaknesses of O-O languages and reveals the critical one: the limitations upon the ways in which active objects can be expressed, i.e. the lack of first-class procedures.

2.2.2.5 Object-Oriented Database Systems.

The development of Smalltalk and other O-O languages has led to several attempts to marry together database and O-O technology. The systems described have been implemented with a varying degree of success.

GemStone/Opal [Maier *et al*, 1986, ServioLogic 1987]. This system is in effect a typed, persistent form of Smalltalk - the language Opal is in many ways indistinguishable from Smalltalk. The environment however provides data management, security mechanisms, concurrent access and database browsers. It also provides interfaces to programs written in C or Smalltalk.

V-base [Ontologic 1986]. This system was thought to be the one which typified the best aspects of OODBs and was regarded as the most likely to produce a commercial

success. Yet the product was withdrawn and replaced by a system which is a back-end to C++. The system provided two languages. The Type Definition Language is used to describe the structure of the database. This includes a description of attributes and specifications of operations, triggers, etc. The implementation of these "active" objects is then produced in a quite separate language, COP - yet another extension to C. The product ran into two problems of user-acceptance. Firstly, TDL and COP were new languages, which brought the customary problems to potential buyers (particularly as there were two and not one new language). Secondly, the product was extremely slow and, as would be expected from a prototype product, somewhat unreliable.

Trellis/Owl [O'Brien *et al*, 1987]. This database system was designed at Digital Equipment Corporation and initially had high research visibility. Recently that has reduced, which may mean that the company are planning to market it. The system is in some ways the database equivalent of Eiffel as it offers multiple inheritance, overriding and static type checking and also has exceptions, although this time in the CLU style. It adds persistence in a manner similar to PS-algol. There is a distinguished class called *DB_COLLECTION*, which has the operations *insert*, *remove*, *elements* (return all the elements) and *select* (return all elements for which some predicate is true). Finally it includes concurrent access by a sharing-by-copy mechanism. That is, a user wishing to change an object checks it out, changes it and checks it back in. While the object is checked out, the old copy is still available for reading. The system is provided in the form of a single, well-engineered language.

O_2 [Lécluse *et al.*, 1988] is an Object-Oriented Database System produced by Altaïr. Programs in O_2 describe classes in a slightly different way from the foregoing, for example:

```
new_type CAR is
  { supertype VEHICLE
    structure tupleof (noWheels: integer r; capacity: integer r; fuel: integer rw)
    methods
      fillup
        begin
            self.fuel := self.capacity
        end
    persist as Car }
```

In this specification, *CAR* is defined to be a sub-type of *VEHICLE*; with the additional information given in the form of a tuple. The attributes *noWheels* and *capacity* will have read operations automatically created for them, while *fuel* will have read and write operations created. There is a set of operations (in this case only *fillup*) followed by a name under which the class will be stored. The special points to mention are: that read and write operations can be automatically created for any attributes; and that the underlying structure of a class is not restricted to being a tuple, as in this case. The **structure** clause can be replaced by:

```
structure integer
```

or

```
structure set of VEHICLE
```

That is, a class can be a set of base type values or a set of sets.

The persistence of objects in O_2 is entirely determined by the class in which they are created. Thus, if a class does not have a **persist as** clause, its objects will not persist. If it does have a **persist as** clause, they will persist. This seems to be confusing two orthogonal issues - the type of an object and its persistence. There is no way to specify a type, some of whose objects persist, while some don't.

The O_2 programming language [Lécluse and Richard, 1989] is multi-language in two senses. In the positive sense, the O_2 environment may be programmed in more than one co-operating language, such as extensions to C or BASIC, (with a common data definition language which provides a common reference definition for these data manipulation languages) and in the negative sense, these extension languages are essentially two languages glued together. To take the positive point first, it is intended that programs can be written in any of a set of languages (CO_2, BO_2, etc) so that a programmer will have access to a favoured style of programming. Modules written in different languages will be mutually accessible. The negative point is that the syntactical device for extending languages to fit O_2 is to provide an O_2 language and then to push bits of this language into modules otherwise written in the host language. These two languages are then separated by an escape character ("$") in a very ugly way. This problem is not just syntactic for there is also a semantic dissonance and consequently a very low-level interface.

These systems are a subset of those that have been described recently, such as (IRIS [Fishman *et al.*, 1987], ENCORE [Hornick and Zdonik, 1987], ORION [Bannerjee *et al.*, 1987], etc.) and they all suffer from a number of problems. Firstly, it has not yet been possible to implement any of these efficiently. O-O database technology is roughly in the same position as relational technology was in the mid-70's. Making these systems run fast is clearly a more difficult problem, but, as part of the goal is to achieve systems which make use of enhanced technology to throw more work onto the computer, the problem may be expected to be solved.

A more serious limitation seems to lie in the ways those systems combine programming language and database ideas. In particular, there is a problem in requiring a programmer to manage two languages. This is not to criticise those systems, like O_2, which seek to provide a number of parallel languages for users, within each of which a whole application can be specified. This seems a highly laudable architectural decision. What seem unacceptable are systems like V-base with TDL and COP, which require an application programmer to know two languages, one for specification and one for implementation. Conversely, the way in which O_2 glues together two programming paradigms seems inferior to the provision of a seamless language as provided by Trellis-Owl. OODB systems are facing the problem of providing a programming language interface to programmers who wish to retain a particular favourite paradigm which may be inappropriate for the kinds of application they are building. There is also a fundamental choice to be made between seamlessness and support for interworking of systems built using different programming technologies.

2.2.2.6 Summary.

The O-O approach has been described and is seen to include the following notions:

objects are single, identifiable entities;

all objects belong to classes;

classes are organised into an inheritance hierarchy;

there can sometimes be a kind of polymorphism in that an object can be treated as being in any class which is above its actual class in the hierarchy;

there may be a separation of class specification and implementation;

the attributes and operations applicable to a class are encapsulated into a single description and frequently all access to objects is restricted to calling the operations;

no code can exist anywhere except in the operations of classes.

There is no problem with any of those statements except the final two. The other statements seem to imply a systematic and clear description of the passive nature of the application world and a clear structure within which to place a great deal of its active component as well. Strict encapsulation (access restricted to operations) leads to verbose code, but the mechanisms of Eiffel and Trellis/Owl resolve this.

The O-O architecture is most deficient on the final point. There are at least three kinds of active feature that are not well represented in this architecture:

Active components: Consider a light button object. This will have components such as its icon and where it is on the screen. It also has a component which is the operation which will be called when the button is pressed. This is not the same as an operation associated with a class. A component will have a different value for every light button - a different operation that will be carried out when each button is pressed. A class operation will be the same code for every button, although it will be bound to different objects.

Dyadic operations: This has already been mentioned but to reiterate, dyadic operations seem most naturally viewed as operations which take two objects of the same type, not as operations over the class which accept another member as a parameter.

Processes: These are stand-alone active objects, not tied to any specific class, which the programmer may wish to manipulate and reason about as though they were objects in their own right. One example of this is getting programs started. Section 5.6 of [Meyer, 1988] identifies this as a problem some people have with O-O systems and then proceeds to show how an Eiffel program is started, which only seems to underline the contorted thinking that is required.

It is hard to see how to model any of the above naturally within the O-O world of Abstract Data Types. Therefore a paradigm which has been introduced principally to provide natural and intuitive programming constructs has failed to do so for some critical parts of the programming problem.

2.2.3 Database Programming Languages.

The next area of the survey concerns attempts to marry together database technology with programming languages, by extending existing programming languages or designing new ones which bring database functionality and programming language expressiveness into one facility. The area is fully surveyed in [Atkinson and Buneman, 1987] and this section will concentrate on a few major examples.

The integration of database systems and programming languages has traditionally either been achieved by providing an interface between the two in the form of a set of low-level subroutine calls or by embedding one language within another. What DBPL's strive to provide is a single language within which to express computation and data manipulation and storage. This raises a number of issues, some of which are already familiar.

What kind of type system should be provided? A strongly typed language will provide early detection of type errors. There should also be a degree of polymorphism in order to provide general purpose procedures. What is the relationship between the type system of a programming language and the notion of a class in a OODB system? Clearly they both describe the properties of a set of objects, but their use from then on is somewhat different. The type system of a language is there to provide support in program compilation. The class of a database provides not only a description but a set of values. Should the type description be matched with a specific extent or not?

Does the language provide persistence, in the sense that any object can exist for as long as required without special handling? Many languages only permit certain types of value to persist. Others use special mechanisms, such as file systems, to make objects persist.

Atkinson and Buneman provide a list of desiderata for DBPL's including: most of the common programming constructs; strong type checking; as much static type checking as possible; data type completeness; a consistent naming system for all objects; a bulk type; inheritance; polymorphism; and orthogonal persistence. They also recommend that the programmer should be free from all concerns about the placement and movement of data.

In this short survey, some of these issues will be considered with reference to a few languages.

2.2.3.1 Relational Programming Languages.

There are several languages which integrate the Relational Model with a programming language. Pascal/R [Schmidt, 1977] will be discussed, but the successor languages Modula/R [Koch *et al.*, 1983] and DBPL [Schmidt and Mall, 1983] as well as Plain [Wasserman et al, 1981] and Rigel [Rowe and Shoens, 1979] should also be mentioned. Pascal/R uses the record type as a platform on which to build types for relations and databases.

The definition of a database begins with the definition of a record type for the tuple of a relation as follows:

```
type bookTuple =   record
                       catalogueNumber: 0..9999;
                       author:          packed array [1..30] of char;
                       title:           packed array [1..100] of char
                   end;
```

and then the declaration of a relation to hold books, with *catalogueNumber* as a primary key:

```
BookRel = relation catalogueNumber  of bookTuple;
```

and finally, a library database is declared, as in:

```
LibraryDB =        database
                       Book: BookRel;
                       Borrower: BorrowerRel;
                       Loan: LoanRel
                   end;
```

All of these are types in the program and these can then be used as follows:

```
var library: LibraryDB;
begin
      with LibraryDB do
            for each B in Book: B.author = "Elizabeth Taylor" do
                  writeln( B.catalogueNumber, B.title )
end.
```

These features, together with operations which are equivalent to the relational calculus, show that it is possible to embed mechanisms for handling relations within a programming language. Two different kinds of problem can be found in Pascal/R. Some of the inadequacies of the language are due to its innovative nature. The language extensions make Pascal/R even less data type complete than Pascal. For instance, **for each** is provided over relations, but not files or arrays, while not every type is allowed to be the field of a record or tuple and, more seriously, of a database. Such problems can potentially be fixed using the same approach, as shown in successor languages, such as DBPL.

A more serious problem concerns the extensibility of an application program written in such a language. In essence, the schema must be verified and fully type checked at compilation time. Thus in subsequent runs, the schema cannot simply be extended without editing and recompiling the software and furnishing translation mechanisms from the old schema to the new one. The eager static type checking of such languages creates a barrier to schema evolution. Another serious problem lies in basing the language on the relational model. As described in section 2.1.2, this raises severe difficulties in capturing the meaning of the application in the database description.

2.2.3.2 Galileo.

Galileo [Albano *et al.*, 1985] is a database programming language which tries to avoid this last problem by centring the language design around Semantic Data

Modelling concepts. It is a strongly, statically typed language with constructs for persistence, modularisation, higher-order functions and data modelling. Providing a persistent library database in Galileo looks like:

```
use LibraryDB :=
        ( type address = ( House: integer
                           and Street: string
                           and PostCode: string )
        and book class
                book <->
                        ( CatalogueNumber: integer
                           and author: string
                           and title: string )
        and borrower class
                .....
        )
```

Here *LibraryDB* is an environment within which a set of type and class objects is maintained, which has been made persistent by the **use** keyword. Environments are thus used as the unit of modularisation of an application. There are a great many type constructors for both concrete and abstract types and sub-typing can be either explicitly stated or, for concrete types, inferred. There is also a notion of subclassing in that the definition:

fiction **isa** *book*

has the expected inheritance of the fields from *book*. Galileo adds to this programming constructs, in a functional style, to make the language sufficient for the expression of the computational aspects of the database.

Whereas Galileo has provided a significant step forward in combining data modelling ideas, persistence, and functional description, to permit a complete database description to be made, it seems to have two main defects. As with Pascal/R, the static type checking gets in the way of database extension, while its data modelling constructs are extremely confused, particularly as regards the concepts of class and type. Two mechanisms for introducing the description of a type are supplied. One is used for those types which have an extent maintained and the other is used for those types whose extent is not maintained. It would be preferable to introduce one construct for describing all types and another for describing their extents. As things stand, in the above example, there is no possibility of creating an extent over addresses at a later point, nor of creating books not held in a class. This point will be discussed further in the conclusions to this chapter.

There is also a dizzying set of syntax for introducing abstract types, which are distinguished from concrete types for reasons which are far from clear. Finally **type inferencing** is introduced into the system. The problems with this are discussed in the next section.

2.2.3.3 Polymorphic Database Programming Languages.

The problem with most statically typed languages, illustrated clearly in the language Pascal/R, is their inability to express code which uses data types drawn from a

set which has yet to be defined. The next group of languages has been designed with this in mind. Although not explicitly designed for use with databases, ML [Milner, 1984] and Poly [Matthews, 1985] provide mechanisms for writing polymorphic code and thus address this problem. Poly which is derived from Russel [Demers and Donahue, 1979] is a language which provides parametric polymorphism - i.e. the ability to describe types which are parameterised by other types. Thus a type for a stack of any similarly typed objects can be defined with the types of the elements being a parameter. This stack will then be available to any subsequently defined types.

ML on the other hand provides polymorphism through type inferencing (touched on in the previous section). This idea is extended in the language Machiavelli [Ohori *et al*, 1989], which has been designed within the framework of database programming, so the discussion on polymorphic languages will centre on Machiavelli.

Machiavelli's design starts from the desire to provide the following kind of polymorphic code: given any relation which has an age and a salary field, produce the salaries of persons aged 28 (or tuples for which the age field = 28). This query must be expressible in a statically typed environment, with no run-time type checking. It would be expressed:

fun *salary28*(*X*) = **select** *x.salary* **where** *x* <- *X* **with** *x.age* = 28;

This is type checked to the type:

{ [("∂) *salary*: "ß;*age*: **int**]} -> { "ß }

which roughly means that the function will take a set of values ("{}") which are records ("[]") which have **at least** a *salary* field of indeterminate type and an integer *age* field and return a set of objects which are the same type as the *salary* field. The key part of the preceding sentence was the "at least" - any other fields may appear and are irrelevant to the type checking.

Using this mechanism, not only can Machiavelli represent relational operations, but also Object-Oriented or Semantic Data Modelling ideas. The notion of class *X* **isa** *Y* is represented by listing the fields of class *Y* in its definition and then listing all the fields of *Y* and the additional fields of *X* in *X*'s definition. The sub-typing is then inferred. Using the type inferencing mechanism in this free and orthogonal style certainly provides a great deal of power, but seems to run into problems in a multi-language environment. It seems likely that one user might introduce type *slimmer*, say, which has merely *name* and *weight* fields, and another introduce a type *ship*, which also has those fields and maybe others, and a completely erroneous inference might therefore be made. This might not harm the computation but if it becomes apparent the programmer could be confused. Therefore, it may be preferable to provide some syntax which limits type inference.

2.2.3.4 Persistent Programming Languages.

Amber [Cardelli, 1984] takes a similar course to provide polymorphism with a universal union type, **dynamic**, out of which types may be projected or coerced. Rather confusingly, it ties this mechanism to persistence since only objects of type **dynamic** may persist since this ensured that the actual type information was stored

with persistent objects. The language is rich in type constructors, having tuple, record, variant, array, function and channel (for concurrency communication) constructors.

The coercion strategy, which is essentially the same as used by PS-algol, permits objects of as yet unknown type to be assigned the temporary type **dynamic**. Objects of this type may be passed around, but in order to use them, they must be forced into a concrete type, at which point the operations for that type become available. These operations can be statically type checked because the coercion operation must appear in the code before the operations and so the concrete type of the operand is known. Objects are made persistent by converting them from concrete types to type dynamic, at which point the concrete type is stored alongside the object's value, and then **exporting** them from the current **module**. The importation and exportation of objects between modules makes Amber a language which begins to support the basic mechanisms required for large software development (see section 2.2.4).

The types are arranged into an inheritance hierarchy by an automatic **type inclusion** algorithm which asserts, for instance, that a record with a set of fields is included in a type which includes only a sub-set of these fields. Similar inclusion rules are developed for variant, array, channel, tuple and functions (the contravariance rule). This is similar in every respect to the type inference of Machiavelli (which was developed from it) and means that Amber, too, can supply the basic functionality of Object-Oriented systems in a properly typed environment.

The ideas in Amber are developed further in [Cardelli and Wegner, 1985] and [Cardelli, 1988]. The former paper puts the type systems of such languages onto a firm basis within a single type calculus, with a basic set of types, some type constructors and universal and existential quantification. The type system of any of the languages referred to here should be describable in terms of this calculus. This work may supply the kind of basic theory to type systems of database programming languages which Codd provided for relational systems.

[Cardelli, 1988] adds a new concept for polymorphism, the **kind**. A kind is a set of types in much the same way as a meta-class is a set of classes in Taxis. There will in general be a kind called Type, for instance, which is the set of all types, and there could be a kind, which is the set of all tuples having an integer field called *age*. Thus, the parameters of functions can have either their type or the kind of their type specified. Producing such a framework for describing types should produce programming languages (like Quest [Cardelli, 1988b]) within which it is convenient to describe everything known about the types in the database. A caveat, however, is that a three level world of values, types and kinds, still feels restrictive. It may be that one will eventually want to talk about a set of kinds. However, the present system is probably complicated enough to describe any conceivable application, and any increase in complexity will surely produce an untenable cognitive load. Also, it is not clear whether or not languages of this kind can be implemented or type-checked.

FAD [Bancilhon *et al.*, 1987] is a language which divides the object space of a program into a transient and a persistent section. Within this division may be specified objects which are defined as tuples, sets or variants or belong to some abstract type. The main construct added by FAD is that of actions. These are pieces of program which may be manipulated like other objects. The language provides inheritance mechanisms, but seems to concentrate on the fine details of language design, without developing a clear underlying semantics. In many respects, it resembles PS-algol.

2.2.3.5 Conclusions.

This section has briefly introduced some of the difficulties in producing a programming language which combines computational completeness, database operations and sufficient simplicity to render the language usable. There follows a survey of some of the problems tackled in these languages:

Persistence. This is usually provided as a bulk object type, all of whose members and components will persist between runs. Thus Pascal/R has the relation, Galileo the persistent environment and Amber the dynamic type. The introduction of such types minimally disturbs the syntax of the language, which is one of the principal intentions of orthogonal persistence. However, in some cases, some of the other types could not be put into these bulk types and so could not be made persistent.

Naming systems. The ability to name persistent and transient objects in a consistent way has proved problematic to language designers, when faced with the problem of extensibility. A complex type system may overcome the problem for types, but the names of objects are either lost when an object is stored or provided as a string component of the object.

Extensibility. Allowing the names of types and their components to persist is usually easily achieved, but in a statically typed system this results in types which are cast in stone and thus not extensible. None of the systems really surmount this problem, but some (e.g. Amber) manage to finesse it by providing a type (dynamic) which defers type checking until run-time by performing type coercion then.

Set Operations. All of the languages, except ML and Poly (which can be extended by the user), include sufficient set constructs to perform the usual database operations (iteration, selection, projection, etc.). Therefore, in providing database operations on statically determined data sets, all prove sufficient.

Polymorphism. There are a number of approaches which allow a single code body to be bound to data of differing types. The parametric polymorphism of Poly allows types to be defined which are generic versions of a set of concrete types. These parametric types can then be used for the parameters of any operation. The Object-oriented systems described in 2.2.2 permit the explicit specification of type hierarchies, so that an operation specified over one type is applicable to all of the types below it. ML and its extensions (Galileo, Machiavelli, FAD and Amber) provide the automatic inference of sub-type relationships. The types required by an operation are inferred from the objects it uses. Any object supplied to the operation will be checked to be a subtype of the inferred type for this parameter. This mechanism may be sufficient for most purposes, but may be overinclusive as discussed above.

Data Type Completeness. The single most simplifying language design goal is the removal of arbitrary distinctions between the way objects of different types can be manipulated. Not all of the languages provide this, but any failings in this regard could be viewed as design choices to get prototype systems running. Any ultimate DBPL will surely be data type complete.

As a final point, the history of the development of DBPL's is quite short (1978 for Pascal/R). The work has proceeded by attempting to build languages in order to reveal the significant problems. These are now sufficiently understood (e.g. [Atkinson and Buneman, 1987]), that rapid progress towards better languages can be expected.

2.2.4 Software Development Systems.

A completely different approach to the provision of improved languages or data modelling tools is to supply improved programming environments. This section will describe a few of these systems and outline a few desiderata.

2.2.4.1 SCCS.

The Source Code Control System [Rochkind 1975] is a fairly primitive tool designed for use within the UNIX system, to aid programming projects to control changes to the source code. A new software module is supplied to the system and its source code is stored. Subsequent changes to this code are then stored in the form of deltas. Deltas are created by editing the last version and supplying the edited form - the delta being automatically derived by comparing the two. Using the original code and the deltas, any version of the code can be recaptured. All versions of a module form a purely linear structure, although the naming system has a 2 dimensional feel to it. The first version is Release 1, Level 1 or version 1.1. Subsequent versions are called 1.2, 1.3, etc until a stable form emerges for a new release, at which point the numbering system continues with 2.1, 2.2, etc.

Within this framework, very little control over the software development process is possible. There is no provision for the splitting of versions; nor is there any way that you can successfully go back to an old version and insert a new version between two old ones. Therefore there is no way of testing two lines of development simultaneously and merging the results.

2.2.4.2 RCS.

The Revision Control System [Tichy 1985] is a development of SCCS which handles the organisation of versions and configurations. It uses the same concepts of versions and deltas of source files as SCCS and the same method of creation and submission of new versions. However, it also includes concurrency control in the form of a check-out/ check-in mechanism. Someone wishing to edit a module checks out and locks that module until finished with it, whereupon he checks it in again and clears the lock. Thus two people will not be trying to change the module in different ways at the same time.

However, RCS does permit multiple versions of a module to be created at any point of the version history, although there is still no automatic merging of these versions provided. The problem of automating this process is extremely complex, but is common to any design process of complex objects. A sophisticated check-in/ check-out mechanism with a rich taxonomy of locks, including notification locks, etc. [Fernandez and Zdonik, 1989] may eventually emerge.

2.2.4.3 UNIBASE/DAMOKLES.

The UNIBASE project [Dittrich *et al.*, 1986] is attempting to provide an open, integrated environment for UNIX called DAMOKLES. The authors note that file based systems such as the above fail to provide the database facilities required, such as data integration, security, access control, etc. They also note that traditional database systems are also insufficient in that they their modelling constructs are too simple, do

not integrate consistency constraints automatically and give poor support to long transactions. They therefore propose a system which is Object-Oriented and provides traditional database facilities such as transactions, concurrency control, security mechanisms, etc. It is intended to manage the whole of the software, include source, object code and documentation.

DAMOKLES is built on a data model called DODM (Design Object Data Model). The description of complex object types in DODM contains a number of simple (i.e. integer, string, etc.) attributes and a number of sub-components with other complex object types. Object types may then be related by relationships which link any number of types and on which cardinality constraints may be imposed. A variety of navigational aids are provided to retrieve information from this structure. The model is used to model software development by incorporating the following notion of versions.

Every object has a **generic** form. All versions of an object will have the same structure, although the values of attributes and actual sub-components may vary. Versions are numbered and ordered either in a list, in a tree or in an acyclic graph. Operations are provided to locate particular versions, to insert versions anywhere in the graph, to locate the generic object and to remove versions.

Further features of the system include the ability to handle multiple databases - an object resides in one database but may be copied to others. Relationships may hold between databases, thus permitting shared software libraries, for instance. A check-out/check-in transaction paradigm is provided to facilitate the long transactions typical of software development. This allows a developer to check out one or more program objects for modification and later to check the modified version back in. This permits other users to access (but not to modify) the objects checked out.

All of this, when applied to a database containing programs, sub-programs, libraries, source code, object code, diagrams of various kinds, etc., seems to promise a very rich environment within which to develop software.

2.2.4.4 Gandalf.

Gandalf [Habermann and Notkin 1986] is a project to develop the ability to generate software development environments quickly and cost effectively. In a Gandalf environment, the centrepiece is a syntax-directed editor for the programming language, called an ALOE editor. This is produced by supplying an abstract syntax (e.g. "WHILE = %bool-exp %assign"); a concrete syntax (e.g. "WHILE = @1 do @2" meaning that two parameters, whose types are described in the abstract syntax, must be described); and a set of action routines, which are called as a given syntax tree is built. These action routines may perform a number of functions, including semantic checks and window and memory management.

Programs are developed using one of the ALOE editors. This permits the user to select program constructs by giving simple commands. Thus typing in "WH" specifies that the next piece of the program is a while statement. Placeholders are supplied for the boolean expression and the assignment command and these are next filled in by clicking over the placeholder icon and (again using structured commands) typing in the code.

One example of an environment developed using this technology is a prototype system (GP) for software development, including version control, incremental compilation and project management. GP was developed by producing an abstract syntax for software development structures (modules, versions, access control lists and documentation). From this an ALOE editor is available for a meta-language, which includes the ability to refer to different versions of modules, to specify modules as exporting various facilities and to compose modules into complete systems. Thus, an impressive feature of Gandalf is its ability to use the same mechanism to specify software and the software control system.

2.2.4.5 Eclipse.

Eclipse [Bott 1989] is an Integrated Project Support Environment developed in an Alvey project, together with tools for software development (rather than project management). Eclipse is built round a kernel, which consists of the Public Common Tool Environment [PCTE, 1986], augmented by another component called the Public Tools Interface, which contains the following:

a **two-tier database** in which the higher level is a PCTE object description, while the lower level is a description of a component object;

an underlying **data model** and **interface description language (IDL)**, which is specific for the support of objects typical of the production of software;

configuration control mechanisms for glueing together particular versions of objects;

a high quality **user interface**; and

a table driven **design editor** for manipulating diagrams.

On top of this kernel has been built a number of tool sets:

MASCOT 3 - a tool set which permits program designs to be captured, manipulated and checked graphically;

LSDM - this supports requirements analysis and system design by providing graphical editors for dataflow diagrams, logical data structures and entity life histories; and

Integrated Ada Development System (IADS) - this supports the whole life cycle of Ada software, including source creation and editing, program libraries, compilation, linking, execution, symbolic debugging and version management.

Hood [Welland, 1989] is a Hierarchical Object-Oriented Design tool generated for the European Space Agency.

Eclipse tackles a great many of the issues of software creation, from the relatively low-level notions of a single-language software development system (IADS) to higher-level project support methodologies. It does so, in an environment with a

common kernel into which a number of complementary modules can be plugged. Presumably, it would not be difficult to produce tool sets to support any software design methodology of whatever level in Eclipse.

2.2.4.6 Conclusions.

The systems described here all attempt in one way or another to supply a structure within which to manage efficiently the development of software. They vary greatly in their sophistication, but the trend is towards the integration of database components to manage the software objects. In order for the database system to be sufficiently powerful, it must have a way of storing code. Although relational systems have been used for this purpose, the contorted nature of using relational schemas to do this is not encouraging. In order to manage this problem, more sophisticated data models must be used. Thus DAMOKLES appeals to a general purpose Object-Oriented paradigm, while Eclipse has a data model specific to project support. The provision of a sophisticated database system with a language which can manipulate program objects would thus be an enabling technology for this work. In particular, it would mean that the application development environment could be viewed as just another application, which could be produced in the same way as any other. Chapter 8 describes the contribution PS-algol has to make in this area.

2.2.5 Specification and Rapid Prototyping Systems.

Attention is now switched to earlier stages of the software design process - the requirements and implementation specification stages. Firstly, a requirements modelling language, developed as part of the Taxis project, is described. The aim of this work is to permit the description of the requirements of an application in such a way that it can be checked for internal consistency. Then SAGA is examined. This includes facilities for writing executable specifications. Next, the major formal specification systems are briefly touched on and ADABTPL is considered as an example of integrating formal specification and database techniques. This is followed by a brief survey of rapid prototyping systems. From all of these it is noted that several different activities are being discussed in very much the same terms. It is possible to conceive of systems in which all stages of the software creation process can be specified in an executable and checkable form. All stages would linked together by a common syntax and a common model of software production, but would allow different levels of detail to appear.

2.2.5.1 RML.

Requirements Modeling Language (RML) [Greenspan, 1984] is a spin-off from the Taxis project. It permits the specification of the entities, activities and assertions that may be defined for an application in a syntax that is essentially the same as that of Taxis. It will be described more fully in section 7.2, but this section touches on a few of the design decisions.

RML is designed to permit the description of the "problem situation" rather than on the "solution system". Definitions in RML are designed to be statements of what must be included in any program modelling the application domain. Such statements are:

PATIENT **in** *PATIENT_CLASS* **with**
association *ward*: *HOSPITAL_WARD*
doctor: DOCTOR
producer *register*: *ADMIT_PATIENT*(*pat:* **self**)
.....

or *ADMIT_PATIENT* **in** *ACTIVITY_CLASS* **with**
input *p*: *PERSON*
precondition *inyet?*: **not** *IN*(*p*, *PATIENT*)
part *check_ID*: *CHECK_ID*(*p*)
Put: *CHOOSE_WARD*(*w*)

or *IN_HOSPITAL* **in** *ASSERTION_CLASS* **with**
argument *p*: *PERSON*
part *patient?*:*IN*(*p*, *PATIENT*)
present: *PHYSICALLY_PRESENT*(*who*: *p*)

All of these statements describe groups of objects as types by supplying everything that is known about such objects at a highly abstract level, within a syntax in which every piece of knowledge is labelled. The first example describes the class of all patients and mentions two other object types with which they may be associated, a ward and a doctor, and an activity which creates such an object, *ADMIT_PATIENT*. All of these are labelled, so that the producer is called *register* and may be referred to as such. Similarly, an activity has associated objects and assertions and then consists of an unordered set of sub-activities. An assertion can also refer to objects and be composed of sub-assertions.

A specification in this system can be checked only insofar as if *A* refers to *B*, then *B* must have the inverse relation to *A*. For instance, a *PATIENT* is created by *ADMIT_PATIENT* and *ADMIT_PATIENT* operates on *PATIENTs*. The references in the description can, though, be left dangling - for instance *CHECK_ID* need not be specified if it is small scale enough to be understood. Given a requirements specification in such a language, it would be a relatively small step to provide an executable version. Section 7.2 shows how to do this by supplying some basic objects and an object creation environment.

2.2.5.2 Formal Specification Systems.

The ability to express with mathematical precision the desired behaviour of an application under construction leads to automatic techniques of system verification. The production of complex software is such an error-prone task that anything which increases confidence that a given program does what it is supposed to do is clearly of great value.

A number of languages and methodologies have been developed to fulfill this purpose. VDM [Bjørner and Jones, 1982], Z [Hayes, 1987] and OBJ3 [Goguen and Winkler, 1988] are three such methods in common use. All permit the formal description of functional (rather than performance) specifications, and these fall into a number of categories [Liskov and Berzins, 1979]. For procedures, there may be input/output specifications (which essentially describe the conditions which are to hold at the start and finish of a procedure) and operational specifications, which give a

highly abstract description of the computation involved in the procedure. For data types, there are again two approaches: an axiomatic approach, in which all the base statements known to be true for that type are recorded; and abstract data models, which correspond to data models described above, except that the data abstractions (tuple, set, etc.) are formally defined.

The SAGA project has designed a software development system, called ENCOMPASS, which attempts to provide an automatic framework for each stage of the software development life cycle [Campbell and Terwilliger, 1986]. It starts from a formal specification of the problem domain written in a language called PLEASE. This specification, which consists of a series of pre- and post-conditions for operations, is executable and so the specification can be tested. Transformation and refinement rules can then be specified which take this specification and transform it, mostly automatically, into program modules. These modules are developed using version control techniques and then combined into working programs by configuration techniques similar to those described in section 2.2.4.

The value of a system which introduces the notion of rigorous specification with automatic aids to produce implementation needs no underlining. Provided the specification can make powerful enough statements, this would seem to be one of the ways in which software development must proceed. However, the formal methodologies rely on separating out the various aspects of software development into discrete stages. The lack of realism in this approach will be further discussed in the conclusions to this chapter.

2.2.5.3 ADABTPL.

ADABTPL [Stemple, 1989] is a system which attempts to combine data modelling and formal specification approaches. It is essentially a three stage process. Initially, using a semantic data model, the user specifies the database schema informally. This is transformed into a formal specification in the ADABTPL language, the theory of which is a version of Boyer-Moore computational logic. This specification can then be augmented by the programmer before being automatically transformed into an implementation in a lower-level language with a persistent object server.

This combined system provides an excellent example of the convergence of the various approaches. A quick intuitive description of the data model is generated in the first phase. This can then be extended in the second within a formal framework, which means that mechanical techniques can be brought to bear to verify constraints on transactions. This kind of convergence will become more common as application implementation systems develop.

2.2.5.4 Some RPT Systems.

Rapid Prototyping is another area of growing utility [Hartson and Smith, 1987]. The ability to lash up a version of a system is an alternative method of verification of requirements to the formal specification techniques. The essence of the technique is embodied in the name. A rough prototype is quickly assembled and demonstrated to customers. Performance criteria aside, the prototype should strongly resemble the

finished product and so demonstrate the effectiveness of the design. One sample language is described to give the flavour of rapid prototyping .

Peridot [Myers and Buxton, 1986] is a program which permits the construction of user interfaces by example. The user draws the screen layouts required for the interface and demonstrates the user's interaction with it by clicking the mouse or typing on the keyboard. The behaviour of the demonstrator is stored as simple condition-action rules and turned into code in the form of procedures, which are combined to create a complete user-interface for the application. In this way, the user interface can be experimented with quickly in a more realistic fashion than by using pencil and paper designs.

2.2.5.5 Conclusions.

This section has described some higher level software production techniques, which can be grouped together since they all deal with "pre-implementation" phases of software development. They all also share the notion that even if they permit a program to be written, it will not be the final product, but a specification or a prototype. Moreover, the final product cannot automatically be derived from the specification. However, it seems that in the long term it is conceivable that all of these high-level tasks will be accomplished in a single language or matched set of languages. This will produce not just executable versions for testing, but finished products which will be reliable and provable according to specification.

This is all the more important since these separate systems depend on a notion of separation of programming into discrete stages which is clearly at odds with the real-world development of computer system. Requirements are usually poorly specified and the customer only really knows what is required when products which are unsuitable are delivered. Each of the phases of modelling, specification, prototyping and implementation are generally intertwined [Swartout and Balzer, 1982] in a way which belies the separation of these activities. Integrating them in a single system seems the only sensible solution.

2.3 The Persistent Programming Research Group.

2.3.1 The Concept of Persistence and the Birth of the PPRG.

The **persistence** of an object is the length of time for which it is accessible by a program. This can range from variables which exist only during the block in which they are declared, to payroll data which will outlast the program run that creates them and probably even the computer system on which they are created. Typically, programming languages manage short-term or transient data, and database systems or file managers handle long-term or persistent data. What is required is for these to be unified into a simple system that manages all kinds of data.

This involves two distinct ideas. Firstly, any type of data should have the right to any degree of persistence. Secondly, the way in which data are referenced should reflect their persistence as little as possible. Taken together, these concepts embody the

Principle of Orthogonal Persistence. Any language conforming to this principle is a persistent programming language.

The need for orthogonal persistence in systems was first expressed in [Atkinson 1978]. Atkinson noted that engineers and scientists were slow to make use of database systems because the data models underlying databases matched poorly with those underlying the programming languages which they were dependent upon in order to express their complex computations. The database systems provided the power to store data on a long term basis in a systematic way. The programming languages provided the power to ensure that the data were in a form which matched the intended algorithm (via their type systems). When the data from a program were stored in a database, however, this typing information was lost. The recommendation was to extend programming languages to allow them to store any long-term data in a simple way.

The first attempts to implement the idea, by Malcolm Atkinson, Paul Cockshott and Ken Chisholm, were proposed extensions to Pascal and Algol 68 at the University of Edinburgh [Atkinson *et al.*, 1981]. This led to the proposal of NEPAL as a brand new persistent language [Atkinson *et al.*, 1982]. At the same time, work at the University of St. Andrews by David Turner and Ron Morrison developed elegant simple forms of algol called algol-s and S-algol [Cole and Morrison, 1982]. S-algol proved to be a suitable language with which to experiment with persistence and so the two groups joined together to create PS-algol [Atkinson *et al.*, 1983a, 1983b, 1983c and 1983d], with the Edinburgh group moving to the University of Glasgow in the process.

From 1983, the two groups began collaborating with ICL Ltd. and from 1985 to 1988, the three groups combined together to form the PISA project, funded under the Alvey initiative. During this period, the bulk of the work reported in this Thesis was carried out, as part of an evaluation of the language. Simultaneously, a successor language, called Napier, was designed and has now been implemented. This language will be briefly discussed in the conclusions to this Thesis.

2.3.2 S-algol.

S-algol [Morrison 1982, Cole and Morrison, 1982] was developed at the University of St. Andrews. Chapter 15 of [Cole and Morrison, 1982] describes the design philosophy behind the language. This was to take the facilities provided by algol-like languages (block-structure, parameterised procedures and static type checking) and to add to these some principles derived from work by Strachey and Landin in the 1960's [Strachey, 1967, Landin, 1966], which should produce simpler languages. These were:

The Principle of Correspondence. This states that the ways in which named objects are introduced should be the same everywhere. In particular, variable declaration and parameter declaration should correspond. One example of a violation of this principle is that types in Pascal may be declared but not passed as parameters.

The Principle of Abstraction. This states that for any semantically meaningful syntactic category of a language, there will be a facility to provide an abstraction over it. One example of this is the provision of functions as an abstraction of expressions.

The Principle of Data-Type Completeness. Mentioned in section 1.2, this means that every data type should have the same rules for manipulation, with no exceptions. Pascal provides a number of examples of violations of this principle - for instance, only some types of object can be the elements of sets.

Using these principles, S-algol was designed as a language with a relatively simple type system, consisting of five scalar types (integer, real, boolean, string and file), the ability to construct freely vectors of any type, and a facility to create types which are the named cross-products of other types (**structures**). At any point of the program, new objects could be declared to be of any of these types. Furthermore, objects of any type could be declared to be constants at their point of creation. Procedures were introduced which could take any number of parameters and return at most one result. Parameter passing is by call-by-value. There are a small number of control structures in the language.

The fine detail of these facilities will not be discussed here as they have been inherited by PS-algol and will be discussed in Chapter 3. Adherence to the principles stated above resulted in a language of great simplicity and elegance. Many of the irritating arbitrary distinctions between the ways different types of objects are manipulated in other algols have been removed. Indeed the University of St. Andrews have found it an ideal vehicle for introducing students to programming for a number of years, because this simplicity has permitted the instructor to concentrate on the critical notions involved and not to be distracted by having to explain the irrelevant exceptions found in other languages.

2.3.3 PS-algol.

The development of PS-algol came about as a result of the Data Curator group at the University of Edinburgh combining with the S-algol group at the University of St. Andrews in order to turn S-algol into a persistent language. Given the power of S-algol, this proved a relatively straightforward task - at least in the domain of syntax extension. The basic syntax was not changed significantly - the added functionality was achieved by functionally extending the language. System procedures were added which provided orthogonal persistence.

This extension relied on a consequence of the data type completeness of S-algol - that the scope and extent of objects need not be the same. If an object was declared in the closure of a procedure or placed in a structure, then it would persist by virtue of being an intrinsic part of the procedure or complex object, even if it went out of scope. The only step that remained to be taken was to establish objects which outlasted the run of a program and to provide some mechanism for attaching other objects to them.

These basic long-lived objects were called, perhaps unfortunately, "databases" and a format for these was created called a "table". The "language extension" required was the provision of the system functions to create new databases and open old ones, to insert and look up objects in tables and to commit any changes to any opened databases. Persistence was then available since any kind of data can be stored in a structure and any structure can be linked into a database. The simple storage of any kind of data embodied the Principle of Orthogonal Persistence and so PS-algol qualified as the first persistent language. This change in semantics required new

implementation strategies [Cockshott, 1983, Brown and Cockshott, 1985, Dearle, 1988, Brown, 1989].

2.3.4 The Place of this Work.

The work reported in this Thesis was part of the effort to investigate the use of PS-algol. It was funded under the PISA project, which was part of the Alvey initiative, as an ICL University Fellowship. The work attempts to be a thorough examination of the language and its ability to provide implementations of data intensive applications, data modelling tools and system building tools, as well as providing the beginnings of a programming methodology for the language.

2.4 Summary of Work Surveyed.

This has been an extensive examination of the various techniques and programming systems proposed for facilitating the task of software production. From this, an attempt will be made to draw a number of conclusions.

Firstly, the various techniques for data modelling, specification, prototyping, software management and programming need to be combined into a coherent and integrated system. It is no surprise to discover that the same techniques of data and procedural abstraction crop up over and over again. These should be brought together into a software development environment which includes at least the following: a data model; a database for software; a database programming language; and user interface tools.

The data model is required for producing schemas of all the applications which are to be produced. Any of the more sophisticated data models may be used, enhanced with a construct for processes or activities as first-class objects of the model.

The software database schema should be written using this data model and include notions of modules, versions, configurations, documentation and specifications.

The database programming language should be designed to have a type system which reflects the underlying data model rather than one which conflicts with it. It should be as simple as possible, using design principles such as data type completeness to control complexity. In so far as possible, the computational model should also be simple, but permit a variety of styles of programming. Moreover, a matched language (perhaps a subset) should be supplied for writing specifications and/or prototypes.

The user interface should be specified in the same data model and written in the same language as the rest of the application.

In short, there is an increasing need to bring the research work from a number of areas together to facilitate software development. Here, a claim will be made that PS-algol represents a first step towards bringing these worlds together.

Chapter 3. The Persistent Programming Language, PS-algol.

The language PS-algol was introduced by the Persistent Programming Research Group (PPRG) as the first in a series of persistent programming languages. The ideas of the PPRG have been described in Chapter 2. These ideas have resulted in languages which are simple and coherent and derive their power from the extension of facilities to remove arbitrary restrictions. With these features, a Persistent Programming Language (PPL) becomes a language for describing two kinds of programming task.

Firstly, it is an appropriate language for programming data-intensive applications. The provision of orthogonal persistence removes the problems of low-level data storage from the application programmer's concern. The provision of graphical types enables the production of the user-interface in the same language as the rest of the program. At the same time, all the usual constructs for programming-in-the-large are available. However, a PPL can also be used for a separate, higher-order, task - the construction of data modelling tools, such as Semantic Data Models and Object-Oriented Data Bases. The programmer of such systems makes use of persistence and of the graphical types, but depends crucially on the provision of first-class procedures, implicit in the data type completeness of the language.

This two-level nature of programming in PS-algol can be confusing, so the first part of this chapter consists of an overview of the features of PS-algol which are typical of languages of the algol family. The second section will then describe the more unusual features of the language. The rest of the chapter consists of detailed consideration of a number of critical aspects of the language which affect its use as an application programming language, followed by some small examples.

3.1 An Introduction to PS-algol.

PS-algol is a block-structured language of the algol family. It is derived from S-algol, a language developed at the University of St. Andrews by Professor Ron Morrison [Cole and Morrison, 1982]. This in turn was derived from a language called algol-s built by David Turner at St. Andrews. The prime emphasis of these languages is on simplicity - a small number of constructs embodying a simple, universal, recursively applicable set of construction principles. The simplicity of PS-algol is one of its most attractive features. The power of the language has not been bought at the cost of complex semantics and a baroque syntax, but rather by increasing the power of already existing language features by extending their scope. For the programmer this results in a language which is very easy to learn. There are no exceptions to syntactic rules. There are no arbitrary restrictions on what may go where - if it makes sense to use an object in a given place, there will be no artificial barriers to doing so.

A formal definition of the language may be found in [PS-algol, 1987] and tutorials in the use of the language in [Carrick *et al.*, 1987] and [Cooper, 1987]. This section introduces the reader to some of the more familiar features of the language.

3.1.1 Values in PS-algol.

Values in the PS-algol world are strongly typed and may be divided into three kinds:

scalar values - such as integers, strings, etc;
composite values - such as vectors, procedures and images;
and **complex objects.**

The latter are structured data objects, in a sense which will be described in section 3.2.3 below, and are similar to Pascal records or objects in an Object-Oriented class, in that they have identity and updatable state, sharable via reference assignment semantics.

In fact, each element representing a value in a program can be considered to be a quadruple of the following attributes:

name: an identifier starting with a letter, then containing letters, digits or ".";
type: drawn from the type system described below;
value: a value from the domain of the type;
and **constancy**: this determines whether the value may be changed or not.

Of these, only the value of the element may change and that only if it has been declared to be variable and not constant.

Objects are introduced solely by **let** declaration clauses which introduce a new object, name it, determine its constancy, provide an initial value in the form of an expression and infer its type from the expression. For instance, consider the following two examples in which two scalars are declared:

```
    let x := 1
and let y = "abc"
```

Here, x is of type **int**, has initial value 1 and is a variable (the colon preceding the equals indicates this), while y is of type **string**, has initial value "abc" and this may never be changed (there is no colon).

3.1.2 The PS-algol Type System.

The scalar types of PS-algol include **int, string, real** and **bool**. These come with the usual operators for arithmetic, boolean and string manipulation (i.e. sub-string selection and concatenation) expressions. There are also scalar types for **files, pixels** and **pictures**. The first of these is to allow PS-algol to gain access to data stored outside of the persistent store, while pixels and pictures are the basic building blocks of the two graphical systems supported by the language.

These scalar types may then be combined using a number of type constructors:

- for any variable type, τ, the type cτ indicates the type of constant objects of type τ - e.g. **cint** is the type of all constant integers;

- the type **#pixel** is the type of an image which is a bitmap made out of a rectangle of pixels; (see section 3.2.2)

- for any types $\tau_1,...,\tau_n$ and τ, **proc**($\tau_1,...,\tau_n$ -> τ) is the type of a procedure having *n* arguments of types τ_1 ... τ_n and one result; (see section 3.1.5)

- for any type τ, the type *τ is the type of a vector whose elements are all of type τ; (see section 3.1.3)

Finally,

- there is a special type, **pntr**, which is the type of all complex objects constructed by a PS-algol structure. (see section 3.2.3)

There is no notion within this type system of union types, except for the predefined union type, **pntr**. For instance, there is no way of stating that the object named *x* is either a string or an integer, the precise type being left indeterminate until run-time. This facility is available in the language Napier88 [Morrison *et al*, 1988b]. Every value has a type drawn from the above type system which is fixed and is statically determinable at compile time. This has the extremely desirable effect of permitting the earliest detection of type errors, estimated by Buneman to constitute 70% of all programming errors [Buneman, 1988]. It would, however, seem to imply a lot of extra programming to overcome the lack of polymorphism. For instance, in order to provide list-handling facilities for lists of strings, lists of integers, lists of reals and lists of booleans, one set of procedures would seem necessary for each type. As will be seen, there are two mechanisms for circumventing this - the **pntr** type and the availability of the compiler as a system function.

3.1.3 Vectors.

Vectors are always composed of sequences of elements of single type - there is no mechanism for introducing vectors some of whose elements are strings while others are integers. As a consequence of the data-type completeness of the language, vectors can be of any type - vectors of constant integers, vectors of procedures or vectors of vectors of strings, for instance. This is an example of the simplicity of the language, combined with its power - the programmer can use vectors of procedures if they are needed and does not have to remember that they are not available.

Vectors of vectors require a little further explanation. The type ****string** represents vectors each of whose elements is a vector of strings. This is not the same as a rectangular array of strings, since each of these elements may be vectors of different length. Again an increase in the range of objects which can be represented is accompanied by no increase in complexity - the programmer only has to remember that vectors can be made of any set of objects of a common type.

3.1.4 The PS-algol Computational Model.

A PS-algol program consists of a sequence of clauses each of which is either an atomic clause or a block (a sequence of clauses or blocks delimited by **begin** ... **end**). Among the atomic clauses provided are: expressions; declarations; assignments; **for**-loops; **repeat...while...do, if..do, if...then..else** and **case** program control clauses; and input/output clauses. Atomic clauses are typed. An expression can be considered to be a simple kind of clause and such a clause has the type of the expression. Clauses such as assignments, which are untyped, are considered to be of type **void**.

The control clauses **repeat...while...do, if...then..else** and **case** can be used to produce typed clauses as their general syntax is, for instance:

if boolean-clause **then** clause1 **else** clause2

and the only restriction on clause1 and clause2 is that they be of the same type. Thus the two fragments:

if *X*<0 **then** *mod* := -1 **else** *mod* := 1

and *mod* := **if** *X*<0 **then** -1 **else** 1

are equivalent. The clause **if..do** is a degenerate form of **if...then..else** with a void alternative. The clause **repeat...while...do** is provided to allow termination condition testing to occur at the beginning, the end or in the middle of the loop.

Blocks are also typed. In fact, they consist of a sequence of clauses of which all but the last must be void. Then the type of the last clause in the block is the type of the block.

3.1.5 Procedures.

Procedures are introduced in the same way as any other value with a **let** clause, for instance the clause:

let *maxint* = **proc(** **int** *A*, *B* -> **int** **)**

which introduces a procedure of type **cproc(int, int -> int)**. A procedure may have any number of arguments, including none, and one or no result. The declaration clause must then be followed by a clause which constitutes the body of the procedure. The type of the clause, which is usually a block, must correspond to the result-type of the procedure. Two possible (equivalent) bodies to *maxint* are:

```
begin
        let max := A
        if max < B do max := B
        max
end
```

and **if** *A*>*B* **then** *A* **else** *B*

In checking the type of a procedure, the parameter names are ignored, so *maxint* has the same type as:

let *minint* = **proc**(**int** *D,E* -> **int**)

PS-algol procedures only come into existence at the end of the body, rather than at the end of the declaration. This means that recursive procedures cannot be written simply like

let *fact* = **proc**(**int** *I* -> **int**); **if** *I* = 1 **then** 1 **else** *I**fact(*I*-1)

but must be written

let *fact* := **proc**(**int** *I* -> **int**); **nullproc**
fact := **proc**(**int** *I* -> **int**)
if *I* = 1 **then** 1 **else** *I***fact*(*I*-1)

in which a dummy declaration of *fact* has been made (using the null procedure **nullproc**) so that some reference to *fact* can be made within the real body. Note here that PS-algol can be programmed entirely in a functional style if that is deemed appropriate for the application.

3.1.6 Miscellaneous Surface Syntax

Any text appearing on a line following a "!" is a comment.

The block delimiters **begin** and **end** may be replaced by "{" and "}" to make the code more concise.

Clauses may be separated by semi-colons, but these may be omitted if a clause ends at the end of a line. Thus none of the examples above show semi-colons, except the two versions of *fact* which separated the procedure specification and body by a semi-colon. Conversely, if a clause needs to be broken into two lines, care must be taken to ensure that the clause is not syntactically complete up to a line break or the rest of the clause will be taken as a new clause.

Programs should be terminated with a "?".

3.2 Advanced Features of PS-algol.

This section will introduce those features of PS-algol which are not typically found in other algol-like languages: the graphics facilities; the use of the extensible union type, **pntr**, to model complex objects and provide a degree of polymorphism; the mechanisms which provide persistence; the use of first-class procedures; and the availability of the compiler as a function at run time.

3.2.1 The Graphics Facilities 1 - Pictures.

There are two separate systems in PS-algol for handling graphical data and these are fully described in [Morrison *et al*, 1986a, Morrison *et al*, 1986b]. As described above, PS-algol provides corresponding types for the two systems. The type **pic** is the type of pictures constructed as line drawings in the Cartesian plane, while the types **pixel** and **#pixel** are the types respectively of single pixels and bitmapped images which are rectangles of pixels.

The **picture** handling facility is a version of the Outline system [Morrison 1982] and manipulates pictures which are logical rather than visible. The simplest kind of picture is the infinitessimal dot, which is introduced by a clause of the form:

```
let dot := [ 4.5, 5.6 ]
```

dot is now a single point at 4.5, 5.6 in two-space. More complex drawings are constructed by the recursive use of the following operators:

```
let two.dots:= [ 4.5, 5.6 ] & [ 6.7, 7.8 ]    ! & puts 2 drawings without joining
                                               !  them with a line.
let line:= [ 4.5, 5.6 ] ^ [ 6.7, 7.8 ]        ! ^ joins 2 drawings with a line.
```

Text can be put into drawings by clauses like:

```
let text.pic = text "hello" from 4.5,5.6 to 6.7, 7.8
```

which means place the string "hello" onto the Cartesian plane between points 4.5, 5.6 and 6.7, 7.8, rotating and scaling the text to fit.

Drawings can then be manipulated by the following clauses:

```
rotate drawing by 45                ! rotate the drawing clockwise 45°

scale drawing by 2.0, 3.5           ! scale the drawing

shift drawing by 2.0, 3.5           ! shift the drawing
```

Finally, the command

```
draw( im, drawing, 1.1, 3.5, 2.2, 5.7 )
```

maps the part of the drawing from 1.1, 2.2 to 3.5, 5.7 onto the image, *im*. This is the method used for making drawings visible. None of the applications called for pictures and so the picture system has not been used in any of the experiments reported in this thesis. However, at the very least PS-algol pictures seem a good mechanism for storing some kinds of pictorial data, maps for instance, and for constructing user interfaces on-line. The reader is referred to [Abdullah, 1990] for a use of the picture system to represent map data - pictures being particularly useful if zooming to arbitrary levels of detail is required.

3.2.2 The Graphics Facilities 2 - Images.

Pixels are objects which in the simple case have one of the values **on** or **off**. In fact, pixels also have depth, for instance **on & off & on** is a pixel object of depth 3 and individual "planes" of the pixel may be selected in a similar manner to sub-string selection. Images may similarly have depth, but this will be ignored in the discussion as no use has been made of depth here.

Images are of two kinds - base images and aliases. The former are introduced for instance as follows:

let *BaseIm* = **image** 10 **by** 10 **of off**

which creates a 10 by 10 rectangle of pixels all of which are off. This is, once again, a logical non-visible object - making it visible will be covered later.

Aliased images are introduced as in:

let *Quadrant* = **limit** *BaseIm* **to** 5 **by** 5 **at** 0, 6

which creates *Quadrant* to be the upper left quadrant of *BaseIm* (pixels are numbered from 0, 0 - the bottom left hand corner of an image). *Quadrant* is not a separate rectangle of pixels in its own right, but merely an alias of a quarter of the pixels in *BaseIm*. Therefore any change to any of the pixels in *Quadrant* will affect that quadrant of *BaseIm* and vice versa.

The contents of an image are modified by use of a set of 8 raster operations (the other 8 possible raster operations can be simply derived by combining two of these). The operations available are: **copy, not, xor, xnor, ror, rand, nand** and **nor** and they all have the general form:

rasterop *Im1* **onto** *Im2*

which causes the pixels of the image expression, *Im2*, to be modified by combining them one pixel at a time with the pixels of image expression, *Im1*, subject to clipping.

For the purposes of this thesis only three of these operators will be illustrated:

copy image 2 **by** 2 **of on onto** *Quadrant*

copies the new image onto the *Quadrant* . In this case, as the first image is smaller than the second, only the bottom left four pixels of *Quadrant* will be modified.

xor *Quadrant* **onto** *Quadrant*

executes an exclusive or of the first image onto the second. In this case, *xor*-ing an image onto itself has the effect of turning all the pixels off. This is the standard technique of clearing an image.

xnor *Quadrant* **onto** *Quadrant*

executes an exclusive nor of the first image onto the second. In this case, *xnor*-ing an image onto itself has the effect of turning all the pixels on. This is the standard technique of making an image completely black.

The use of these techniques will now be demonstrated with a fragment which turns *BaseIm* into a simple 2 by 2 chequer board, as shown in Figure 3.1:

```
let leftUpperQuadrant = limit BaseIm to 5 by 5 at 0, 5
xnor leftUpperQuadrant  onto  leftUpperQuadrant
let rightLowerQuadrant = limit BaseIm to 5 by 5 at 5, 0
xnor rightLowerQuadrant onto  rightLowerQuadrant
```

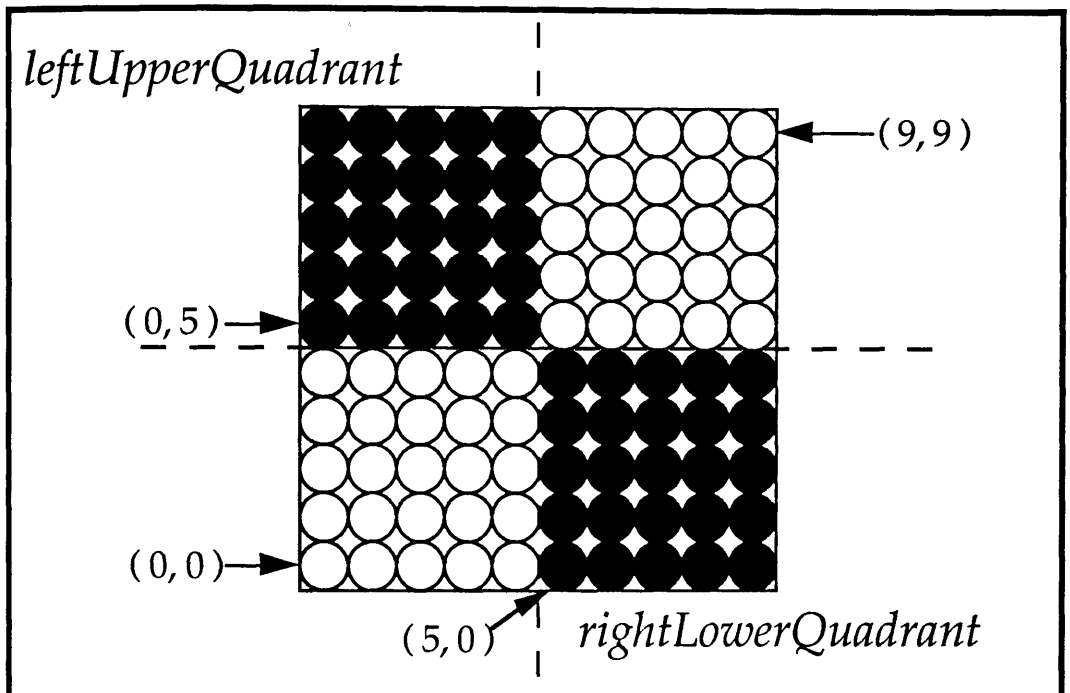


Figure 3.1 *BaseIm* with a Checkerboard Image.

So far, all of these objects are only logical and non-visible. In order to make them appear on the screen, two system objects are provided:

screen is a base image which contains the screen window in which the program was started;

while **cursor** is a small image containing the icon displayed as the cursor which tracks the mouse.

Images are made visible by copying them onto the screen or part of the screen - for instance:

```
copy BaseIm onto limit screen at 100, 100
```

will put the chequer board onto the screen at pixel 100, 100.

Some of the other facilities available using images are now briefly mentioned:

- the system functions *X.dim* and *Y.dim* return the x- and y- dimensions of an image;

- a **print** procedure, which rasters the value of any scalar onto any image in any font, using any of the eight raster operations [Philbrow *et al.*, 1988a];

- the function *string.to.tile* takes a string and a font-name and returns the image which contains the string displayed in that font;

- the function *menu* takes in a title image, a set of icons, action procedure pairs and a boolean to determine whether the menu is to be shown horizontally or vertically. It returns a procedure which when applied will put up a menu, wait until the mouse is clicked over one of the icons and then apply the associated action procedure. This is the first example of the kind of interface tool which can be easily built in PS-algol.

- the functions *cursor.off* and *cursor.on* make the cursor image invisible and visible respectively.

Finally, one other system function must be mentioned - *locator*. This procedure of no arguments tracks the mouse and returns a structure which contains the position of the mouse and which buttons, if any, are currently being pressed. Using the *set.locator* function, the functionality of *locator* can be modified, so that, for instance, it only returns when a mouse event (movement or button press) occurs. These tools were themselves written in PS-algol and *menu* and *string.to.tile* will be discussed in some detail in Chapter 4.

3.2.3 Structures.

PS-algol allows the construction of complex objects in a way that is superficially similar to the Pascal record-type. A class constructor may be defined, which consists of a name for the class and a set of names and types of the fields which make up the class. For instance:

structure *Address*(**int** *house*; **string** *street, city*)

defines a class of addresses consisting of three fields, one integer and two strings and a constructor for generating them. Then:

let *CS* := *Address*(17, "Lilybank Gdns", "Glasgow")

generates an instance of the class. The type of *CS* is **pntr**.

The fields of such an object can be dereferenced and re-assigned, as in:

let *CShouse* := *CS*(*house*)

and *CS*(*house*) := 18

The variable, *CS*, can be thought of as holding a reference to the object and so the assignment:

let *alsoCS* := *CS*

will create another reference to the **same** object. Any changes to any of the fields of *CS* will also appear as a change to the same field of *alsoCS*. The underlying semantics of the copy operation is therefore reference based. Two value based semantics may be built on top of this. **Shallow** copying is simply achieved by:

let *newCS* := *Address*(*CS*(*house*), *CS*(*street*), *CS*(*city*))

The implementation of **deep** copying is a more complex matter and a general purpose mechanism is developed in section 4.3.

The semantics of equality is similarly reference based. Thus the tests

alsoCS = *CS* and *newCS* = *CS*

return **true** and **false**, respectively.

As PS-algol is data-type complete, the fields of such a structure can be of any type. They can be vectors or procedures as well as strings or integers. In particular they can refer to other structure instances - i.e. objects of type **pntr**. Structures were introduced as a technique for modelling complex data. A simple example of this is the following structure which models a node in a list of strings:

structure *stringNode*(**string** *this*; **pntr** *next*)

Using **pntr** fields, structures of arbitrary complexity can be manipulated as single objects - the list of strings can be transmitted merely by passing the reference to the head of the list. It is easy to write a list-processing package to manipulate such a list. It might contain the following three procedures:

```
let StringList := nil                    ! introduces a variable which refers to the list

let ClearStringList := proc( )           ! clears the list
    StringList := nil

let AddStringList:= proc( string new )              ! adds a new string to the by making
    StringList := stringNode( new, StringList )     !  a new node at the head of the list

let PrintStringList := proc( )           ! print the list
  begin
    let P := StringList
    while P ~= nil do
      begin
          print P( this )
          P := P( next )
      end
  end
```

Figure 3.2 A List Processing Package for Strings.

This illustrates the fact that PS-algol contains some of the features of an object-oriented language. Complex data objects have identity and can be manipulated as a single entity. In this package, the variable *StringList* at all times contains a reference to the list as a whole because it refers to the head of the list.

However, the important feature is that all objects created as instances of any of these class constructors are of the same type: **pntr**. Therefore every **pntr** variable object or structure field can contain a reference to an object of any class. This means that programs can be written which manipulate objects without knowing which class they are in. This allows programs to be split into parts which handle objects of any class and parts which actually dereference the fields of the structure and so have to know what that structure is. To illustrate this, the list is generalised to contain different types of value at the nodes, for instance using the structure:

structure *anyNode*(**pntr** *this, next*)

Two packaging structures are introduced for integers and strings:

structure *stringPack*(**string** *stringValue*)
structure *intPack*(**int** *intValue*)

and the package can be re-written:

```
let AnyList := nil                      ! introduces a variable to hold the list head

let ClearAnyList := proc( )             ! clears the list
    AnyList := nil

let AddAnyList:= proc( pntr new )           ! adds a new object to the by making a new
    AnyList := anyNode( new,  AnyList )     !   node at  the head of the list

let PrintAnyList := proc( )              print the list
  begin
    let P  := AnyList
    while P ~= nil do
      begin
          print if P is stringPack      then P( this )( stringValue )
                                        else P( this )( intValue )
          P := P( next )
      end
  end
```

Figure 3.3 A Polymorphic List Processing Package.

The procedures which clear and add do not need to know what class of object they are handling. The calling program determines the class of the data which is to be entered into the list. Thus these procedures are truly polymorphic, but the style of writing polymorphic procedures explicitly identifies the token for the values, **pntr**, which is used in most implementations of parametric polymorphism [Cardelli and Wegner, 1985]. The advantage of exposing this is that there is also a structure with polymorphic fields, which is exploited extensively.

Conversely the print procedure does need to know the class of object, so that it can correctly dereference the information contained. This procedure has also been written to be polymorphic, but this time the polymorphism is bounded to the two packaged classes *stringPack* and *intPack*. If any other class is encountered, an error will be generated. The form of bounded polymorphism seen here is that the parameter is bound to an explicitly stated set of classes.

What has been achieved by splitting the package is increased software re-usability. The amount of recoding to be done is minimised when the list is increased to include other sorts of object, by deferring the binding of program to data-type as much as possible - in this case until the list has to be printed. This technique of deferring the binding is one which is used over and over again in PS-algol programs. As well as reducing the amount of coding and re-coding, it has the effect of clarifying the nature of the operations which are being coded. The decision to defer the binding to another part of the program is a conscious one, which can only be made in the light of a full understanding of what the program is trying to achieve.

The **pntr** type has effectively partitioned the type space in PS-algol. On one side there are scalars, vectors, images and procedures and on the other are the PS-algol classes. The types of the former are checked at compile time, while the check of classes is deferred until instances are used. Deferring the type checking of programs is essential for the incremental development of complex applications and for allowing applications to be rebound to new databases [Atkinson *et al*, 1988].

3.2.4 Tables.

The "table", a system supplied example of a PS-algol structure, is used to illustrate further the notions of structures and of deferred binding. The table structure is of particular importance as it is used as a principal component in the persistence mechanism of PS-algol and this will be described next.

A table in PS-algol is a structure which contains a set of one-to-one mappings from strings or integers to objects of type **pntr**. This associative mechanism is used to build keyed sets of objects. For instance, there might be a table of addresses, in which the string "CS" is inserted as the key paired with the object *CS* declared in section 3.2.3. Operations are provided to create an empty table, to insert such a pair, to retrieve the object from the key and to apply a procedure to all the objects in the table. These will now be described.

The command:

```
let T = table()
```

creates an object of type **pntr** which refers to an empty table. Into this table could then be inserted the pair described above with:

```
s.enter( "CS", T, CS )
```

which puts the string key, object pair into the table. The object may then be retrieved by the command:

```
let retrievedCS = s.lookup( "CS", T )
```

Finally, the command:

```
let n = s.scan( T, P )
```

applies the procedure, *P*, to every pair in the table. The procedure, *P*, must be of type **proc(string, pntr -> bool)** where the two arguments are local variables which hold the values of the string key and paired object. The procedure returns a boolean, which is usually **true**, but may alternatively be **false** to bring the scan to a premature end. As an example the fragment:

```
let double = proc( string S; pntr V -> bool )
  begin
      V( house ) := 2 * V( house )
      true
  end
let n = s.scan( T, double )
```

will double the house number of every address in the table, since it will apply the procedure with *V* pointing to each object in the table in turn. There are also operations, *i.enter*, *i.lookup* and *i.scan* which behave equivalently with integer keys.

This is a clear use of the notion of deferred binding. These facilities can be used to create tables which contain any class of data or any mixture of classes of data. The tables are built up blind, with the operators never knowing what sort of data they are manipulating. Therefore tables are available to as yet undreamt of programs, handling original data classes without needing to be rewritten at all. The table procedures are themselves written in PS-algol.

3.2.5 The Persistence Mechanism.

Tables are of particular importance because they are used as the structure within which persistence is implemented. The notion of persistence in programming, as described in section 2.3, is that the effort to make data outlive the program is minimised. In PS-algol, the mechanism for making data persist is its insertion into a structure which is reachable, by following **pntr** chains, from some persistent root, which will itself be an object of type **pntr**. Since any data object may be put into a structure, any data object may be made to persist and so the provision of persistence is orthogonal to data type.

A persistent root of the PS-algol system is called a "database". A database has a table structure, although in theory this could be replaced by any bulk data structure into which objects can be inserted. In some implementations there is only one database, in others there can be many - the more general latter case will be described. Control of concurrent data access is fixed at the database level - that is, any database can be open to many readers or one writer. This is the only notion of concurrency control in PS-algol and an inadequate one (see section 9.6.6).

Databases are created by commands such as:

```
let DB = create.database( "mydb", "mypass" )
```

The system function, *create.database*, returns a table, which is called the top-level table of the database and which will automatically persist if the *commit* function is applied. The creating program has write-access to the database and may now start entering key, value pairs into it. When *commit* is executed, any entries in the table will be made to persist and, if the objects in the table have pointers to other objects (they may for instance themselves be tables), those objects will also persist. In fact, all objects reachable from the top-level table via pointer chains will be made to persist.

Causing any new or updated objects to persist is achieved with a clause like

```
if commit() is error.record      then print "commit failed"
                                 else print "commit succeeded"
```

where *commit* is a procedure which returns **nil** if it has succeeded and an *error.record* structure if not. The latter could be caused by lack of disc space or by trying to commit changes to a database to which the program does not have write-access. Commit can be applied at any time in the run of the program and when it has succeeded it may not be undone. Commit makes permanent all changes to all databases open with write-access - there is no notion of partial commit. Both of these restrictions are important when considering the structure of a database application, such as the Bibliographic Reference Database described in Chapter 5.

Having been created, a database can be re-opened with

```
let DB = open.database( "mydb", mypass", "read" )
```

which again returns the top-level table, this time with only read-access. To get write-access instead, the "read" should be replaced with "write". If the database is already open for writing by someone else, the command will return an *error.record*, as it will if it is open for reading and the program tries to open it for writing. There are also system functions to allow databases to be deleted and have their names or pass-words changed.

This short section describes the whole persistence mechanism, which is simple and elegant. For instance, to make the string list persist all that is required is to put the list header into a structure reachable from a database and to perform commit. The whole of the list is then saved, since every element is reachable from the head of the list. The list can be retrieved by re-opening the database and performing sub-object dereferences until the head of the list is retrieved.

3.2.6 First Class Procedures.

Here the underlying principle which makes PS-algol such a powerful language is re-iterated - data-type completeness. Its effects have been seen in a number of ways in the preceding sections:

- since the fields of structures can be of any type, they can be of type **pntr** and so data of arbitrary complexity can be modelled;

- since vectors can be of any types, there can be multi-dimensional vectors which are not purely rectangular;

- since data of any type can be put into the structures which are held in databases, all data types have the same right to persist;

and
- most importantly, the programmer's world is greatly simplified, since there is no need to remember arbitrary restrictions.

This section concentrates on the effect of first-class procedures [Atkinson and Morrison, 1985a] - i.e. procedures which can be manipulated in the same way as any other object. It is one of the main contentions of this thesis that first-class procedures are an invaluable tool in writing complex application programs. That "object-oriented" languages like Smalltalk choose to force procedures into a second-class rôle as "methods" will provide the crucial limit on their usefulness and hence durability.

To reiterate, in PS-algol, procedures can be the values of variables, the fields of structures and the argument or result of other procedures. One example of this has been seen in the *s.scan* function which takes, as one of its arguments, the procedure to be applied. Another example is the *menu* function which takes as arguments, among other things, parallel vectors of icon images and associated procedures. *menu* then builds a procedure which calls *locator* to see where the mouse is and as soon as it has been clicked over one of the icons, the associated procedure is applied. This built procedure is then returned by *menu* as its result. The type of *menu* is **proc(#pixel, *#pixel, *proc(), bool -> proc(int,int))**. That is, its arguments include the title image, the icons, the actions and whether it is horizontal or vertical, and its result is a procedure which takes in an x, y position at which the menu is to appear and then displays it. These kinds of facilities would be very much more difficult to construct if procedures could not be referred to in isolation.

Section 3.4 will describe some of the advantages of first-class procedures, but here is a brief list of their uses:

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

- the production of abstract data type representations of data, since packages of procedures can be returned from data creation procedures;

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

3.2.7 The Callable Compiler.

The existence of first-class procedures in the language has meant that it has also been possible to provide a function which calls the compiler. A string may be built which represents a procedure. This can be passed to the compiler function, which returns the compiled procedure. This may then be used in the same way as any of the statically written procedures of the program, as in Figure 3.4.

```
structure procHolder( proc() theProc )
let Pstring = "proc(); write 12345 "
let emptyHolder = procHolder( proc(); nullproc )
let compHolder = compile( Pstring , emptyHolder )
let Pcompiled = compHolder( theProc )
Pcompiled()
```

Figure 3.4 Run-time Compilation

This will have the same effect as

```
let Pcompiled = proc(); write 12345
Pcompiled()
```

In the example of using the callable compiler, note that an empty structure was provided into which the compiler can put the compiled procedure. This is because the compiler has been written to be a completely general function which can compile procedures of any type - another example of deferred binding. The mechanism which has been used to defer the choice of the type of the procedure is to force the caller to specify this by providing a package of the correct type - after all, the user knows what type it is. This means that although the compiler does not know the type of procedure it is to compile until it is actually called, the type of the resulting procedure is known at compile-time and so can be statically type-checked. To put it another way, the choice of type has been deferred when writing the compiler function until writing the calling program.

To use the compiler as just illustrated would clearly be of little value - there is no point in compiling a program every time it is run. The mechanism comes into its own when writing programs designed to run against an unbounded set of different classes of data. This will be illustrated in section 3.3.10 and used extensively throughout this thesis.

3.2.8 Exceptions.

PS-algol includes an exception mechanism [Philbrow *et al.*, 1988b] which permits program events, which the current procedure is not designed to handle, to be passed into successive outer blocks, until code which can handle the event is found. The exception mechanism is based on that designed for the CLU language [Liskov and Snyder, 1979], but makes extensive use of the **pntr** type.

In fact an exception in PS-algol is an instance of a PS-algol class. An exception class is created in the same way as any other. Take for instance an application which reads some data from a file and requires an exception for the file being exhausted. The following structure is created:

```
structure fileExhausted( string whichProc )
```

which is used as in the following:

```
raise fileExhausted( "Reading a number" )
```

The effect of this statement, which appears in some low-level procedure which reads a number, is to halt execution of the current block and pass execution back into successively higher blocks until one is encountered which contains some error handling code. This looks like:

```
when fileExhausted as FE do
        print "The file is exhausted in procedure: ", FE( whichProc )
```

The execution resumes at the end of the block containing the error handler.

The use of this technique is to avoid the necessity for low-level procedures to return exception information by use of the normal parameter passing mechanism. For instance, in writing a compiler without exceptions, the character input procedure and each procedure that calls it would have to pass back failure codes when the file was exhausted. Using exceptions, execution immediately returns to a higher-level block where the condition is handled. The system supplies a number of exceptions for common occurrences.

3.3 The Advantages of the PS-algol Approach.

The features described in the previous section give a number of clear advantages when programming a large scale application. The reader is referred to [Atkinson and Morrison, 1985a, Atkinson and Morrison, 1985b, Morrison *et al*, 1986b, Morrison *et al.*, 1988a and Atkinson *et al.*, 1988] for fuller treatment of these and other issues.

3.3.1 Low level data management is handled for you.

In a persistent system, the sections of program concerned with the organisation of data for input and output are redundant - a survey by IBM Ltd. estimated that 30% of the programming effort in producing a large program typically goes into this part of the program [IBM, 1978]. In PS-algol, the organisation that is imposed on the data in order to handle it within the program is the structure in which it is stored. To store a list of strings, all that is required is to enter the head of the list into the database. To take another example, consider a program dealing with relational databases, in which a single structure contains a header for the relation. The mechanism for storing a relation in the persistent store consists merely of entering a pointer to this structure into the store. All of the data in the database is then pulled into the persistent store as a consequence of being part of the data structure which is referred to from the header.

A further saving of programmer effort is that there need be no recourse to external data handling programs, such as file management systems, with a consequent saving in the amount of information the programmer needs. In fact, this is an example of a more general advantage of using a persistent language. All programming jobs can be done in the same language since the language exists in the context of a unified world. Reducing the complexity of the programmer's world confers a benefit if the programming is, of itself, necessarily complex.

3.3.2 Strict type checking gives early detection of data mis-use.

As mentioned before, type information is stored along with the object. This means that it is not possible, for instance, to store a numerical object and re-load it as a string. Attempts to mis-use data in this way are detected at program compilation time, data loading time or data reference time, but always before the data is used. The programmer therefore discovers any error at the earliest possible time, which leads to a consequent saving in the time to develop a system.

3.3.3 The graphics facilities provide tools to produce user interfaces.

In producing programs for the software market today, a great deal of attention must be paid to the user interface. Having the necessary tools to produce a good interface within the language is of great benefit. Using external packages is fraught with the problems of forcing unnecessary constraints on programs and of restricting the kinds of interface that can be provided. Again, the programmer has been relieved of learning other languages - the one used to program the package and the one provided by the package to interact with the provided operations. Within PS-algol, not only are a number of sophisticated tools provided, but the existence of the two graphics types within the language permits users to provide themselves with their own set of tools, at small cost. For instance, if the PS-algol pop-up menu is not what is required, the user may create a personally tailored one.

Using PS-algol, it is relatively simple to produce: a variety of menu- and form-interfaces; window management systems; good quality iconic interfaces; and direct manipulation tools. The development of these is discussed in Chapter 4.

3.3.4 Image and picture objects model graphical data.

Having the types, image and picture, alongside such traditional types as integer and string, allows graphical data to be stored in exactly the same way as textual or numerical data. Clearly, this is of value if the application is actually handling pictorial data - maps, for example. It relieves the programmer from having to invent a coding strategy to handle the pictures. If an object requires an iconic interface, this can then appear as one of the fields of the structure which contains its attributes.

3.3.5 First-class procedures model actions.

As stated previously, procedures can also be manipulated in the same way as graphical, numerical and textual data. Thus it is possible to model and store activities. This is of interest, for instance, in modelling office systems. It is very useful to be able to refer to objects which model office procedures. If these objects are themselves compiled procedures then two advantages appear - they run efficiently and they correspond closely to the object they are trying to model (thus, once more, reducing the complexity of the program).

Assertions, conditions or triggers may also be modelled by procedures which take in values for the variables of the assertion and return a boolean result. Further,

objects which are pairs of condition procedures and action procedures can be created. It is then simple to write a program fragment which loops, testing conditions and applying the paired action if the condition is **true**.

Another example of the use of first-class procedures is given in [Cooper, 1987] in the context of an implementation of the video-game, Snake. In this game, the player controls a snake as it moves in one of the four directions: up, down, left or right. At any time, the snake's direction may be changed by 90 degrees by pressing a mouse button. The program could be written with a flag indicating which direction the snake is moving in and then contain several tests of this flag. Conversely it could use five procedure variables:

- *move* - this contains a procedure which moves the mouse one unit. Its value will be one of the four constant procedures *up, down, left* or *right.*

- *varChangeUp* - this is the procedure which will be applied when the up button is pressed - if the current direction is up or down, it will do nothing, if the current direction is left or right its value will be *ChangeUp,* which will be a constant procedure which sets *move* to *up,* and changes all the four *varChange* procedures appropriately;

- similar procedures *varChangeDown, varChangeLeft* and*varChangeRight.*

Programming using these procedure variables was found to be simpler and faster than using flags.

3.3.6 First-class procedures facilitate incremental compilation.

Having developed a program in the classical structured way, PS-algol provides an ideal framework in which to program the modules. Each module is written as a PS-algol procedure and stored in the persistent store. Dependent modules can then access this module by retrieving it from the store, using any one of a number of different binding styles. Chapter 8 provides much more detail on this.

Several benefits accrue from this:

- source modules can be kept short with a consequent saving in compilation and debugging time;

- new versions of modules can replace old ones without having to re-compile or re-run the whole of the program;

- there is no need for separate library database mechanisms, type checking linkers or loaders, since procedures are treated like any other value - thus the programming environment is simplified and the system implementation task is reduced;

and

- alternative versions of the same module can be provided - for instance, a number of editors could be stored and the user could select which one to use by menu. If another editor were added, it could be made to appear automatically on the menu.

3.3.7 First-class procedures facilitate Abstract Data Types.

The notion of the Abstract Data Type is the restriction of access to data to a set of operations defined on it. Since procedures can be put into structures and structures can be returned as the results of procedures, ADT generating procedures can be constructed which return a structure containing a package of procedures. To illustrate this, the string list package could be produced as one procedure which is called to generate a new list (Figure 3.5).

```
let newStringList = proc( -> pntr )
  begin
     structure stringNode( string this; pntr next)
     let StringList := nil                ! introduces a variable to hold the list head

     let ClearStringList := proc( )       ! clears the list
           StringList := nil

     let AddStringList:= proc( string new )          ! adds a new string to the by making
       StringList := stringNode( new, StringList )   !  a new node at the head of the list

     let PrintStringList := proc( )
       begin
           let P := StringList
           while P ~= nil do
             begin
                print P( this )
                P := P( next )
             end
       end

     structure stringListPack( proc() clear; proc( string ) add; proc() print )
     stringListPack( ClearStringList ,AddStringList ,PrintStringList  )
  end
```

Figure 3.5 A String List Processing Package as an Abstract Data Type

The string list can then only be used through its operations, i.e.

```
let SLadt  = newStringList ()
SLadt( clear )()
SLadt( add )( "one" )
.....
SLadt( print )()
```

there being no other way to access the list. This prevents users from corrupting the list. This technique is very similar to "Object-Oriented" objects, which consist of an invisible state and a visible set of methods. This technique will later be extended to implement an Object-Oriented system in PS-algol (Section 7.4).

3.3.8 The pntr type models complex data.

Since PS-algol is data-type complete, the fields of a data structure can be of any type. This means that complex data objects can be constructed which combine

numbers, textual information, graphical data, activities and assertions. Furthermore, objects with a more complicated structure can be modelled by using pointer fields to sub-objects. Lists, trees and graphs of all kinds are simple to manipulate and traverse in PS-algol. Even baroque structures like the data for a cricket match can be well handled using structures.

3.3.9 The pntr type permits delayed binding of programs to objects.

As the language is strictly type-checked, the type of each object in a program must be specified before it is used. However, it is possible to write general purpose procedures which manipulate objects of a number of types by packaging the objects of different types into structures and passing around pointers to those structures. Thus, for instance, it is possible to provide a list processing package in which the list contains a number of types. The insert procedure would use a pointer to a package as its argument and could be used as in:

```
structure intPack( int intValue )
structure stringPack( int stringValue )
insert( intPack( 1 ) )
insert ( stringPack( "two" ) )
```

The elements of the list can be passed around without reference to their type, until the values are required, for instance in a procedure which prints out the contents of the list. Only then, is it required to check the type.

```
let Pthis := P( this )
let CI  = class.identifier( Pthis  )          ! returns the class structure as a string
let fieldNames  = ....                        !       derived by string manipulation
let source := "proc( pntr PP )                ! beginning building the source as a string
              begin
                 structure " ++ CI ++ "'n"    ! 'n means newline
for i = 1 to upb( fieldNames ) do
    source := source ++ "print PP(" ++ fieldNames  (i ) ++ ")'n"
source := source ++ "end'n"                   ! the end of the source

structure PrintHolder ( proc(pntr) PrintProc )
let emptyPackage = PrintHolder (proc(pntr  X); nullproc)
let CompiledProcPack=compile( source, emptyPackage  )
let CompiledProc = CompiledProcPack( PrintProc )          ! unpackage the proc.
CompiledProc ( Pthis  )                                   ! and apply it (at last!)
```

Figure 3.6 A Fully Polymorphic List Printing Procedure

3.3.10 The run-time compiler facilitates polymorphism.

The run-time compiler allows programs to be written which run against an unbounded set of data classes. Such programs are written so that they discover the class of data they are expected to deal with this time and, using string manipulation, merge the class information with the algorithm and compile the resulting procedure. For instance, the print procedure of the package for list processing *anyNode* lists might be generalised by replacing the clause which does the printing as in Figure 3.6.

To explain this, the class description of the object to be printed is discovered by use of the standard function, *class.identifier*, which returns it as a string. This information is manipulated to derive the field names of the class and these are then embedded into some code to print the fields. The variable, *source*, holds the source of a procedure to do this printing. In the case of the address structure introduced in section 3.2.3, the following source would be produced:

```
proc( pntr PP )
    begin
        structure Address( int house; string street, city )
        print PP( house )
        print PP( street )
        print PP( city )
    end
```

where the parts which are underlined depend upon the particular class and have been embedded into a template representing the printing algorithm.

Once compiled, the program would be stored so that it need not be regenerated every time an object of that class is encountered. Tables with the procedure stored against the class identifier as a key can be used to "memo-ise" the function. This binding together of algorithm and data class information to create flexible programs will be seen a number of times. Essentially it has two uses. Firstly, polymorphic programs can be written which will extend to any subsequent data type. Secondly, programs can be written which are efficient for any data structure without relying on interpretation which is intrinsically slow.

An earlier thesis from the Persistent Programming Research Group [Owoso, 1984] noted the need to program "universally applicable" algorithms when behaviour depends on the type structure of the values being manipulated. At the time, this did not appear possible in a fully type checked, largely statically bound language. This combination of the **pntr** and the callable compiler not only enables this form of programming, but also makes efficiency possible.

3.4 Conclusions.

This chapter has described those features of PS-algol which make it attractive for data-intensive programs. They include the provision of orthogonal persistence, graphical data types, the ability to model complex objects, the availability of first-class procedures and the run-time compilation system. Together these facilities provide a sufficient set for use in the programming both of database applications and of data modelling tools.

The next chapters continue this description with some examples of PS-algol programming: an application programmed in PS-algol; an efficient relational system; some semantic data modelling tools; and finally a use of the language to provide better application development environments.

Chapter 4. Building Tools in PS-algol.

In the last chapter, the basic facilities of the language PS-algol were introduced. In this chapter, the functionality of the language is extended by building tools which assist the programmer. Firstly, some user interface tools will be introduced. Then the PS-algol Database Browser will be described, followed by a description of sets of tools for dealing with complex objects and compiler generation. The chapter will also provide a tutorial in some of the basic techniques of PS-algol programming.

There are three ways in which these tools can be provided - as stand-alone programs, as "public standard functions", or as user-defined procedures stored as objects in the Persistent Store. Examples of the three types described in this chapter are the Browser, the *menu* function and the Chooser. The standard functions are automatically available without extra programmer effort, but can only be inserted by reconfiguring the PS-algol system - a privileged operation unavailable to applications programmers. This is, of course, the same for most programming languages. However, user-defined procedures can easily be inserted into the Persistent Store and the system functions can act as templates for these.

It is very useful to adopt some discipline when inserting utilities. The PS-algol system provides complete freedom in the way procedures may be stored in the database. Therefore any appropriate framework within which to insert user-defined operations may be created. Chapter 8 describes one method of organising a library of utilities and further functions which could be added, such as configuration management and version control. This chapter concerns itself solely with the kinds of tools themselves.

The principal demonstration of this chapter is the ease with which generally available functions can be added to the system. An application programmer can create new procedures which have different functionality to the ready-made software. The resulting applications are not then constrained by the restrictions of the standard facilities.

To summarise, this chapter includes the following:

- a demonstration of the usefulness of the primitives of PS-algol;
- a tutorial in the currently accepted methods for using these primitives;
- a demonstration of the ease with which the system can be extended by any user;
- an illustration of how complex programs can be written which are usually outside the range of a strongly typed programming language;
- some criticism of the facilities provided.

4.1 User-Interface Tools.

The graphics facilities of PS-algol were described in sections 3.2.1 and 3.2.2 above and this section shows how they may be used for constructing user-interface tools.

The section demonstrates how some of the tools have been built, starting with two standard functions, *string.to.tile* and *menu*. Then more sophisticated menu tools, a form interface, some textual display tools and a simple string editor are described.

4.1.1 Multi-Font Display.

The use of different fonts creates a more interesting user interface and helps to convey information - for instance, the relative importance of the text. PS-algol defines a structure for fonts and provides a standard procedure for using this structure to transform a string into an image. The structure is

```
structure font(     string fontname;
                    int font.height;
                    *#pixel the.chars;
                    string description )
```

where the most significant field is the *the.chars* field, which is a vector of images containing one image for each of the 127 characters in the PS-algol character set. A specific database, called "FONTS", is set aside to hold these structures.

This is used by *string.to.tile*, as in the following:

```
let helloImage = string.to.tile( "hello", "cou20" )
```

which will create the smallest image which will hold the word "hello" in the font *Courier 20*. Figure 4.1 gives the *string.to.tile* procedure.

```
let string.to.tile = proc( string S, F -> #pixel )
  begin
    let FontDB = open.database( "FONTS", "friend", "read" )
    let theFont = s.lookup( F, FontDB )     ! Find the font.
    let theWidth := 0                       ! Used to calculate the width of the image.

    for i = 1 to length( S ) do             ! Calculate the total width.
            theWidth := theWidth + X.dim( theFont( the.chars )( code( S( i | 1) ) ) )
                                            ! Create the image initialised to white.
    let theImage = imagetheWidth by theFont( font.height ) of off

    theWidth := 0
    for i = 1 to length( S ) do             ! Fill the image with the characters.
      begin
        let theChar := theFont( the.chars )( code( S( i | 1) ) )
        copy theChar onto limit theImage at theWidth , 0
        theWidth := theWidth + X.dim( theChar )
      end
    theImage                                ! Return the image.
  end
```

Figure 4.1 The Standard Procedure *string.to.tile*.

In this procedure, an image is created to hold the text. The width of the image is the sum of the widths of the constituent characters and its height is obtained directly from the height field of the font. The required characters are then copied into the image, which is the returned result of the procedure.

This simple procedure implements the facility without the expected recourse to low-level programming. The structures of the PS-algol graphics system are such that the manipulation of graphical objects has a similar feel to familiar manipulations of textual or numerical values. Furthermore, the procedure will work without modification, re-compiling or even re-loading, if the contents of the fonts database are changed. It will use any font that is in the database at run-time. The writing of this procedure has been entirely separated from the choice of font.

4.1.2 Textual Display Tools.

Consider first a message facility which will put a message in a box of a given size at a given place and then wait for a mouse click on button 1 before it is removed. The procedure is used as for instance in -

message("You shouldn"t have done that", 100, 100, 400, 50)

which displays the message in a box whose dimensions will be 400 by 50 at 100, 100. An additional convention is introduced: if the x-value of the origin is -1, this means use the centre of the screen, while if the x-value of the size is -1, then the smallest box which will contain the message with a border of 10 pixels of space around it is used. Figure 4.2 contains such a procedure.

```
let message = proc( string  mess; int xo, yo, xs, ys )
  begin
     let imess = string.to.tile( "mess", "met22" )          ! Convert the message into
     let xim = X.dim( imess );  let yim = Y.dim( imess )    !   an image and find its size.

     if xs = -1 do {xs := ximess + 20;  ys := yimess + 20 }  ! Border of 10 pixels if x = -1
     if xo = -1 do {   xo := ( X.dim( screen ) -xs )  div 2  ! Origin set so that the
                       yo := ( Y.dim( screen ) -ys )  div 2 }  !   centre of the image is the
                                                            !   centre of the screen.
     let xl = ( xs - ximess ) div 2                  ! The border dimensions - also where
     let yl = ( ys - yimess ) div 2                  !   in the box the message appears.

     let box = limit screen to xs by ys at xo, yo    ! The location of the box on the screen
     let save = image xs by ys of off                ! The screen area to be remembered.
     copy box onto save                              ! Remember the screen image.
     xnor box onto box                               ! Clear the box to black.
     let inner = limit box to xs - 4 by ys - 4 at 2, 2  ! The interior of the box.
     xor inner onto inner                            ! Set the interior to white, thus
                                                     !   leaving a two pixel line.
     copy imess onto limit box at xl, yl             ! Copy the text into the box.
     let maxwell := locator( )
     while ~maxwell( the.buttons )(1) do maxwell := locator( )! Wait for button press.
     while maxwell( the.buttons )(1) do maxwell := locator( )! Wait for button release.
     copy save onto box                              ! Restore the screen.
  end
```

Figure 4.2 A Message Display Procedure.

This procedure creates an image for the message and then resets the size and origin, if they have been entered as -1's. It then picks out the required part of the screen as *box* and saves the current contents of that part of the screen as *save*. Then it

puts an empty box into that area by blackening the whole rectangle (**xnor**) and then whitening everything except the outermost two pixels (**xor**). This leaves a frame two pixels thick. Then it copies the message into the box and waits until the mouse button has been clicked and released before removing the error message (actually copying the saved contents back into position). The mouse is monitored by calls to *locator* - a function which returns a structure containing the current mouse position and a vector of booleans which specify which buttons have been pressed. The procedure keeps calling *locator* until the mouse button has been pressed and then keeps calling *locator* until it is released again.

Another textual display tool is based on the UNIX™ *more* facility. It takes in a vector of strings representing the text to be displayed and an origin. It then displays the first "page" - the first 15 strings - in a box and uses the mouse buttons so that one button displays the next page, if there is one, another displays the previous page, again if there is one, and the third button quits the display removing it from the screen. A procedure to provide this is given as Figure 4.3.

```
let more = proc( *string text; int xo, yo )
  begin
    saveScreen()
    let maxLines = 15                              ! The page size.
    let noLines = upb( text )
    let firstLine := 1
    let lastLine := min( noLines, maxLines)
    showText( text, xo, yo, firstLine, lastLine )  ! Show first page.

    let buttons := locator()( the.buttons )        ! Which buttons are pressed?
    let finished := false
    while ~finished do
      begin
        case true of
          buttons(1): if firstLine ~= 1 do         ! Back one page.
            {   lastLine := firstLine-1
                firstLine := max( 1, firstLine-maxLines )
                showText( text, xo, yo, firstLine, lastLine ) }
          buttons(2): if lastLine ~= noLines do    ! Forward one page.
              { firstLine := lastLine+1
                lastLine := min( noLines, lastLine+maxLines )
                showText( text, xo, yo, firstLine, lastLine ) }
          buttons(3): finished := true             ! Set termination condition.
          default: {}
        buttons := locator()( the.buttons )        ! Re-sample the buttons.
      end
    replaceScreen()
  end
```

Figure 4.3 A *more* Facility.

This code assumes five procedures: *min* and *max* to provide the larger and smaller of two integers, *showText*, which displays a subset of strings from a vector at a given point on the screen, and *saveScreen* and *replaceScreen*, which store and restore the screen. The code for the *showText* procedure is very similar to that for the *message* procedure given above.

Figures 4.2 and 4.3 show once more how a few simple constructs can be combined to create general purpose user interface components in short procedures. The facilities are sufficiently simple and powerful to allow the resulting procedures to reflect the algorithm, without distracting details obscuring it.

4.1.3 The PS-algol Menu Function.

The power of menu-based interfaces is well established. For many purposes, and particularly for naïve users, they are superior to command language based systems. The standard procedure, *menu*, of PS-algol provides a function for generating menus. Its use can best be seen from a code fragment in Figure 4.4.

```
let finished := false;      let done := false
let title = string.to.tile( "Pick Command", "fix13" )
let icons = @ 1 of #pixel [ string.to.tile( "Add", "fix13" ),
                            string.to.tile( "Delete", "fix13" ),
                            string.to.tile( "Quit", "fix13" ) ]
let actions = @ 1 of cproc( int, #pixel ) [                    ! A vector of procedures.
                            cproc( int I; #pixel J )
                                    .....    Code for add,
                            cproc( int I; #pixel J )
                                    .....    Code for delete,
                            cproc( int I; #pixel J )
                                        finished := true ]
let Menu = menu( title , icons , icons , true )
while ~finished do done := Menu( 100, 100 )
```

Figure 4.4 An Example of Using the *menu* Procedure.

In this piece of code, four input parameters to the *menu* system function are specified - an image containing a title icon; a vector of images containing choice icons; a vector of procedures specifying the actions to be associated with the icons; and finally, a boolean specifying whether the menu is to be presented horizontally or vertically. The function returns a procedure of type **cproc(int, int -> bool)**, which when executed will display the menu at a position specified as the two arguments. When one of the icons has been selected, the associated procedure is applied and the procedure exits with the value **true**. If the mouse is clicked over some other part of the screen, it exits with the value **false.** The action procedures are forced to have type **cproc(int, #pixel)**, so that inside them the integer position in the menu and the icon are also available - (i.e. if the "add" function is picked the first parameter will have the value 1, while the second has the "add" icon). The author has never personally found a use for these, but they may sometimes come in handy.

This facility is constructed from the simplified version of the *menu* procedure given in Figure 4.5. The procedure builds the image containing the menu from the title and action icons. Then it returns a procedure which: saves the screen under the menu; waits for the mouse to be clicked and released; checks if it has been clicked over the menu, returning false if not and finding which icon has been chosen and applying the associated procedure if it is; finally it restores the screen contents, before quitting with **true** or **false**.

```
let menu = proc( #pixel title; *#pixel icons; *cproc( int, #pixel ) actions;
                                        bool vertical -> proc( int, int -> bool ) )
  begin
    let theWidth := X.dim( title )              ! Calculate the size of the image.
    let theHeight := Y.dim( title )
    for i = lwb( icons ) to upb( icons ) do if vertical
            then { if theWidth < X.dim( icons( i ) ) do
                                    theWidth := X.dim( icons( i ) )
                 theHeight := theHeight + Y.dim( icons( i ) ) }
            else .... equivalent for horizontal case ....
    let theImage:= image theWidth by theHeight of off      ! Create the image.
    .... copy title icon into theImage ....
    for i = lwb( icons ) to upb( icons ) do if vertical
            then .... copy icon vertically ....
            else .... copy icon horizontally ....

    proc( int X,Y -> bool )                          ! The returned procedure
      begin
        .... save the screen area and present the menu (as Figure 4.2) ....
        ....      wait for button press and release (as Figure 4.2)      ....
        let Xmouse = maxwell( X.pos )                ! The mouse position.
        let Ymouse = maxwell( Y.pos )
        if Xmouse < X or Xmouse > X + theWidth or
             Ymouse < Y or Ymouse > X + theHeight
             then { .... restore saved screen area ....; false }   ! Mouse not over menu.
             else begin
                  let choice = if vertical                ! Find the index of choice.
                       then .... function of Ymouse and heights of icons ....
                       else .... function of Xmouse and widths of icons ....
                  actions( choice )( choice, icons( choice ) )      ! Execute action.
                  .... restore saved screen area ....
                  true                                    ! O.K. exit.
             end
      end
  end
```

Figure 4.5 An Outline of the *menu* Procedure.

This procedure has used a similar technique to *string.to.tile* (Figure 4.1) of glueing together smaller images into bigger ones and then rastering these onto the screen. Then *locator* is used to monitor the user's mouse activity and when the mouse is clicked and released over one of the icons, the equivalent procedure is activated. All of this is dependent on the availability of first-class procedures, which are passed in as parameters and then executed having been selected from the vector. The way in which *menu* returns a menu producing procedure, rather than displaying the menu itself, also relies on first-class procedures. The result of *menu* is the procedure defined in the second half of the body of *menu*. This ability to define one procedure inside another, binding into values derived from parameters, is a very powerful technique which will be used often in this work.

4.1.4 Other Menu Facilities.

This menu facility can be used as a basis for more sophisticated menu operations. For instance, the author built a "variable length menu" procedure. The call to *vmenu*, as it is called, is the same as to *menu*, but the resulting procedure takes three parameters. In addition to the X and Y position, it requires a vector of booleans of the same length as the vectors of icons and actions. The effect of this parameter is

that in any call to the menu only those options whose boolean value is **true** will be available during this call. Unavailable items are not shown in this implementation, but a version could easily be provided which "greyed" the unavailable options. The code for *vmenu* differs from that for *menu* in that a lot of the image building is done inside the returned procedure, rather than beforehand. This, of course, makes the returned procedure a little slower.

Another menu constructor is the Chooser. This creates a menu interface to the choice of one member from a set of objects. It takes in a vector of strings which are identifiers for members of the set and returns a package of procedures, among which is one for presenting a menu of these identifiers. In designing the Chooser, the author took account of the following problems:

- the set of identifiers may grow large - in fact very large - and the method of selection must reflect this - not only will all the items not fit on the screen at once, but there must be some mechanism for going quickly to an item remote from those currently displayed;

- the image may have to be constructed dynamically for large sets, but need not be for small ones;

- in a typical application, the membership of the set may be changing, yet to continually keep creating the menu afresh would make it too slow;

- in PS-algol, **tables** are often use to contain sets, so one common use of the Chooser will be to select between the keys of a table.

The use of the package is illustrated in the context of a table of "widgets". To generate a Chooser, given *widgetTable* to be a pointer to this table, the following call would be made:

let *widgetChooser* = *set.up.choose*(*sort.strings*(*table.to.text*(*widgetTable*)))

which makes use of three procedures which are provided together:

- *table.to.text* - this is a procedure which takes in a table and returns a vector of all the string keys in the table;

- *sort.strings* - this is a procedure which takes a vector of strings and sorts them alphabetically (note: the string keys of the table are **not** held alphabetically);

- *set.up.choose* - this procedure takes the alphabetically ordered string keys and returns a Chooser package.

This package has the following structure:

```
structure ChooserPack(  proc( string, int, int -> string ) do.choose;
                        proc( string ) add.choose;
                        proc( string ) remove.choose;
                        proc( int, int ) list.choose )
```

and its elements do the following:

- *do.choose* - takes in a title and x- and y-positions, displays the menu and returns the selected identifier - this will be described in more detail below;

- *add.choose* - adds a new identifier to the menu;

- *remove.choose* - deletes an identifier from the menu;

- *list.choose* - lists all the identifiers in the menu using the *more* facility, described above.

The *add.choose* and *delete.choose* operators are provided to cover the case in which the program is adding and deleting members of the set. When an add or delete is performed, a call to one of these operators revises the menu much more efficiently than a fresh call to *do.choose* would do.

The operation of the resulting menu addresses the other points raised above. At any time, a subset of the options is displayed together with forward and backward scroll tiles if there are earlier or later entries and a "quit with null" tile. Initially, the first 15 entries are displayed with the "forward" tile and the"quit with null" tile. The user may now either: select one of the displayed entries; go "forward"; "quit with null"; or type a letter on the keyboard. The effect of the latter is to restrict the menu to entries beginning with that letter. More letters may be typed to restrict the selection further. Figure 4.6 shows a menu after typing "wa" - note that the typed string appears just under the title.

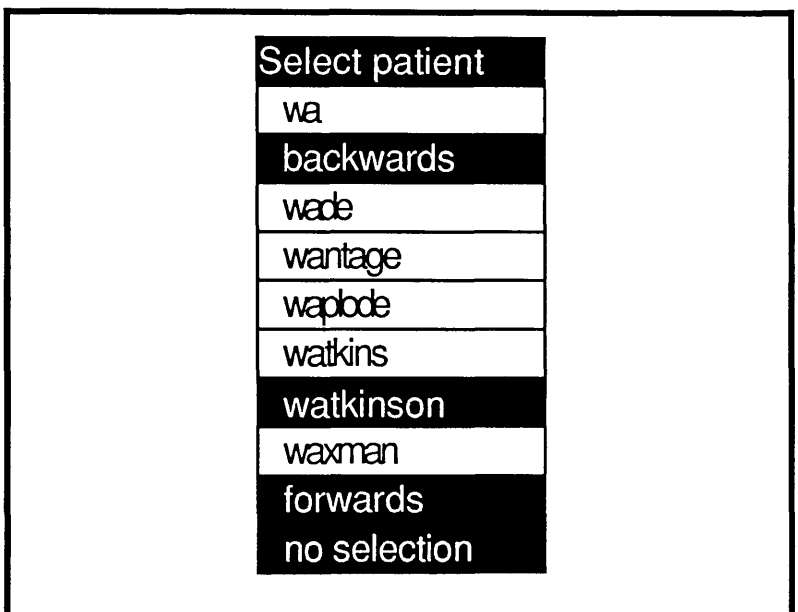


Figure 4.6 A Sample Chooser Menu.

The forward and backward tiles are available. Selecting one of these displays the 15 entries before or after the current ones as usual and has the effect of cancelling the typed string. This mechanism gives a neat balance between menu- and text-driven selection - you can select the object you want by typing enough of its name to identify it uniquely.

4.1.5 A Dialogue Box Interface.

A dialogue box is a similar mechanism to a menu, except that the layout of the active boxes, or "light buttons", is not constrained to fit a linear sequence - they can appear anywhere on the screen - and light buttons can be added or removed from the screen at any time. To support the use of such dialogue boxes, the author has written a utility which generates them. The utility manipulates light buttons (rectangular boxes with textual messages in them) associating each one with a parameterless procedure. This implements an activity which should occur when the corresponding button is selected. The functions provided include facilities to add new light buttons, to remove them and to monitor the box for mouse activity.

The calling program includes a line of the form:

```
let newForm = Form.generate( )
```

and now *newForm* is a package of the following procedures:

```
structure(  proc(string,int,int,int,int,bool,proc(),pntr->pntr) Form.add;
            proc( pntr ) Form.show;
            proc( ) Form.all.show;
            proc( pntr ) Form.remove;
            proc( string, pntr ) Form.update;
            proc( ) Form.clear;
            proc( -> pntr ) Form.mouse;
            proc( ) Fender;
            proc( ) Form.monitor  )
```

Note that this is an Abstract Data Type for dialogue boxes. Each time *formGenerate* is called, a new one is created and each dialogue box is manipulated by the operations in the structure above. The dialogue box is represented internally as a vector of light buttons, each of which has the following structure:

```
structure AREA(
    #pixel under;          ! stores what was on the screen before the
                           !  light button was displayed
    string strip;          ! contains the message displayed in the
                           !  light button
    int axo, ayo, axh, ayh; ! the origin and size of the light button
    bool redisplay;        ! if the LB should be redisplayed after its
                           !  action has been executed - i.e. if the
                           !  action affects that part of the screen
    proc() action;         ! the procedure activated by clicking over
                           !  the button
    pntr afont )           ! the font that the message is displayed in
```

The operations have the following effects:

- *Form.add* - add a dialogue box element. It takes as parameters the string to be displayed in the box, the origin and size of the box, a boolean for redisplay, a procedure and a pointer to the font it is to be displayed in - i.e. values for all the fields of the *AREA* structure, except the *under* field. It automatically calls *Form.show* to display the new element.

- *Form.show* - display a light button, given a pointer to it.

- *Form.all.show* - display all the light buttons.

- *Form.clear* - clear all the form elements from the display, by displaying their *under* images. NB this only clears the display not the vector.

- *Form.remove* - remove an element from the form, given a pointer to it. This removes the light button both from the display and from the vector.

- *Form.update* - update the text associated with a light button, given the new string and a pointer to it.

- *Form.mouse* - return a pointer to the selected element.

- *Form.monitor* - wait until a light button is selected and then execute the procedure associated with it. Continue until a mouse button associated with *Fender* is called.

- *Fender* - This procedure must be associated with one of the light buttons to terminate the call of *Form.monitor*.

There is also a procedure:

- *Form.null* - which acts a bit like *form.add*, taking all the same parameters except for the procedure and the boolean. It is used for parts of a form which are not light buttons.

The most common method of working with the Form Package consists of four steps: the form is generated by a call to *Form.generate*; for each light button required, a series of calls to the *Form.add* procedure is made; a final call to *Form.add* is made, associating *fender* with a button labelled "quit", say; *Form.monitor* is called to provide the functionality of the dialogue box.

4.1.6 A Simple String Editor.

Any application will require the ability to edit the objects which it manipulates, perhaps in the form of a set of syntax-directed editors for those objects, such as that described in [Blott and Campin, 1987]. Alternatively an object can be transformed into a textual form and be edited with a text editor, such as that produced by Douglas MacFarlane and described in [Cooper *et al.*, 1987a]. Here a simpler tool is described, which could be amplified to form the basis of either of the tools. It permits the editing of a single string, in a box on the screen which looks as shown in Figure 4.7.

This window contains all or part of the string being edited, with a title above it in a smaller font and two small scroll bars to either side. Above the string, a cursor position is indicated by a vertical bar. The principal purpose of providing such a tool is to make it easy for many programs to present an identical mechanism for users to supply and adjust text parameters. As procedures are values stored in a persistent context, this version could be replaced later, automatically changing the way text is

edited for all the programs using this, without further re-compilation or re-loading. Therefore, a supplied set of software could then be tailored to the customer's needs.

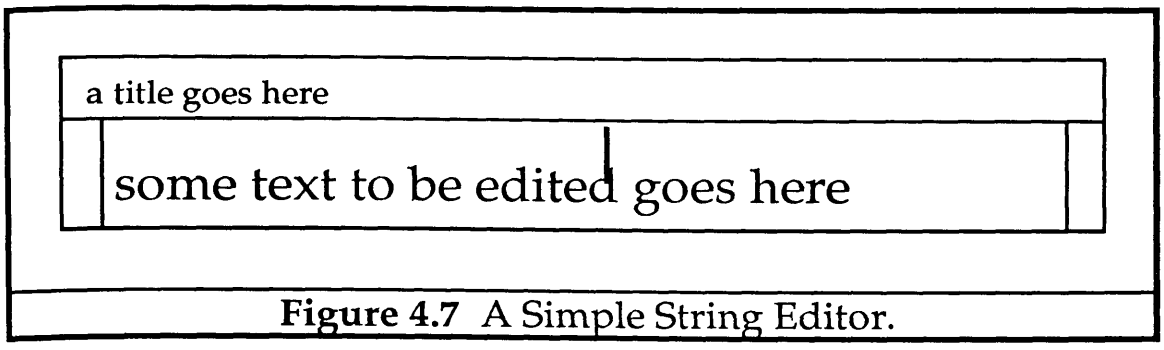


Figure 4.7 A Simple String Editor.

The editor is called by a line of the form:

let *newString* = *seditor*(*title, oldString, xo, yo, xh, yh*)

where the parameters are two strings containing a title for the editing operation and the string to be edited, and four integers for the origin and size of the editing box. The edited string is returned as the result of the procedure.

The editor takes mixed mouse and keyboard input and functions as follows:

- the del key erases one character to the left of the cursor;
- the oops key erases all the characters to the left of the cursor;
- the return key quits, returning the string to the left of the cursor;
- the line-feed key quits, returning the whole string;
- printing characters are inserted at the current cursor point;
- selecting the text area moves the cursor to that point in the string;
- selecting the right-hand scroll bar moves text that is off the screen to the right into the text window so that the character to the right of the cursor is now the leftmost of the window;
- selecting the left-hand scroll bar similarly moves text to the left of the window into the window so that the character to the left of the cursor is now the rightmost in the window.

The editor is implemented by a PS-algol procedure. It uses a slightly more complex form of the box creation - this time the box is divided into two sections, an upper one containing the title and a lower one with the text to be edited. The procedure makes use of two inner procedures: *CursorDisplay* which returns the pixel position of the cursor; and *showText*, which clears the text display box and displays the current text and the cursor. The text state is maintained in variables which contain the

text to the left and right of the cursor and the whole text. The cursor position is also kept.

The main part of the procedure starts by displaying the text with the cursor at its right hand end. It then circles round a loop, getting input from the mixed input procedure and reacting as follows:

if it is a mouse-click over the text window, the cursor is moved and then a new cursor position is calculated, before the whole text and cursor are re-displayed;

if return is pressed, the exit condition is set;

if the "oops" key is pressed, the left hand text is erased by a call to *showText.*;

if the "delete" key is pressed, one character is stripped off the left hand text;

any other key press is added to the left hand text.

This procedure illustrates further how little code is required to provide reasonably powerful facilities. This simple editor has proved a reasonable input method for short strings.

4.1.7 Summary of User Interface Tools.

A set of tools of increasing complexity has been developed using the bitmap graphics facilities. The methods of manipulating these objects have clear similarities to the more usual arithmetical and textual operations, and therefore soon become familiar and easy to use. Indeed, the coherence of the numerical, textual and graphical facilities makes the modelling of complex objects with components drawn from all three of these domains particularly straightforward. It is therefore simple, for instance, to create and manipulate light button objects which are a direct representation of the functionality of a light button.

Furthermore, the ability of the ordinary user to write procedures which directly manipulate the user interface gives increased freedom in designing the interface. Most systems provide libraries which are cast in stone and which the user must use for the interface. If the operation of the components of the library is slightly different from what is required, then nothing can be done about it. In PS-algol, a fresh procedure can be written and used instead of the standard one.

Moreover, the Chooser illustrates a further bonus. It permits the choice between a set of objects to be varied dynamically. If the membership of the set changes, this will immediately be reflected in the choice available. That is, if a new object is inserted into a table with the choice being made via the Chooser, then the next time the Chooser is invoked, the new object will be available for selection.

Therefore, the work reported in this section can be summarised in two statements. Firstly, the graphics facilities enable the manipulation of graphical objects in a way that feels the same as manipulating numbers and text. Secondly, the freedom to create user interface procedures liberates the interface design from control by externally produced software.

4.2 A Database Browser.

For application debugging purposes, there is a requirement for a general purpose mechanism for browsing the persistent store, in order that the effects of programs can be verified. Using the browser, it becomes possible to check that the structure and contents of the database are consistent with the design of the application. At first sight, the strong typing of PS-algol would seem to be an insuperable barrier against creating such a tool. However, Dearle and Brown have created a browser which can navigate around the persistent store without violating the security of the type system [Dearle and Brown, 1988]. The only restriction is that there is no way to enter a procedure closure. Otherwise the browser lets the user traverse the values in the database, navigating by following pointer chains and lists of vector elements. As this introduces a technique which will be exploited repeatedly, the browser will now be described in some detail.

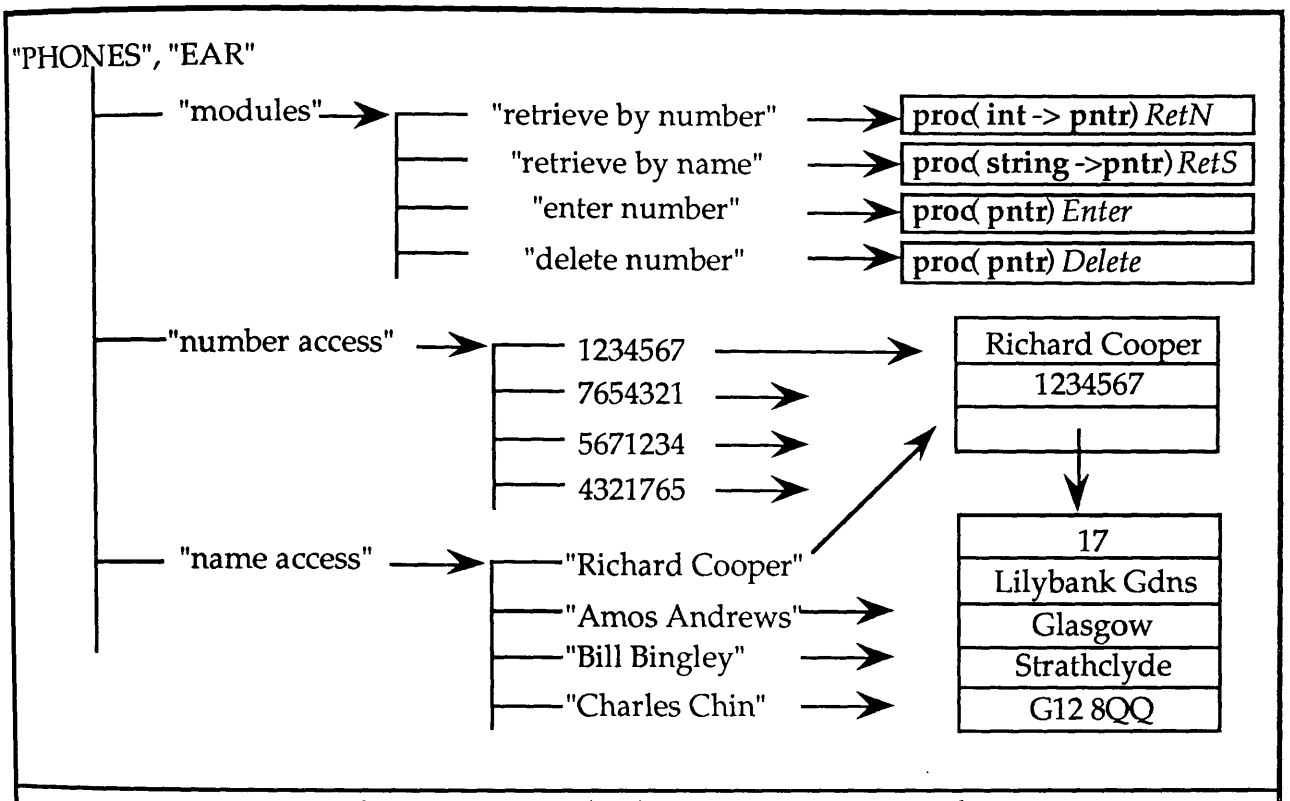


Figure 4.8 A Telephone Directory Database.

4.2.1 A Functional Description of the Browser.

The functionality of the browser will now be described in an idealised form. At present little consideration has been given to the user interface, which is not quite as easy to use as the one described. In summary, the user summons the browser and then selects a database to browse. The browser then provides a menu of the keys of the items in the top-level table of the database. The user selects one of these keys and now the associated object becomes the focus of attention, for which a further menu is provided. If the object is another table, this allows selection of one of the objects in the table. If it is not a table, the menu shows the names of the fields of the object. Selecting one of these names: displays the field value if it is a scalar; displays a menu to select an element if the field is a vector; or, if the field is another complex object, this becomes the focus of attention and another menu is displayed in the same way. All

menus have an entry to return to the previous menu. If the object is a procedure, the error message "Cannot Traverse a Procedure" is displayed.

To illustrate the facility further, consider the database shown in Figure 4.8. This is a PS-algol database (name "PHONES", password "EAR") which supports a telephone directory. The top level table contains just three entries: a table of the modules comprising the software which implements the access methods to the phone directory; and two tables which provide different access paths to the directory itself (one via the number, the other by the name). Each entry in the module table is a packaged procedure, while each entry in the directory tables points to a structure of the following kind:

structure *phoneEntry*(**string** *Pname*; **int** *Pnumber*; **pntr** *Paddress*)

where the *address* field points to a structure of the form:

structure *address*(**int** *Hnumber*; **string** *street, city, county, postcode*)

To browse such a database, the browser is called and given the database name and password. Then menu A from Figure 4.9 appears.

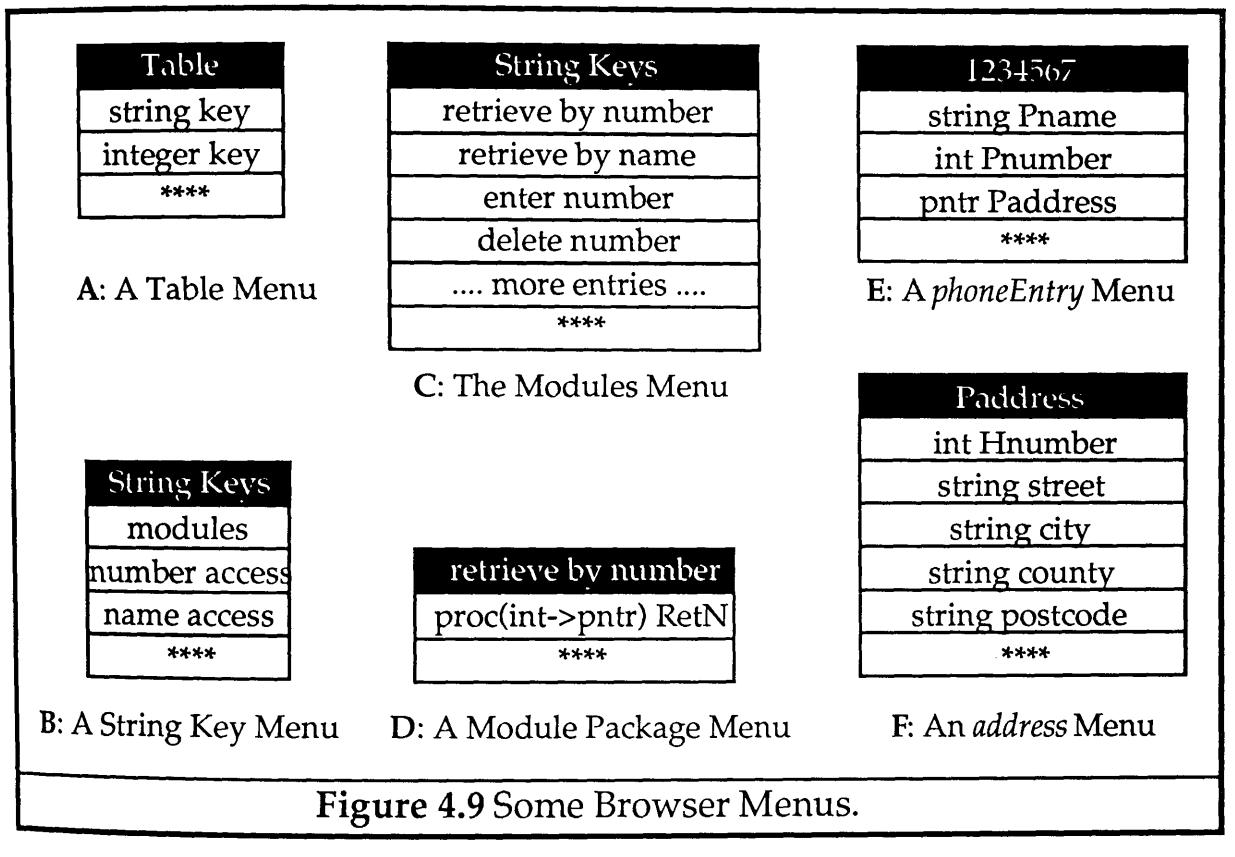


Figure 4.9 Some Browser Menus.

This menu reflects the fact that the object being browsed is a table, the top-level table. The three options have the following effect:

string key: if selected this will provide a menu of all the string keys in the table - selecting one will mean traverse to the object keyed by that string;

integer key: equivalent to the above, except the menu is of the integer keys in the table;

****: this appears in all menus and means quit this level of browsing and return to the previous level - in this case quit the browser all together.

Therefore, in this case selecting "****" will quit the browser, selecting "integer key" will display the message "no integer keys to select from" and selecting string key will provide menu B from Figure 4.9. From Menu B, selecting "****", as might be expected returns to menu A. Selecting each of the others results in other "table" menus, similar to the previous one, except with a different heading.

Assuming "modules" is selected, the three options of the resulting table menu now have the following effects: "****" returns to menu B; "integer key" displays the message "no integer keys to select from"; while "string key" displays a menu of all the modules. Suppose "string key" is selected, the menu of keys from the procedure table appears (menu C in Figure 4.9). Selecting "retrieve by number", a menu of two entries is given (menu D). Selecting "proc() retrieveProc" results in the message "Cannot Traverse Procedures" - the limitation of this technique has been reached. It is not possible to look inside procedure closures.

If another branch of the database is traversed instead - say "number access", a table menu now appears whose options result in: "****" - back to the top-level table menu; "string key" - message "no string keys to select from"; but "integer key" now provides a Chooser menu of numbers. When a number is selected, menu E from Figure 4.9 is shown.

The four options have the following effects:

string Pname: this displays the message "String field Pname has value "Richard Cooper" ";

int Pnumber: this displays the message "Integer field Pnumber has value "1234567" ";

pntr Paddress: this provides a fresh menu (Menu F) to access the address object;

****: this returns to the menu of phone number keys.

Selecting a **pntr** field corresponds to traversing to a component object of the current object - in this case moving from a *PhoneEntry* object to an *address* object - and results in a fresh menu (menu F), this time with six entries - one for each of the five fields of the structure (selecting any of these displays the value of the field), and one for the "****", which quits back to the *PhoneEntry* menu.

Vector objects have a special menu of their own. This consists of three entries:

index on: provide an integer and traverse to the object with that index in the vector;

show all: traverse each object in the vector in turn;

****: quit and return to previous menu, as usual.

Using this technique of following pointer references, it is possible to browse all the objects in the persistent store except, as has been mentioned, when they lie within a procedure closure. The next section describes how it has been possible to write such a program, without breaking the type security of the object store. The browser is written in PS-algol and is not a C-program which interprets the objects as bit-strings. All objects are only used within the restrictions of their type.

4.2.2 The Implementation of the Browser.

The strategy used for implementing the Browser is well described in [Dearle and Brown, 1988] and this section is a summary of that paper, with slight amendments to the user interface.

Taking the elements of the program in the same order as Dearle and Brown, the procedure in Figure 4.10 will produce the *phoneEntry* menu.

```
let traversePhoneEntry = proc( pntr p )
   begin
      structure phoneEntry( string Pname; int Pnumber; pntr Paddress )
      let return := false
      let entries = @ 1 of string [ "string Pname",
                                    " int Pnumber",
                                    "pntr  Paddress",
                                    "****"  ]
      let procs = @ 1 of proc() [
                        proc(); message( "Pname has value" ++ p( Pname) ),
                        proc(); message( "Pnumber has value" ++ p( Pnumber ) ),
                        proc(); traverse( p( Paddress ) ),
                        proc(); return := true  ]
      let thisMenu = menu( "phoneEntry", entries, procs )
      while ~return do thisMenu()
   end
```

Figure 4.10 A Structure Traverser.

This example uses a special version of *message*, which displays a string in a box in the centre of the screen, and a special version of *menu*, which now takes a string title, string entries and parameterless action procedures and executes without returning any value. The part of this procedure which provides a problem is the call to the procedure *traverse*. This is being passed an *address* object and must handle it - how can it do so?

One technique consists of creating a table of *traverse...* procedures, contained in structures with just one field of type, **proc(pntr)**, and keyed by some information which identifies which class they operate on. PS-algol provides a standard function which can be used for this purpose. The clause

let *phoneEntryClass* = *class.identifier(p)*

(where *p* points to a member of the class *phoneEntry*) returns all the class information as a string. Figure 4.11 shows how this is used to traverse any structure.

```
let traverse = proc( pntr p )
  begin
    structure traversePack( proc( pntr ) traverser )
    let class = class.identifier( p )              ! Find the class.
    let look = s.lookup( class, traverseTable )    ! Find the procedure for this
    if look is traversePack
                then look( traverser )( p )        ! Unpack and apply the procedure.
                else error()                       ! Somebody forgot to put it in.
  end
```

Figure 4.11 A First General Purpose Traversal Procedure.

The procedure works by trying to find an appropriate menu-producing facility for an object of *p*'s class and then using that traverser or reporting an error. In order to use such a traversal mechanism, programmers would have to remember to write traversal procedures for each class of object they wish to store - an unacceptable programming overhead. It would be better to populate the table automatically.

This can be achieved by making use of the compiler function described in 3.3.10. The extended form of traverse, which constitutes the Browser, replaces the call to *error* by code to create, store and use an appropriate traverser. This builds a traversal procedure for the object class, then compiles it, stores it in the table for re-use and then calls it. Figure 4.12 shows a revised version of Figure 4.11, this time calling a general purpose traverser maker, if one does not already exist.

```
let traverse = proc( pntr p )
  begin
    structure traversePack( proc( pntr ) traverser )
    let class = class.identifier( p )              ! Find the class.
    let look = s.lookup( class, traverseTable )    ! Find the procedure for this
    if look = nil do
      begin                                        ! Traverser not found.
        look := makeTravProc( class )              ! Make a new procedure.
        s.enter( class, traverseTable, look )      ! Store the new procedure.
      end
    look( traverser )( p )                         ! Unpack and apply the procedure.
  end
```

Figure 4.12 A Second General Purpose Traversal Procedure.

As this piece of code is extremely complex and also provides a template for many examples in the following chapters, it will be defined incrementally. The first attempt at producing a procedure which, given the class identifier of an object, will produce a traversing procedure is shown in Figure 4.13.

This traverser-making procedure builds up a procedure of the form shown in Figure 4.10 in the string variable, *program*, and then compiles it. The method for automatically building a procedure is illustrated here for the first time and requires amplification. A template for the procedure is created as a string, held in *program* in this case. The template consists of mixtures of code and dummy names for information dependent on the class structure.

```
let makeTravProc = proc( string class - > pntr )
  begin
    let program :=                                  ! Program Template as per Figure 4.10
"           proc( pntr p )
              begin
                #STRUCTURE                          ! Dummy class structure.
                let return = false
                let entries = @ 1 of string [ #ENTRYVEC TOR) ]      ! Dummy entries.
                let procs = @ 1 of proc() [ #PROCVECTOR]            ! Dummy procedures.
                let thisMenu = menu( #CLASSNAME, entries, procs) ! Dummy class name.
                while return do thisMenu()
              end
"

    let className = getClassName( class )           ! String handling procedures to
    let classStruc = getClassStruc( class )         !   information from the class
    let cFieldTypes = getClassTypes( class )        !   structure.
    let cFieldNames = getClassFieldNames( class )

    replace( program, #STRUCTURE, classStruc )          ! Insert structure.
    for i = 1 to upb( cFieldTypes ) do                  ! Insert Entries.
        replaceVector ( program,
                #ENTRYVECTOR,cFieldTypes( i ) ++ " " ++cFieldTypes( i ) )
    replaceVector ( program, #ENTRYVECTOR, "****" )
    endVector( program, #ENTRYVECTOR )
    for i = 1 to upb( cFieldTypes ) do if cFieldTypes( i ) = "pntr"   ! Insert procedures.
        then replaceVector( program, #PROCVECTOR,
                    proc(); traverse( p(" ++ cFieldNames( i ) ++ ") )" )
        else replaceVector( program, #PROCVECTOR,
                    proc(); message( "" ++ cFieldNames( i ) ++ " has value'"
                            ++ p(" ++ cFieldNames( i ) ++ ") )"
    replaceVector ( program, #PROCVECTOR, "proc(); return := true" )
    endVector( program, #PROCVECTOR )
    replace( program, #CLASSNAME, className)        ! Insert class name.

    structure traversePack( proc( pntr ) traverser )
    let dummy = traversePack( proc( pntr p ); nullproc )
    compile( program, dummy )
end
```

Figure 4.13 A First Traverser Maker.

In this case, the template contains placeholders for the class structure; the entries in the menu (being field name, field type pairs); the action procedures (either calls to message for scalar values, or further calls to *traverse* for **pntr** values); and the class name. These placeholders are then filled by use of the following three procedures:

- *replace* - this takes a place holder and makes a simple replacement by a string provided as a parameter;
- *replaceVector* - this takes a place holder for a vector and inserts a string followed by a comma before it;

and

- *endVector* - takes a place holder and removes it (and a preceding comma) to tidy up a vector definition.

The text of the code using these procedures has been simplified by assuming four string handling procedures which take the class identifier and return the following: the structure name; a PS-algol structure definition; a vector of field type names; and a vector of field names. The problem however is that *program* will not compile as it stands, because it has no referend for *traverse* any more than it had one in Figure 4.10. In order to make a reference, *traverse* will have to be passed in as a parameter as Figure 4.14.

```
let makeTravProc = proc( string class - > pntr )
  begin
     let program :=                                  ! Program Template as per Figure 4.10
"          proc( proc( pntr pntr ) doTraverse -> proc( pntr ) )          !***
            proc( pntr p )                                               !***
              begin
                 #STRUCTURE                          ! Dummy class structure.
                 let return = false
                 let entries = @ 1 of string [ #ENTRYVEC TOR) ]     ! Dummy entries.
                 let procs = @ 1 of proc() [ #PROCVECTOR]           ! Dummy procedures.
                 let thisMenu = menu( #CLASSNAME, entries, procs) ! Dummy class name.
                 while return do thisMenu()
              end
"

     let className = getClassName( class )          ! String handling procedures to
     let classStruc = getClassStruc( class )        !   information from the class
     let cFieldTypes = getClassTypes( class )       !   structure.
     let  cFieldNames = getClassFieldNames( class )

     replace( program, #STRUCTURE, classStruc )          ! Insert structure.
     for i = 1 to upb( cFieldTypes ) do                  ! Insert Entries.
          replaceVector ( program,
                    #ENTRYVECTOR,cFieldTypes( i ) ++ " " ++cFieldTypes( i ) )
     replaceVector ( program, #ENTRYVECTOR, "****" )
     endVector( program, #ENTRYVECTOR )
     for i = 1 to upb( cFieldTypes ) do if cFieldTypes( i ) = "pntr"   ! Insert procedures.
          then replaceVector( program, #PROCVECTOR,
                         proc(); doTraverse( p(" ++ cFieldNames( i ) ++ ") )" )     !***
          else replaceVector( program, #PROCVECTOR,
                         proc(); message( "" ++ cFieldNames( i ) ++ " has value'"
                                   ++ p(" ++ cFieldNames( i ) ++ ") )"
     replaceVector ( program, #PROCVECTOR, "proc(); return := true" )
     endVector( program, #PROCVECTOR )
     replace( program, #CLASSNAME, className)       ! Insert class name.

     structure genTraversePack( proc (proc( pntr ) ->proc( pntr ) ) genTraverser )   !***
     let dummy = genTraversePack( proc (proc( pntr ) D-> proc( pntr ) ); nullproc )  !***
     let genTraverser  = compile( program, dummy )                                   !***
     genTraverser( traverser )                                                       !***
end
```

Figure 4.14 A Second Traverser Maker.

This procedure now has been complicated by an extra level of indirection (the changes from the previous version are in lines ending with !***). *program* now contains, not a procedure which traverses a structure, but a procedure which generates such a procedure given the general traverser as input. The changes necessary to accomplish this are as follows:

- replace the simple procedure signature with a signature which indicates that the procedure expects *traverse* as a parameter;
- make the procedure return a procedure by placing a signature for this resulting procedure as the second line of *program;*
- in the action for **pntr** types, call the input parameter, *doTraverse* , instead of calling *traverse*
- adjust the structures associated with the call of the compiler;
- call the compiled procedure to bind *traverse* into the procedure.

This general traverser can now be written as in Figure 4.12. There is nothing left to do except to add some details to deal with vectors and tables in the ways described above. The core of the browser is in the last two procedures given.

This experiment shows how a relatively small amount of code has provided a facility of high order. The availability of first-class procedures and, in particular, of a compiler which is a first class object has allowed the resolution of two apparently incompatible goals. Nowhere in this code is the type system violated and yet the program is able to adapt in a polymorphic way to new classes of data.

4.3 Operations on Whole Objects.

Section 3.2.3 introduced structures and the **pntr** type, showing that the copy and equality testing operations have reference semantics. That is, two variables of type pntr are the same if they refer to the same objects, while making an assignment of one pntr variable to another makes them both point to the same object. Thus, following the fragment:

```
let myAddress = address( 17, "Lilybank Gdns", "Glasgow" )
let yourAddress =  myAddress
```

any change to any of the fields of *myAddress* also affects *yourAddress*. Similarly, after:

```
let hisAddress = address( 17, "Lilybank Gdns", "Glasgow" )
```

the test *myAddress* = *hisAddress* returns **false** as there is only comparison of pointers, not of contents. Compare this with FAD [Bancilhon *et al.*, 1987], which has three notions of equality:

- **identity equality** - as described above;
- **shallow equality** - the fields of the object are the same;

and
- **deep equality** - two objects are the same if they have the same structure and values down to arbitrary levels of pointer chains.

Given the following:

```
structure outer( string one; pntr into )
structure inner( integer two)
let I1 = inner( 1 )
let O1 = outer( "A", I1 )
let O2 = O1
let O3 = outer( "A", I1 )
let O4 = outer( "A", inner( 1 ) )
let O5 = outer( "B", inner( 2 ) )
```

then, the following hold:

- *O1* and *O2* have identity, shallow and deep equality;
- *O1* and *O3* have shallow and deep equality;
- *O1* and *O4* have deep equality only;

and
- *O1* and *O5* are not equal in any sense.

That is, deep equality implies shallow equality, which in turn implies identity equality. Deep versions of copy and equality (and display as well) which use the whole of the object are sometimes wanted. These will enable general purpose print operations, value based equality tests for objects and the ability to take a fresh copy of an object for modification, for instance. In this section, the method of building operations on whole objects on top of the **pntr** type using the run-time compiler will be described.

Some care is required over cyclical pointer references. Consider the following example:

```
let A := outer( "X", nil )
A( into ) := A
let B := outer( "X", nil )
B( into ) := B
```

In testing *A* = *B*, there are two problems. Firstly, the program must not circle endlessly. To avoid this, a check is kept of all objects encountered so far, so that objects are not decomposed more than once. Secondly, the semantics must be correct. *A* and *B* are different objects and their *into* fields point to different objects, so they certainly do not exhibit identity or shallow equality. With regard to deep equality, there is a choice of interpretation. The two objects have the same structure and the same "base" data value (the "X"), so they seem to be the same in the deep equality sense. What, however, should the test make of the following example?

```
let C := outer( "X", nil )
let D := outer( "X", C )
C( into ) := D
```

Now *C* and *D* point to a two-element cycle of objects and therefore they are not structurally the same as *A* and *B* . They should not have deep equality. Figure 4.15 shows a structure diagram for these objects.

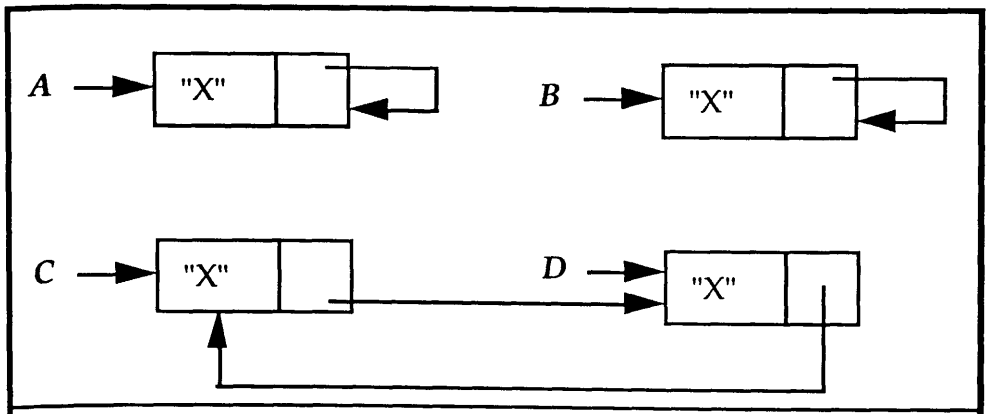


Figure 4.15 Some Cyclical Structures.

Using reference semantics, however, it is very difficult to distinguish these two cases. The deep equality test here essentially consists of: the structures are the same; the scalar field is the same; the **pntr** field points to an object of the same structure; therefore they must be the same. The implementation choice is whether or not to follow cyclical pointer chains. If they are followed, then *A*, *B*, *C* and *D* are all the same. If not, then none of them are the same. In this work, the former choice is taken arbitrarily, since either is possible.

4.3.1 Deep Print Operations.

A procedure is required which, when given a pointer to a *phoneEntry* object, for instance, will print something like

```
The structure name is: phoneEntry
   Pname: "Richard Cooper"
  Pnumber: 0413398855
  Paddress:
     The structure name is: address
      Hnumber: 17
      street: "Lilybank Gardens"
      city: Glasgow
      county: Strathclyde
      postcode: G12 9QQ
```

A specific printer for *phoneEntry* objects is given as Figure 4.16.

```
let printPhoneEntry = proc( pntr X ; string IND)
  begin
     structure phoneEntry( string Pname; int Pnumber; pntr Paddress )
     print IND, "The structure name is: phoneEntry"
     print IND, "    Pname: '" ",X( Pname )," "'"
     print IND, "   Pnumber: ",X( Pnumber )
     print IND, "   Paddress: "
     printAddress( X( Paddress ), IND ++ "    " )
  end
```

Figure 4.16 Printing a Phone Entry.

This is a fairly straightforward piece of code - the *IND* string is the current level of indentation, increased every time a sub-object is printed.

To generalise this, the same technique is used as in the browser - the creation of a general printer procedure which takes in an object and an indentation. The general printer then calls specific printers from this table, creating them as required. The specific printers take in the object, the table and the indentation and also a copy of the general printer. The procedure is given in Figure 4.17 and shows a marked similarity to the general purpose traverser in Figure 4.12.

```
let deepPrint = proc( pntr X; string IND )
  if X = nil then print IND," NIL"
  begin
     structure PrinterPack( proc( pntr, pntr, string, proc(pntr, pntr, string ) ) Printer )
     let class = class.identifier( X )
     let packedPrinter := s.lookup( class, printerTable )
     if packedPrinter = nil do
          begin
               packedPrinter := makePrintProc( class )
               s.enter( class, printerTable ,packedPrinter )
          end
     packedPrinter( Printer )( X, IND, deepPrint )
  end
```

Figure 4.17 A General Purpose Deep Print Procedure.

For this procedure to function properly, the automatically generated printer procedures need to have recursive calls to *deepPrint*. A reference to *deepPrint* is therefore passed in as a parameter to the printer, as shown in Figure 4.18.

```
proc( pntr X; string II; proc( pntr,pntr,string ) DP )
  begin
     structure phoneEntry( string Pname; int Pnumber; pntr Paddress )
     print IND, "The structure name is: phoneEntry"
     print IND, "   Pname: '" ",X( Pname )," "'"
     print IND, "   Pnumber: ",X( Pnumber )
     print IND, "   Paddress: "
     DP( X( Paddress ), IND ++ "   " )
  end
```

Figure 4.18 An Automatically Generated Print Procedure.

Now a *makePrintProc* procedure to generate such procedures is written. It is very similar to traverser maker shown in Figure 4.14 and consists of the following steps:

i) Provide a template for the procedure. Essentially it looks like the Figure 4.18, except that the underlined structure-specific information is replaced by place holders - two scalar ones for the structure and structure name and one vector of statements for dealing with each of the fields.

ii) Decompose the class structure into its fields.

iii) Replace the simple place holders for the structure and the structure name.

iv) For each scalar field add a line of the form:

print *IND*, " fieldname:", *X*(*fieldname*).

and for each **pntr** field, add two lines of the form:

print *IND*, " fieldname:"
DP(*X*(*fieldname*), *IND* ++ " ").

v) Compile the procedure and return it.

The procedure is still deficient in two respects: it does not handle cyclical structures; and it does not handle the whole type system, notably constants and vectors. The first deficiency is corrected by making some additions to *deepPrint* so that it maintains a list of the objects it has already printed. After checking from **nil** structures, *deepPrint* checks to see if the object is already on this list. The procedure numbers all objects, so that it can print references to previously displayed objects.

The second deficiency is more difficult to rectify and requires the parsing of the type description of each field. The procedure, *cFieldTypes*, now no longer returns a vector of strings, but a vector of lists of parsed types. The structure of a list element is:

structure *typeList*(**string** *S*, *R*; **pntr** *N*)

where *S* contains a parsed atom, *R* contains the rest of the type descriptor (used only in the deep copy procedure) and *N* a pointer to the parsing of *R*. For instance the type ***c*int** would result in a list of three elements:

E1 = *typeList*("*", "*int", *E2*)
E2 = *typeList*("c", "int", *E3*)
E3 = *typeList*("*", "int", *E4*)
E4 = *typeList*("int", "", **nil**)

Using this structure, the loop in *makePrinter* can be expanded to handle any type. This is illustrated for the output of a vector field with the structure:

structure *array*(***c*int** *element*)

for which a print procedure is given as Figure 4.19.

```
proc( pntr X; string IND; proc( pntr,pntr,string ) DP )
  begin
     structure array( *c*int element )
     print IND, "The structure name is: array"
     print IND, "   element: ***"
     for i = lwb( X ( element ) ) to upb( X( element ) ) do          ! Outer loop.
       begin
          print IND, i:5, "    : ***"
          for ii = lwb( X ( element )( i ) ) to upb( X( element )( i ) ) do   ! Inner Loop.
            begin
               print IND, i:5, " , ", ii:5, X( element )( i) ( ii )    ! Process Element.
            end
       end
  end
```

Figure 4.19 A Print Procedure for a Vector Field.

Step (iv) of the description of the printer maker given above is replaced with a more general purpose description.

Given the following:

FT - the type description of the object;

CIV - the current vector index variable - initially *i* with another *i* added at each recursion;

COB - the name of the current object - initially " XX(field name)" but with "(" ++ CIV ++ ") added at each vector recursion;

INDXS - the string used to print the indexes - initially "i:5, " then "i:5, ii:5, ", etc. - note the ":5" merely means use 5 character positions.

a recursive procedure, *oneField*, is provided which returns the string which will handle a single field. It proceeds as follows:

i) If the next part of the type descriptor is a "c" (meaning constant field), ignore it and recursively call *oneField*, with the rest of the type descriptor;

ii) If it is a scalar, return the current object, surrounded by quotes for strings, angle brackets for booleans or nothing for numbers.

iii) If it is a **pntr** field, return a call to deep print on the field value, as before.

iv) If it is a vector, embed a recursive call to *oneField* (changing the values of *CIV*, *COB* and *INDXS* appropriately) in a block which indexes over the whole of a vector object, using *CIV* as the index variable and *COB* as the object whose bounds are used to delimit the scan.

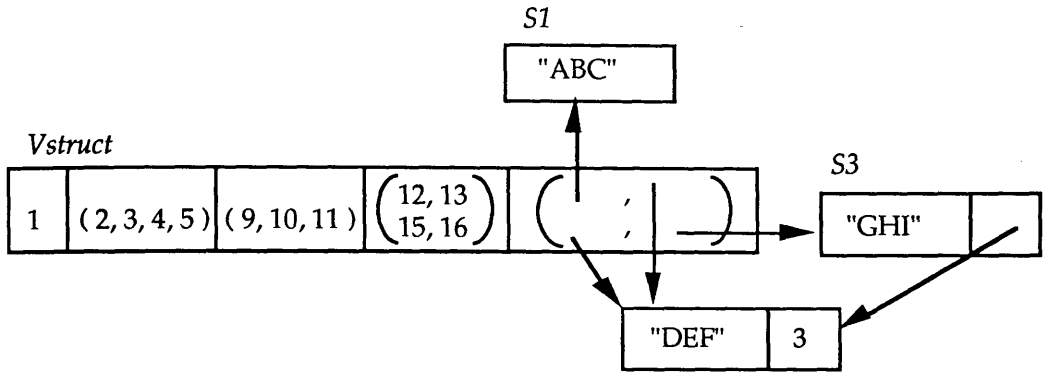
```
let oneField := proc( pntr FT; string  CIV, COB, INDXS ); nullproc
oneField := proc( pntr FT; string CIV, COB, INDXS )
    case FT( S) of
      "c":              oneField( FT( N ),CIV, COB, INDXS )
      "string":         ": "#Q #Q," ++ COB ++ " , #Q "#Q#Q"
      "bool":           ": <#Q," ++ COB ++ " ,  #Q>#Q"
      "int", "real":    ": #Q, " ++ COB ++ ""

      "pntr":           ": #Q'n      DP( " ++ COB ++ ", TT, II ++ #Q   #Q )"
      "*":
        begin
          ": ***#Q
          for " ++ CIV ++ " = lwb( " ++  COB ++ " ) to upb( " ++ COB ++ " ) do
              begin
                  print II, " ++ INDXS ++ "#Q"
          oneField( FT( N ), CIV ++ "i", COB ++ "(" ++ CIV ++ ")",
                                          INDXS  ++ "#Q, #Q, " ++ CIV ++ "i:5, " ) ++
           "    end'n"
        end
      default: {}
oneField(cFieldTypes( i ), "", "i", " X(" ++ cFieldNames( i ) ++ ")", "i:5, "  )
```

Figure 4.20 The Core of the Printer Maker.

Figure 4.20 shows this recursive procedure, followed by a typical top-level call to it. Note there is one slight syntactical simplification in this figure. The program needs to place quotation marks into the program. In PS-algol this means putting '" into the strings which make up the procedure and this makes the text unreadable. These escaped quotation marks have been replaced by the string "#Q" which is automatically translated back into '" before compilation.

Finally, the whole of the *deepPrint* program was produced, with one added sophistication to be noted. The procedure also takes in a table to hold the printers. Therefore the calling application manages the printers in its own space, for two reasons. If the printers for all applications were held together this would be a large set to scan and therefore retrieval would be slow. More critically, if there is a single table for all applications, this must be held in a table opened in write-mode. Only one application could therefore function at a time, and this would be unacceptable. The section is concluded with the output generated by the fragment given as Figure 4.21.

Structure Diagram



PS-algol Description

```
structure Vstruct( int NX; *cint NCN; c*int CNN; **int NNN; *c*cpntr PPP )
structure S1( string F1 )
structure S2( string F2; int F3 )
structure S3( string F4; pntr F5 )
let X1 := S1( "ABC" )
let X2 = S2( "DEF", 3 )
let X3 = S3( "GHI", X2 )
let VV = Vstruct( 1,
                  @ 3 of cint [ 2,3,4,5 ],
                  @ 77 of int [ 9,10,11 ],
                  @ 5 of *int [     @ 17 of int [ 12,13],
                                    @ 27 of int [ 15,16] ],
                  @ 3 of c*cpntr [  @ 47 of cpntr [ X1, X2],
                                    @ 57 of cpntr [ X2, X3] ] )
DeepPrint(VV, "")
```

Figure 4.21 A Structure Description for Printing.

The output is given as Figure 4.22

```
Object number  1
The structure name is: Vstruct
NCN:  ***
  3 :       2
  4 :       3
  5 :       4
  6 :       5
 CNN: ***
 77 :       9
 78 :       10
 79 :       11
 NNN: ***
  5  :***
  5 ,  17 :       12
  5 ,  18 :       13
  6  :***
  6 ,  27 :       15
  6 ,  28 :       16
PPP:  ***
  3  :***
   3 ,  47 :   Object number  2
       The structure name is: S1
              F1:  "ABC"

   3 ,  48 :   Object number  3
       The structure name is: S2
              F2:  "DEF"
              F3:        3

  4  :***
   4 ,  57 :  ALREADY OBJECT  3
   4 ,  58 :   Object number  4
       The structure name is: S3
              F4:  "GHI"
              F5:  ALREADY OBJECT  3

NX:        1
```

Figure 4.22 The Print Out of the Structure.

4.3.2 Deep Equality Testing.

```
proc( pntr X, Y,; proc( pntr, pntr -> bool ) EQ -> bool )
  begin
    structure phoneEntry( string Pname; int Pnumber; pntr Paddress )
    let result := true
    result := result and X( Pname ) = Y( Pname )
    result := result and X( Pnumber ) = Y( Pnumber )
    result := result and EQ( X( Paddress ), Y( Paddress ) )
    result
  end
```

a) Equality Test for the *phoneEntry* structure.

```
proc( pntr X, Y; proc( pntr,pntr-> bool ) EQ -> bool )
  begin
    structure array( *c*int element )
    let result := true
    for i = lwb( X ( element ) ) to upb( X( element ) ) do
      begin
        for ii = lwb( X ( element )( i ) ) to upb( X( element )( i ) ) do
          begin
            result := result and
                  X( element )( i) ( ii ) = Y( element )( i) ( ii )
          end
      end
    result
  end
```

b) Equality Test for the Vector Structure.

Figure 4.23 Two Examples of Equality Test Procedures.

The deep equality procedure builds procedures appropriate to the required structure of which two examples are given in Figure 4.23. The general method of

these procedures is to "**and**" together the equality tests on each field. For scalar fields these tests are straightforward. For vector fields, they require a scan of all the elements. For **pntr** references, recursive calls to the equality procedure are made.

The equality procedure is in all respects the same as the traverser in Figure 4.12 and the deep printer in Figure 4.17. It finds the structure of the objects (checking that they are the same). Then a specific equality test procedure of the type shown in Figure 4.23 is either retrieved from a table, or generated by the equality test maker. This in turn is similar to the one described in the previous section for making printer procedures. Care has to be taken when following cyclical pointer chains.

4.3.3 Deep Copy Operations.

The final member of the set of three similar procedures is one to take a copy of an object. This automatically generates procedures like those given in Figure 4.24. In these procedures, the copy is created with dummy values and then the fields are re-assigned to the copied values, one at a time. For scalars, this is a simple assignment. For **pntr** values, this requires a recursive call to the deep copy procedure. For vectors, some extra work has to be done to create a vector of the correct dimensions. This is then populated by element-at-a-time assignments.

```
proc( pntr X,; proc( pntr, pntr -> bool ) DC -> pntr )
  begin
    structure phoneEntry( string Pname; int Pnumber; pntr Paddress )
    let result := phoneEntry( "", 0, nil )
    result( Pname ) := Y( Pname )
    result( Pnumber ) := Y( Pnumber )
    result( Pname ) := DC( X( Pname ) )
    result
  end
```

a) Deep Copy for the *phoneEntry* structure.

```
proc( pntr X, Y; proc( pntr,pntr-> bool ) DC -> bool )
  begin
    structure array( *c*int element )
    let result := array( vector 0::0 of vector 0::0 of 0 )
    result( element ) := vector lwb( X ( element ) )::upb( X( element ) ) of vector 0::0 of 0
    for i = lwb( X ( element ) ) to upb( X( element ) ) do
        result(element )( i ) :=vector lwb( X ( element )( i ) ) ::upb( X( element )( i ) ) of 0
    for i = lwb( X ( element ) ) to upb( X( element ) ) do
      begin
        for ii = lwb( X ( element )( i ) ) to upb( X( element )( i ) ) do
          begin
            result( element )( i) ( ii ) := Y( element )( i) ( ii )
          end
      end
    result
  end
```

b) Deep Copy for the Vector Structure.

Figure 4.24 Two Examples of Deep Copy Procedures.

The deep copy facility consists of a general purpose procedure, with the same structure as the traverser in Figure 4.12 and the deep printer in Figure 4.17, and a copy procedure maker, similar to that described for making printers.

4.3.4 Summary.

These three procedures have exploited the power of a compiler callable at run-time. Purely polymorphic programs have been written to achieve functions apparently unavailable in a strongly typed programming language. The technique of binding together string representations of data structure and algorithm and then compiling the result is a safe way of circumventing the restrictions of the type system, without any loss of data security.

4.4 Compiler Tools.

Some of the most powerful tools available in UNIX™ are those for the automatic generation of compilers, such as LEX and YACC. In the PS-algol environment an equivalent set of tools has been created by Stephen Blott, working as a vacation student for three months. These are: *lgen*, which is a lexical analyser generator; *pgen*, a parser generator; and *sgen* a syntax-directed editor generator [Blott and Campin, 1987].

4.4.1 A Lexical Analyser Generator.

Lexical analysis is the process of taking a stream of characters and returning them as a stream of semantically meaningful tokens or "lexemes". The structure of a lexical analyser will be similar for all languages and so the creation of lexical analysers is a process which may be automated. If programmers create lexical analysers directly, they end up rewriting much the same code. *lgen* is a program which, given the description of a language, will return a lexical analyser for that language.

In essence, *lgen* is a procedure which takes in a string containing a description of the language and returns a lexical analyser. The description consists of a set of triples, each containing: a token name; a pattern to match; and a private/public flag. The latter indicates whether this token will be returned by the lexical analyser or is only for internal use. The pattern is in the form of a stylised regular expression (RE) which can be either: a literal; the token name of another lexeme; a repeated RE; an optional RE; a sequence of REs; or an alternative between a number of REs. Some examples are:

```
capital: A | B | ..... | Z              ! alternative literals
letter: "capital" | a | b | .... | z    ! alternative token and literals
name: "capital" [ "letter" ]           ! sequence of token and repetition of token
title: "Mr" | "Mrs" | "Ms" | "Dr" | "Sir"
fullname: <"title"> ' "name" [ ' "name" ]   ! sequence of optional token,
                                            ! space ( indicated by ' ), token and
                                            ! repetition of space and token.
```

The lexical analyser returned by *lgen* is also a procedure - this time taking in a pointer to an "input stream" and returning a package of a table of all the lexemes found and a lexical analyser bound to this input stream. The input stream consists of a pair of procedures, one of which returns the next character in the input while the other takes back a character into the input stream for re-use. The bound lexical analyser returns the next lexeme found in the input stream as a triple: the actual string which makes up the lexeme; the token name of the lexeme class; and a pointer which is available to the user to add extra information if required. By next lexeme is meant the longest string starting at the next character and representing a single lexical unit. In the above language, the lexical analyser would return a *fullname*, if it could match one, and not the title, etc.

Figure 4.25 shows the use of *lgen* for the case in which the lexemes of the string, *INPUT*, are to be retrieved according to the above grammar.

```
structure lexan.box( proc( pntr -> pntr ) lexan.place )      ! Holds a lexical analyser.
structure l.stream( proc( -> string ) get; proc( string ) put )  ! An input stream.
structure l.pack( pntr spellings; proc( -> pntr ) lexan )    ! A bound lexical analyser.
structure s.entry( string s.lexeme, s.token; pntr s.tail )   ! A lexeme.

let language = "..... the language as given above....."
let inputStream =                                            ! An input stream which
  begin                                                      !   takes for its input, the
    let inputPointer = 0                                     !   string, INPUT.
    let theINPUT := INPUT
    let theGet = proc( -> string )                           ! Get next character.
        { inputPointer := inputPointer +1; theINPUT( inputPointer | 1 ) }
    let thePut = proc( string IN )                           ! Put back a character.
      {     theINPUT := IN ++ theINPUT( inputPointer + 1 | ..... )
            inputPointer = 0 }
    l.stream( theGet; thePut )
  end

let genLexan = lgen( language )( lexan.place )     ! Generate a lexical analyser for the
                                                   !    language.
let myLexanPack = genLexan( inputStream )          ! Generate a procedure which will
                                                   !    analyse INPUT.
let myLexan = myLexanPack ( lexan )                ! Unpack the lexical analyser.
while true do
  begin
    let next = myLexan( )                          ! Return the next lexeme.
    print "A ", next( s.token ), " has been found with value ", next( s.lexeme )
  end
```

Figure 4.25 Lexical Analysis Using *lgen*.

The details of *lgen*, which uses a series of standard algorithms, are beyond the scope of this thesis. In outline, *lgen* proceeds as follows:

1/ Take the string containing the language and produce a vector of pointers to structures containing the various lexeme classes.

2/ Take the vector of lexemes and produce a non-deterministic finite-state machine - this will be very large.

3/ Turn this into a deterministic finite-state machine.

4/ Turn this into the smallest finite-state machine.

5/ Optimise the implementation of this machine to produce simpler and more efficient code.

6/ Produce and return a procedure which interprets this final machine and requires an input and output stream.

It is clear that none of the parts of this implementation are intrinsically revolutionary, but that their implementation has been facilitated by using a language which combines: (a) object identity; (b) computational completeness; and (c) first-class procedures. At each stage, it is possible to create structure classes which exactly correspond to the objects being handled. (The input stream structure is a case in point.) A lexical analyser is represented directly by a procedural object. This can be created by the generator and stored and manipulated as if it were a procedure entered by hand. This greatly simplifies the programmer's view of the world.

4.4.2 A Parser Generator.

The second in this set of tools is a generator for parsers, which take the lexemes found by the lexical analyser and form them into a structure which reflects the meaning of the input as a whole. The generator, *pgen*, takes in a BNF description of the language and returns a parser, which takes in a lexical analyser and returns a parse tree of the string.

pgen takes two parameters, a vector of specifications and a vector of the names of nonterminals for which parsers are required. The specifications are objects with two fields, a string containing a textual specification of some rule in the BNF grammar and a user-defined procedure which is to be executed whenever a grammatical unit of this type is encountered. This procedure can be used for any incidental computation to be executed as the tree is being built up.

A rule is of the form:

name ::= list of names of constituents separated by spaces

where the constituents are either non-terminals or terminals of the language. The latter are essentially lexemes and are distinguished by being preceded by "#".

The associated procedure takes in a vector of pointers to "values" of the children of this node and returns a "value" for this node. This procedure may be used to maintain an abstract syntax tree, keep a current value, or for any other activity.

The result of *pgen* is a PS-algol table of parsers, one for each rule in the specification, whose name is in the required list. Each of these parsers takes in an *l.pack* produced by *lgen* and returns a pointer to a parse tree, whose leaf nodes are *s.entry* nodes produced by the lexical analyser and whose other nodes reflect the tree structure of the result using the following PS-algol structure:

```
structure p.tree(    pntr gSymbol;      ! points to a symbol of the language
                     *pntr children;    ! point to child nodes
                     pntr parent;       ! points to the parent node
                     pntr value )       ! points to the value manipulated
                                        !      by the user-defined procedure.
```

The use of *pgen* will be illustrated by returning to the name example given in the description of *lgen*. The lexical language is modified by removing the last rule, which is now moved to the parser. The decision between where lexical analysis ends and parsing begins can be a nice one. The division is indicated by deciding which rules go in the lexical language and which in the parser description. The tension is usually between over-complicating the parser and retaining sufficient structure after the lexical analysis stage. A program to parse names into their component parts is given as Figure 4.26

```
structure a.parser( proc( pntr -> pntr ) the.parser )
structure nameBox( string aName )
let buildName = proc( *pntr V -> pntr )     ! This procedure will build up the name
  begin                                     !    as it is parsed into a string.
    let fullname = ""
    for i = lwb( V ) to upb( V ) do fullname := fullname ++ V( i )( s.lexeme )
    nameBox( fullname )
  end
let spec = @ 1 of pntr [    p.spec( "fullname ::= #title #name #name", buildName ) ]
let required = @ 1 of string [ "fullname" ]
let parsers = pgen( spec, required )          ! Generate a set of one parser, for fullname.
let parserPack = s.lookup( "fullname", parsers )   ! Retrieve the parser.
let nameParser = parserPack( the.parser )          ! Unpack the parser.
let wholeNameTree = nameParser( myLexanPack )! Apply it to the lexical analyser.
```

Figure 4.26 Parsing Using *pgen*.

The result of this, when run on an *INPUT* value of "Mr Richard Cooper", results in a parse tree of four nodes. *wholeNameTree* points to a *p.tree* node whose value is the whole input string (rebuilt by *buildName* when the parser encounters a fullname). The three children nodes to this one are three lexemes returned by the lexical analyser for the three parts of the name.

4.4.3 Compiler Tools Summary.

These two tools and the syntax directed editor which accompanies them illustrate once more the power of first-class procedures in the language. At each point in which the object being manipulated is a piece of computation, a PS-algol procedure is used to represent it. *lgen* creates lexical analysers and the program representing it produces procedures. The ability to pass around procedures in this way greatly simplifies the implementation of an otherwise extremely complex task.

The other simplifying construct used here is the **pntr** type. As the fields of the complex objects of this type can be of any type, the implementation has been able to create object classes for any kind of object, whatever its constituent parts. It was decided, for instance, that the most appropriate constituents of an input stream are a get procedure and a put procedure. The implementation of the input stream reflects

this exactly. This ability to create structures which truly reflect the nature of the modelled objects clarifies the program greatly.

4.5 Conclusions.

This chapter has described a number of tools that have been provided for use within PS-algol's persistent environment, all of which have been written in PS-algol and thus within the environment. These have ranged from relatively low-level user interface tools to sophisticated tools such as the browser or the compiler tools. They have been provided in three different ways. The menu facility is a standard function of PS-algol, which is automatically available to any PS-algol program. The browser is a stand-alone program. The other tools are procedures which have been stored in the Persistent Store for retrieval by any program that knows where to find it. The question of supporting the storage and retrieval of procedures in a systematic way is tackled in Chapter 8.

It has been relatively easy to construct the tools. Here is a list of facilities which have been found beneficial:

- the language is computationally complete. Unlike in some database programming languages, any tool can be specified in PS-algol;
- the persistence mechanism provides a simple method for storing utilities once created;
- data-type completeness simplifies the writing of any program, since exceptions to general rules do not have to be remembered;
- the complex object structures are invaluable for setting up program objects which correspond exactly to the object being modelled - it was easier to create a dialogue box when a "light button" object could be referred to;
- the graphics constructs are powerful enough to create user interface tools in a simple and consistent way;
- the graphics constructs also allow these interface tools to be stored and retrieved in the same way as other tools, thus eliminating the need for a library of interface tools reached in one way, while other tools are reached in another way;
- the provision of first-class procedures facilitates the writing of tool generators - utilities which create tools appropriate for particular data structures;
- first-class procedures also enable the direct manipulation of processes or operations - again a menu or dialogue box facility is much easier to implement if procedures may be directly associated with the event which invokes them;
- first-class procedures further enable the representation of some objects as abstract data types - a dialogue box, for instance, is represented as an abstract data type;

- the availability of the first-class compiler enables the writing of procedures which are polymorphic over any data class without violating the type security of the persistent store.

Therefore, the chapter has shown that the persistent environment is at least sufficiently powerful to provide the low-level aspects of application programming. The language has powerful primitives and can be extended with suitable tools for a number of purposes.

The next three chapters continue to consider the construction of applications in PS-algol: firstly, how to construct a database application directly in PS-algol; then how to construct a Relational Database System which can be used to write database applications; and finally, how Semantic Data Models can be constructed in PS-algol. Chapter 8 shows how to augment the PS-algol system with system construction tools.

Chapter 5. Building a Database Application in PS-algol.

PS-algol can be used either as a language in which to program database applications or as an implementation vehicle for higher-level data models. In this chapter, a typical example of programming an application is described, leaving until Chapters 6 and 7 descriptions of building data models. This chapter presents the methodology for designing and constructing an application.

In particular, the modular and incremental construction of the system will be described, with particular emphasis on the identification of re-usable software components and the coherent structure which was imposed on the structure of the whole program. At the same time, problems in using PS-algol will be identified and mechanisms for circumventing them described. These include the development of a transaction system on top of the primitive constructs of PS-algol - a further illustration of the extensibility of the system as a whole.

The specific application described here is a database for bibliographic references [Cooper *et al.*, 1987b]. The program provides facilities for bulk loading and dumping of references, editing and browsing of references, and the automatic creation of bibliographies. The program is described as an example of the kind of facility which can be created in PS-algol. In doing so, the graphics facilities are used to provide a good user interface, persistence is used to limit the programming of data storage and first-class procedures are used to model the operations provided.

First the setting for such a program is described, then an overview of the facilities provided is given. The implementation method is described and, finally, the benefits and drawbacks of the PS-algol language in this particular exercise are discussed.

5.1 Document Manipulation Programs.

A persistent store is ideally suited to the development of software which supports the production of all kinds and sizes of document. The storage within the same space of the text and diagrams of papers, a body of references and all the software to maintain them provides an extremely powerful environment for developing both the system and the documents. Such a system could encompass, among other functions, word-processing, diagram manipulation, automatic bibliography construction, the production of indexes, page make-up and the maintenance of mailing lists. A persistent environment has the particular merit of facilitating the replacement of code, so that improved versions of tools can be easily inserted and more than one tool could be provided for any function. The user can pick a favourite word processor or choose a simple one for a simple job, turning to a more powerful one where necessary.

When producing software for document production, a number of problems arise:

- the conflict between ease of use and fine control over the structure of the document;

- the past history of users of other products who want to be able to use all the features to which they are accustomed and thus avoid relearning;

- the document's layout may need to be completely re-shaped, while its contents remain the same;

- the desirability of linking together the various components of a document, systematically and flexibly.

A persistent environment assists in the solution of these problems. Versions of software may be provided with equal availability so that the user can choose the most appropriate or the most familiar. These versions can be constructed using components such as the tools described in Chapter 4, with all versions re-using the same components, without any unnecessary recompilation. Moreover, all of the versions may be made available through the same mechanism, either by providing a set of programs as usual, or by by providing a dynamically varying menu of the versions, using the Chooser (see Section 4.1.4), for instance. The result could be a complete document preparation system, in which the user may specify which editor is used, which indexing system, etc. The architecture for such a system is shown as Figure 5.1, in which is seen a complete system depending upon a set of editors, which in turn depend on a set of low-level components.

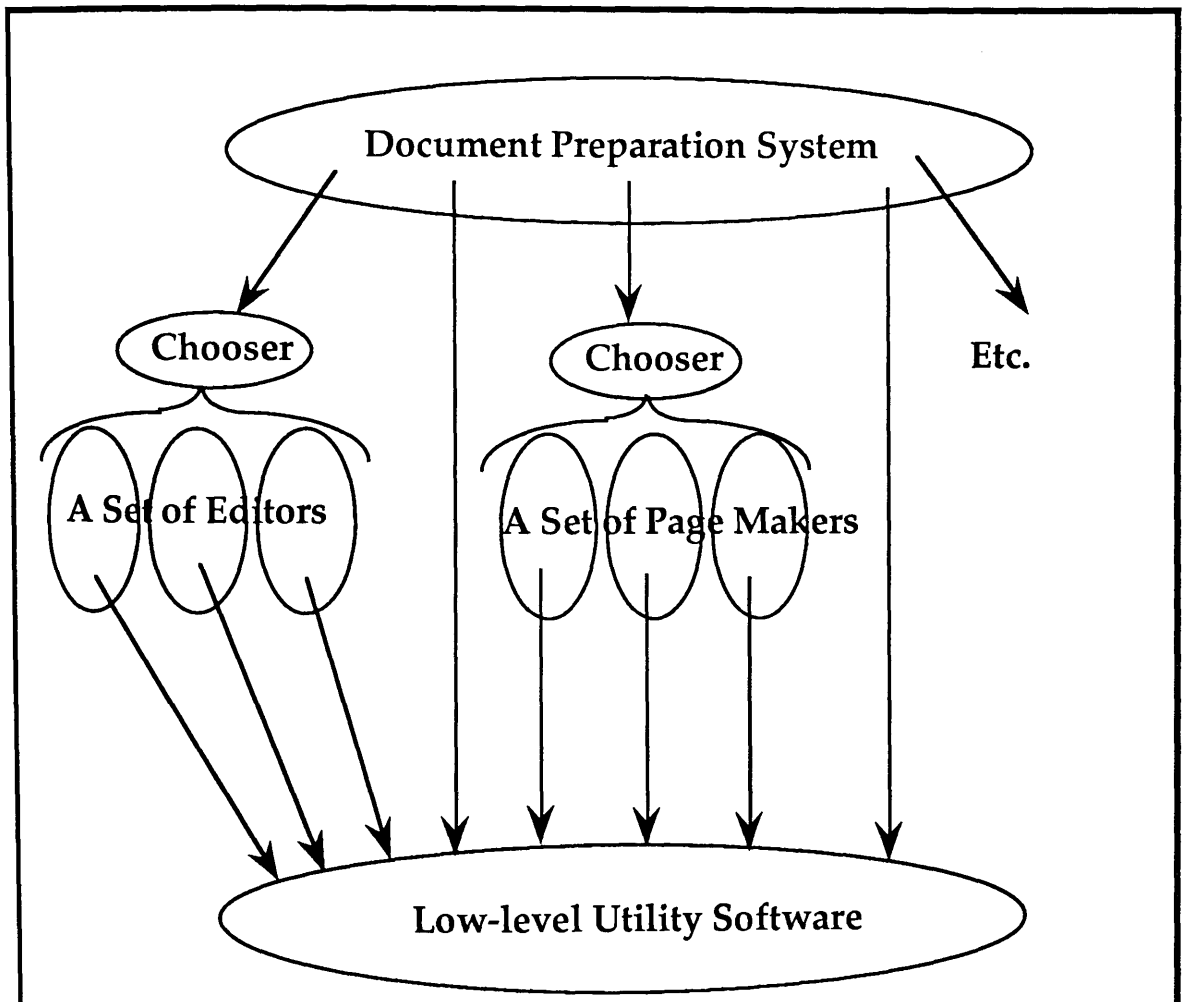


Figure 5.1 An Architecture for Document Production.

Two simplifications are present in this architecture. Firstly, if one of the low-level modules is replaced, the calling modules can be made to rebind automatically to the new. Secondly, if a new version of a higher-level module is inserted, this can immediately be linked into the calling program. All of the links are updated automatically and so the insertion of new versions is simplified.

Software can be provided which presents a different "view" of the same document to vary its presentation. It also becomes easier to provide facilities for mixing classes of data in a system which allows any data structure to be stored. For instance, it is simple to describe the insertion of a diagram into a piece of text in a more meaningful way than, say, the MacIntosh Clipboard [Apple 1984], which merely copies the component in one particular representation and keeps no dependency record. Viewing all the various components of the system, including simple data, complex data structures, pictures and operations, as objects has a considerable effect in reducing the information load on the programmer and gives the resulting program a closer relation to the system being modelled. The **pntr** type enables appropriate polymorphism, in that objects of any type may be put into and removed from the "clipboard" by the same code, whatever their type and representation.

As a start to developing a complete system for manipulating documents, a bibliographic reference database program, based on parts of the Scribe system [Unilogic, 1985], has been implemented. This includes both facilities for the maintenance of references (they can be entered, edited, deleted and browsed) and for scanning a document for references and then automatically creating a bibliography for all or part of the document. A variety of procedures is available for producing the bibliography in a number of different text processing languages. The database also holds the formats required by a number of journals and there are facilities for maintaining this set of formats. The method for incrementally adding compiled code, for generating different formats and supporting a variety of hosted text processors illustrates the significant advantage provided by first class procedures in a persistent store.

5.2 System Overview.

The Bibliographic Reference Database Program (BRDP) manipulates **references**. A reference may be regarded as potentially consisting of the following information:

- the **type** of reference it is - whether it is a book, a paper, etc.;
- a **citation key** for insertion into the paper to be processed;
- a set of **fields** and their values, such as "author", "title", etc.;
- a set of **key-words**;
- an **abstract** of the paper;

and

- for a complete library system, the **text of the paper** itself.

The present system only makes use of the first three of these. The set of available Reference Types is derived from the Scribe Manual [Unilogic, 1985]. Associated with each Reference Type is a set of fields essential to the specification of the kind of publication, as well as a set of fields which may optionally be present. For instance, a reference of the "Article" Type must have a pages field, but one of the "Book" Type need not. The way in which a reference will be laid out in the bibliography produced by the program will vary from Type to Type and so each Type has two associated layout specifications. One of these describes the way the citation keys will appear in the final text and the other describes the way each reference in the bibliography will be laid out.

Like Scribe, the system supports a number of "Reference Formats", which determine the preferred layout styles for various publishing organisations, such as "IEEE", "CACM" and "SIAM". For each of these Reference Formats, the Reference Types available and the required fields and layout specifications for those Types may vary. Each Reference Format also contains a specification for the order in which the references will appear in a bibliography - for instance alphabetically on author name or in the order in which they are cited in the paper.

To support this taxonomy, the BRDP maintains the following structural data:

- a set of all the **field names** known to the system;
- a set of all the **Reference Types** known to the system, with default values for the fields required for this type and a layout specification.
- a set of all the **Reference Formats** known to the system, each containing a set of Reference Types and a sort order specification.

The references are divided into Topic areas for storage. The BRDP maintains a set of such Topics, each of which contains a set of abbreviations, a set of references and a set of all the authors in the set of references. The abbreviations consist of pairs of strings containing the short and long forms of the abbreviated string. They are used to shorten the amount of data which needs to be entered and stored in the table of references. Within this table, the long form of any abbreviated string may be replaced by "**@value**"* followed by the short form. These data, together with the structural data and the program modules, are stored in a database in the manner shown in Figure 5.9 later in the chapter. The organisation of this database will be described in more detail below.

The BRDP supports the following functions:

- a facility to set up a fresh database;
- editors for each of the sets of structural data (fields, Types and Formats);
- an editor for the set of Topics, enabling Topics to be added and deleted;

* Since our research is not about bibliographic systems *per se*, most of the notations were copied from Scribe.

- an editor for the abbreviations in a Topic;

- bulk load and bulk dump facilities for a Topic, permitting the data to be transferred between this program and others;

- a facility to browse the references by author name;

- an editor for the set of references in a Topic;

and
- a facility for creating the bibliography for a paper.

The last of these reads through the text of the paper, replacing the following:

- **@cite** followed by a citation key in brackets is replaced by the citation key in the layout required by the chosen Reference Format;

- **@partbibliography** is replaced by a list of references. This is the set of references found since the last @partbibliography or since the start of the text if this is the first one. The layout of the references and the order in which they appear also depend on the Reference Format;

- **@bibliography** is replaced by the list of all the references since the start of the text.

When creating a bibliography, the user specifies the following:

- the **Reference Format** to be used;

and
- the **Output Medium** to be used.

The former determines the layout of the final document, by referring to three strings:

- the **sort order** associated with the Reference Format. This consists of a series of letters which specify the order of fields on which the references are to be sorted. For instance "AY" means sort first on author, then on year.

- the **key layout** associated with the Reference Type within this Format. This is a set of strings concatenated with the following structure:

  - "@" followed by a field name means print the value of the field;

  - "#" followed by a string means print that string;

  - "'n" and "'t" mean newline and tab, respectively.

- the **reference layout** associated with the Reference Type within this Format is a string structured in the same way as the key layout.

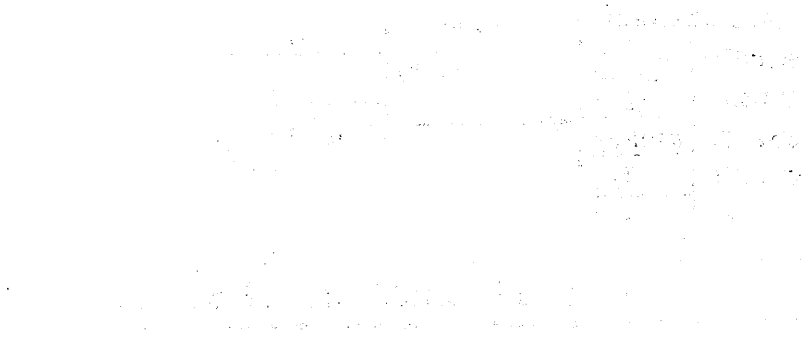
The Output Medium determines the way in which the output will be produced, whether to the screen or to a text file or to a file suitable for input in a text processor, like TEX [Knuth 1984], for instance.

To appreciate the scale of the implementation task, the reader may read next the functional description given in the remainder of Section 5.2. The important issue is the extent to which the implementation was facilitated by using a Persistent Programming Language. This is discussed in Section 5.3.

5.2.1 Introduction to Using the System.

The modules of the system are controlled through an interface consisting of a hierarchy of menus and dialogue boxes of the type described in Section 4.1, each of which contains options to obtain help and to quit to the next highest level. The other options either generate a further sub-menu or provide a dialogue which controls interaction with the user to achieve the operation selected.

The structure of the menu hierarchy is shown in Figure 5.2. Menus are shown in rectangular boxes and forms in rounded boxes. Moving down the hierarchy is achieved by clicking over a light button on one of the forms or menus. At the end of a chain of selections, the user interacts via a dialogue consisting of operations provided by different modules of the program or the following tools described in Chapter 4: the simple String Editor (4.1.6) indicated as"S.edit"; the Chooser (4.1.4) to select an object of the required kind; and the More facility (4.1.2) to show the requested text or list of object names. "Select" means that the user indicates which object is to be operated on by clicking the mouse over the form element corresponding to that object. "Sord" means call the Sort Order Editor (see Figure 5.6), while "Refer" means enter the Reference Editor (see Figure 5.8).



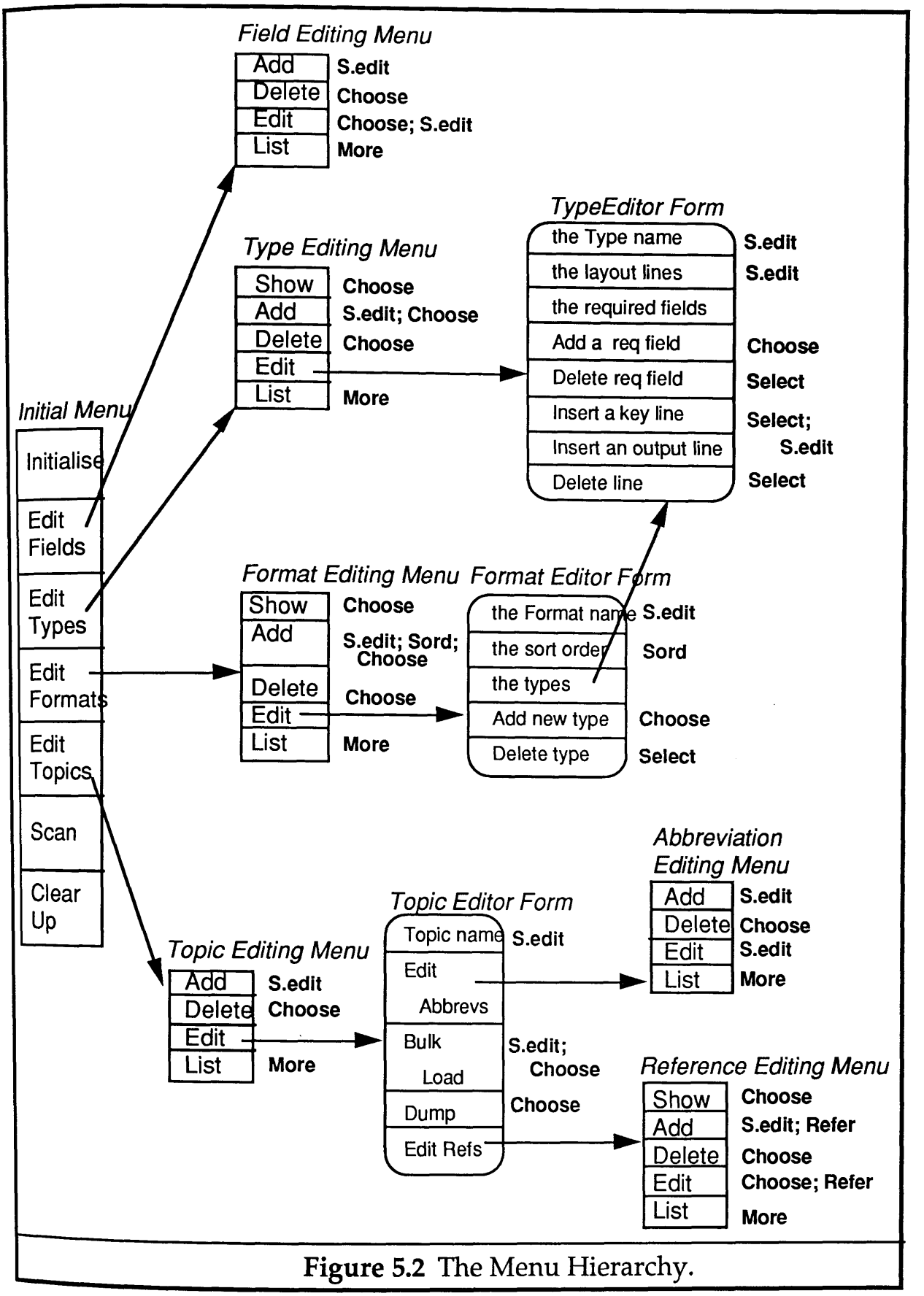


Figure 5.2 The Menu Hierarchy.

5.2.2 Getting Started.

The initial display and the first level of menus is shown as Figure 5.3. The start up screen consists of the heading and the vertical menu shown in bold on the left-hand side of the screen. The options of this menu have the following functions:

- **Help** - displays a short description of the options of this menu at the centre of the screen, until the mouse button is clicked. All "help" buttons function in this way;
- **Edit fields** - allows the vector of field names to be edited;
- **Edit types** - allows the table of Reference Types to be edited;
- **Edit formats** - allows the tables of Reference Formats to be edited;

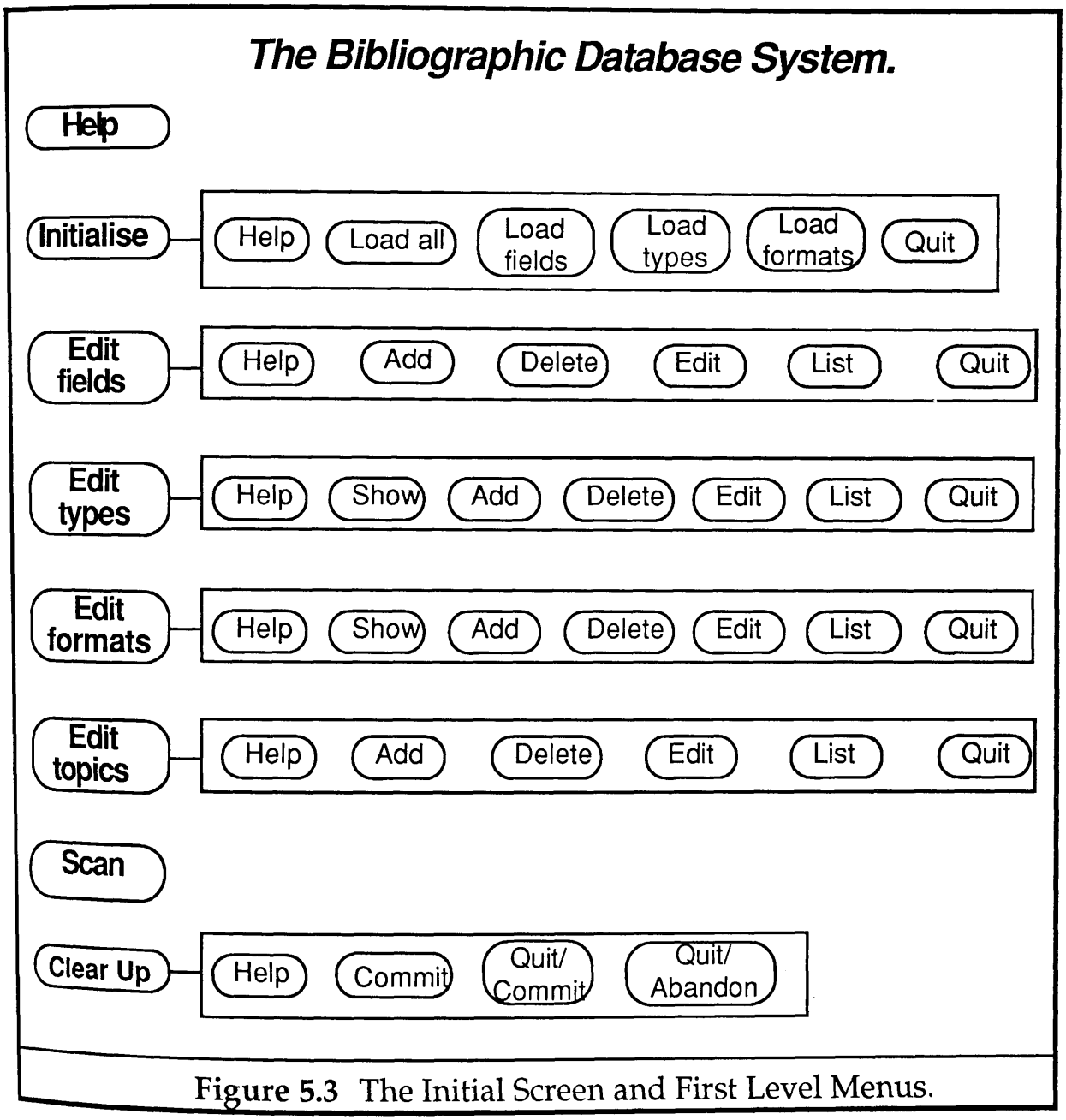


Figure 5.3 The Initial Screen and First Level Menus.

- **Edit topics** - allows the set of Topics to be edited;

- **Scan** - initiates the dialogue which leads to the building of a bibliography;

- **Clear Up** - handles both the committal of data to the database and exit from the program.

5.2.3 The Set Editors.

Four of the options of the initial menu lead to sub-menus which control the editing of a set of objects. These sub-menus have similar structures. They all contain the options:

Add - add an item. Typically, this calls the String Editor to allow the user to specify an identifier for the new item and then makes further calls to the String Editor, to the Chooser or to the editor specific to an item of this kind to generate the values of other attributes of the item.

Delete - delete an item. The item to be deleted is selected via the Chooser.

Edit - edit the value of the item. Again an item is selected via the Chooser and then the current value is provided to the editor of the appropriate kind, which will announce itself by creating a new window in the screen in which to operate. If the identifying information is edited (for instance, the Reference Type name), a new object is created and the old object is left intact. If only a change to the identifier is required, then after the editing has been done, the old object must be explicitly deleted. This design decision is discussed further below (5.3.5).

List - provide a list of the identifiers of every item of this kind with the **More** module.

Additionally, the menus for editing the Formats and the Types include:

Show - display one of the items of this kind. The item to be displayed is selected by use of the Chooser. The information remains on the screen until the mouse button is clicked.

5.2.4 Editing the Set of Known Field Names.

To edit the set of field names, choose the **Edit fields** option of the initial menu. The sub-menu contains the options, **Add, Delete, Edit** and **List**, which operate as just described. In particular:

- adding a field name consists of typing a new name into the String Editor;

- editing consists of choosing a field name and then changing it with the String Editor.

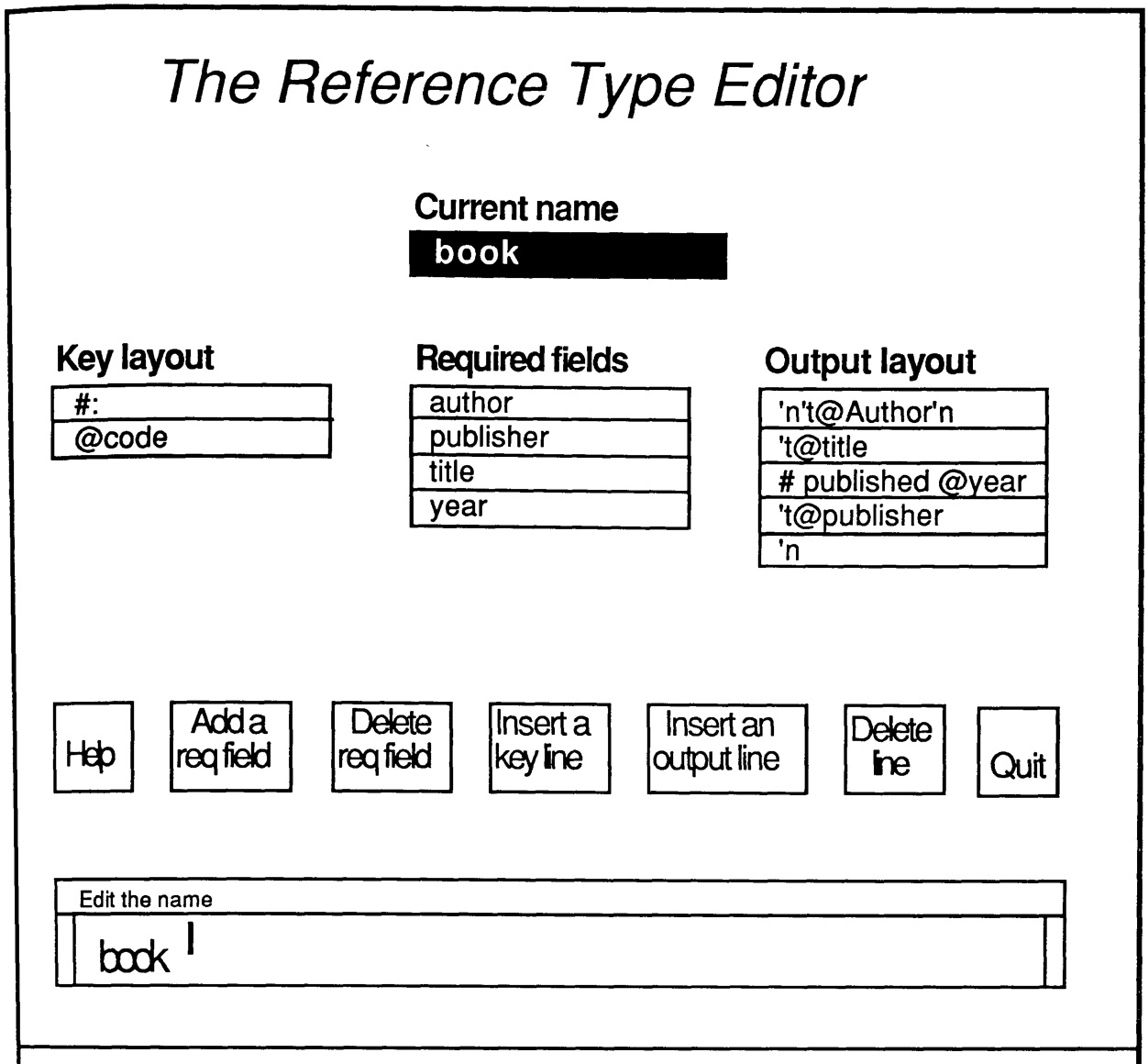


Figure 5.4 The Reference Type Editor.

5.2.5 Editing the Set of Default Reference Types.

Selecting the **Edit types** option of the initial menu summons the sub-menu, with all of the usual options, including **Show**, with the following particular details:

- Adding a new Type requires three calls to the String Editor to supply the Type name, a citation key layout and a reference layout. Then fields are added to the required fields list by menu selection from the vector of valid field names.

- Editing a Type requires the Type to be edited to be selected with the Chooser and then uses the Type Editor (Figure 5.4). This announces itself as a new window on the screen, within which the Type name is displayed at the top, under which is shown the information about the Type in three columns: one each for the key layout, output layout and the list of required field names. Selecting the Type name or any of the layout lines summons the String Editor to change them. The editor also has a row of

light buttons at the bottom of the display, which include "help" and "quit" buttons and also:

Add required field: select a field to add via the Chooser;

Delete required field: click the mouse over the field to be deleted;

Insert key layout line: click over the position at which the line is to be inserted and then input it via the String Editor;

Insert output layout line: as for inserting a key layout line;

Delete layout line: click the mouse over the line to delete.

- A Type is displayed by choosing the **show** option and then using the Chooser to pick which one to display. The Type is then displayed in a consistent fashion to the layout of the editor.

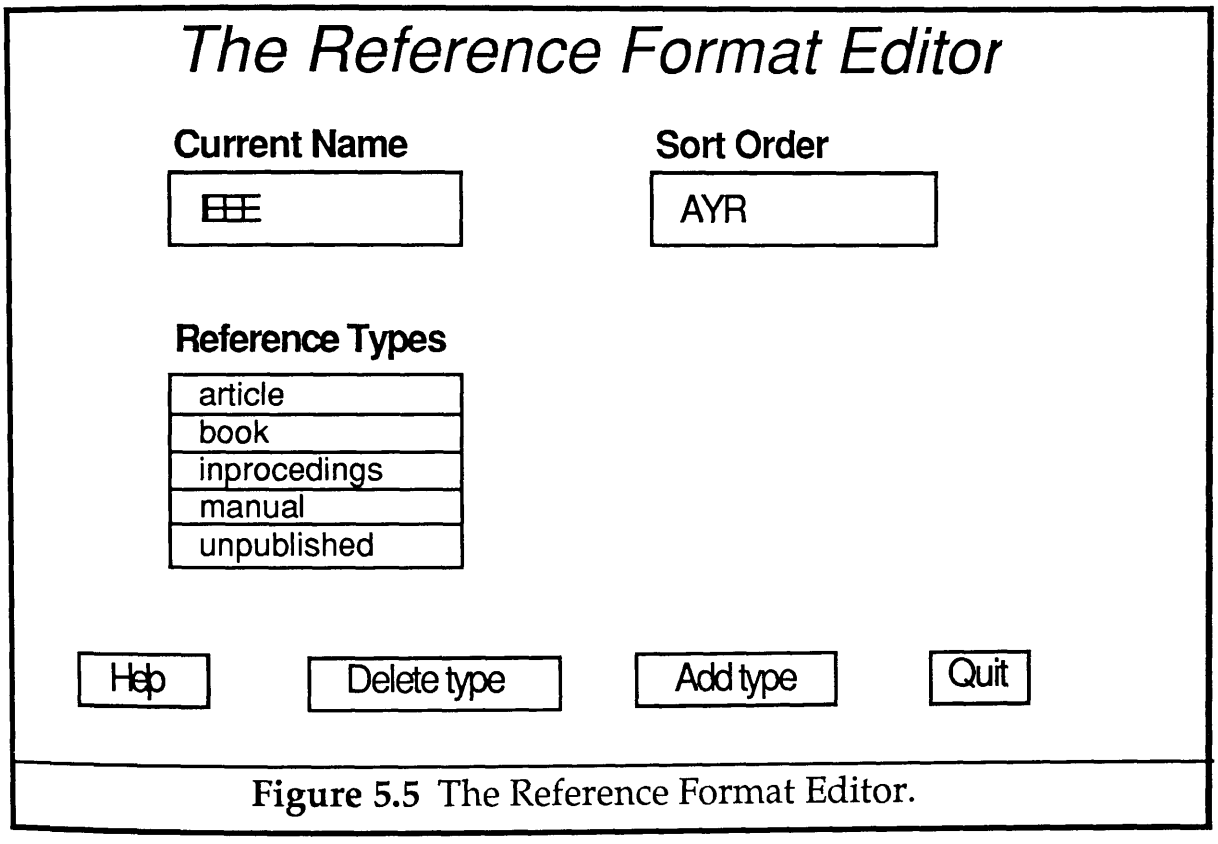


Figure 5.5 The Reference Format Editor.

5.2.6 Editing the Reference Formats.

The **Edit formats** option of the initial menu brings up a sub-menu, which has the full set of five options which operate as already described, with the following particulars:

- Adding a new Reference Format consists of providing a new name via the String Editor and a sort order via the Sort Order Editor (described in the

next section). Then the user loads in Reference Types from the default Reference Type table, via the Chooser.

- Editing a Reference Format is done via the Reference Format Editor shown as Figure 5.5, after selection of a Format to edit by use of the Chooser. This displays the name and the sort order at the top of its window and the set of Reference Types vertically. Each of these may be clicked over to summon the String Editor, the Sort Order Editor or the Type Editor, respectively. There are further light buttons at the bottom of the display, including "help" and "quit" as usual, as well as buttons to add a new Type (via the Chooser) and delete a Type (by clicking the mouse over it).

- Displaying a Format requires selecting which one to show using the Chooser. It is then displayed in a layout similar to the editor's.

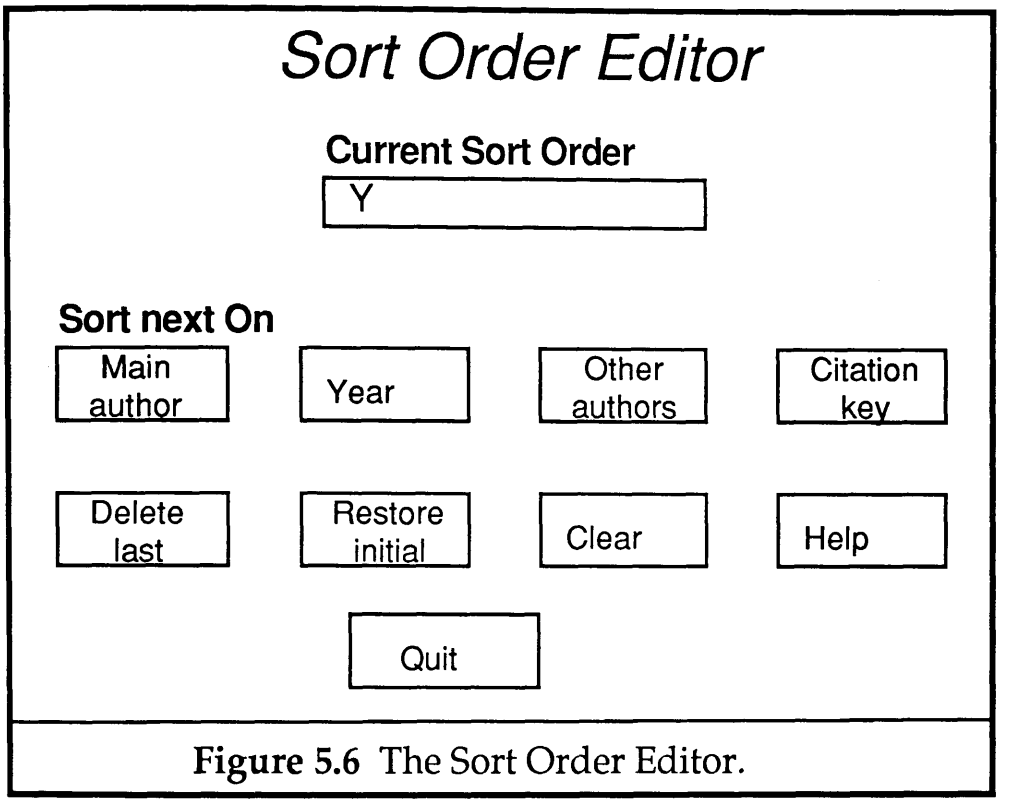


Figure 5.6 The Sort Order Editor.

5.2.7 Editing a Sort Order.

The sort order for a Reference Format is changed by using the Sort Order Editor shown in Figure 5.6. The current value of the sort order is displayed towards the top of the display, and under this there are nine light buttons, including the "Help" and "Quit" buttons. The buttons on the top line insert further sort key letters into the sort order string, thus adding fields to break ties between references which can be distinguished on the sort order so far. The other options give the following operations:

- **Delete last** - remove the last sort key letter;

- **Restore initial** - return to the sort order string as it was on entry to the editor;

- **Clear** - clear the string to nothing.

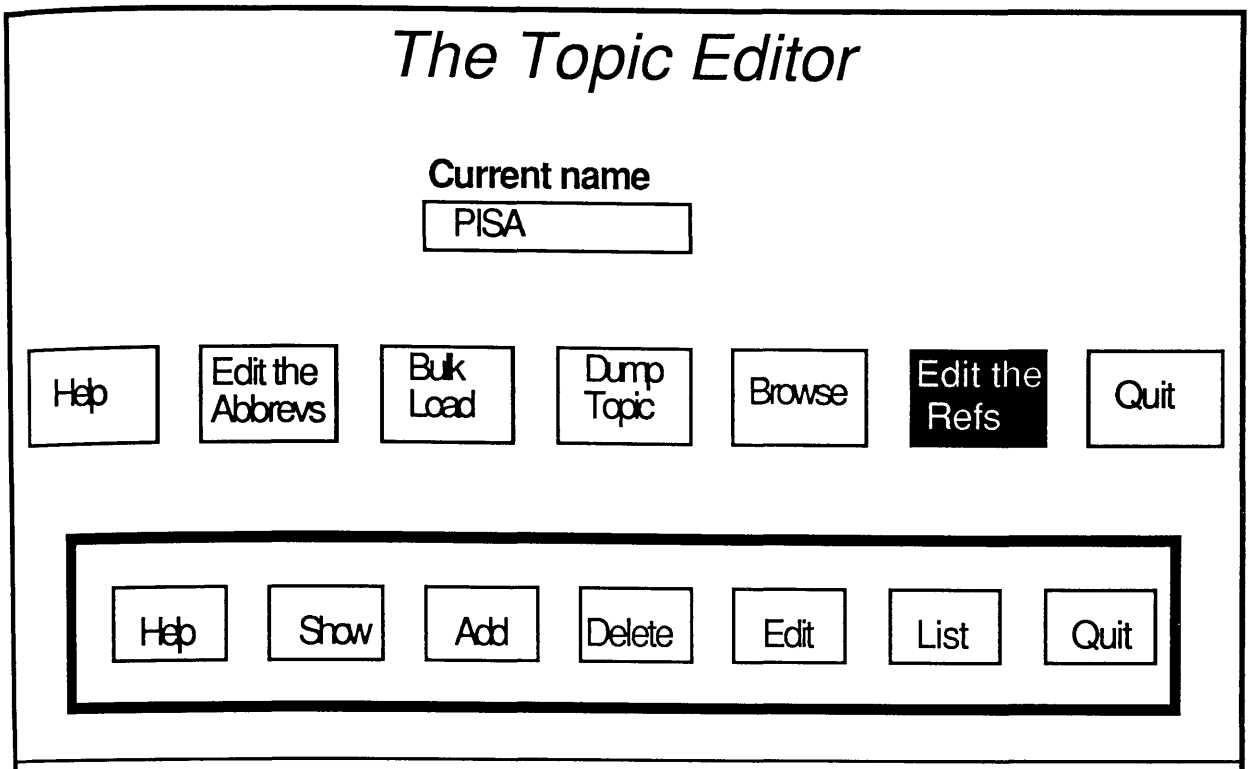


Figure 5.7 The Topic Editor and Reference Editing Menu.

5.2.8 Editing The Topics.

The set of Topics may be edited by selecting the **Edit topics** option of the initial menu. The sub-menu which then appears does not include a "show" option, as there is too much information stored for each Topic to fit on the screen. The **Delete** and **List** options function as previously described, while:

Adding a new Topic requires a new name to be entered via the String Editor. A new entry in the database is created pointing to three empty tables which will hold the abbreviations, the author lists and the entries.

Selecting the edit option summons the Topic Editor which is shown as Figure 5.7. It displays the name of the Topic at the top and this may be clicked on to call the String Editor to change it. The operations of the Topic Editor are selected by the row of light buttons underneath this name. They include "Quit", "Help" and the following:

Edit the Abbrevs - a sub-menu appears with the usual structure for menus which control the editing of sets of objects. **Delete, List** and **Show** all function in the usual way. The other options work as follows:

adding an abbreviation requires two strings - the abbreviation and the full form - both of which are entered via the String Editor;

editing an abbreviation proceeds by selecting which to edit from a menu of the short forms and then modifying the short and long forms using the String Editor.

Bulk Load - a file name is requested using the String Editor and the format of the file is requested using the Chooser. All of the references and abbreviations found in the file are loaded into appropriate slots in the Topic's structure. At present, the file must be in Scribe format or Refer format.

Dump Topic - the contents of the Topic are dumped in a format selected by the Chooser from those available. If they are to be dumped in a re-loadable format (e.g. Scribe), then a file name is requested using the String Editor. If, on the other hand, the dump is for viewing purposes, an Output Medium is selected via the Chooser. For further description of the Output Media, see the section on producing the bibliography.

Browse - the contents of the Topic are opened for browsing. The only browsing mechanism implemented as yet consists of traversing lists of papers with the same author. Therefore the browse option starts by requesting an author name by menu and then traversing the list by using a menu of the following options:

List - display a list of all the keys;

Show - display details of the current paper;

Next - proceed to the next paper;

and **Find** - supply a year and go to the first paper of that year.

Edit the Refs - a sub-menu appears underneath the row of light buttons, which includes the same set of options that have been seen in the higher level menus. This is the lower set of buttons, shown boxed in Figure 5.7. The options **Delete** and **List** behave in the expected way, while **Show** displays an entry in a form compatible with the Reference Editor. The **Add** option requests a key via the String Editor and then calls the Reference Editor to fill in the fields. The **Edit** option calls the Chooser to select an entry to edit and then calls the Reference Editor.

5.2.9 The Reference Editor.

The Reference Editor is shown in Figure 5.8. At the top the following are displayed: the key under which it has been stored in the database; the Type of reference it is; and the list of authors. Underneath this field are shown the required fields and under these, the optional fields. Selecting the key or any of the fields results in the String Editor being called to modify these. Selecting the Type allows it to be changed using the Chooser. Towards the bottom, there is a row of light buttons, including "Quit", "Help" and the following:

Add field - a new field name is selected from the set of valid field names and added to the list of optional fields. Then the String Editor is called to enter a value for the field.

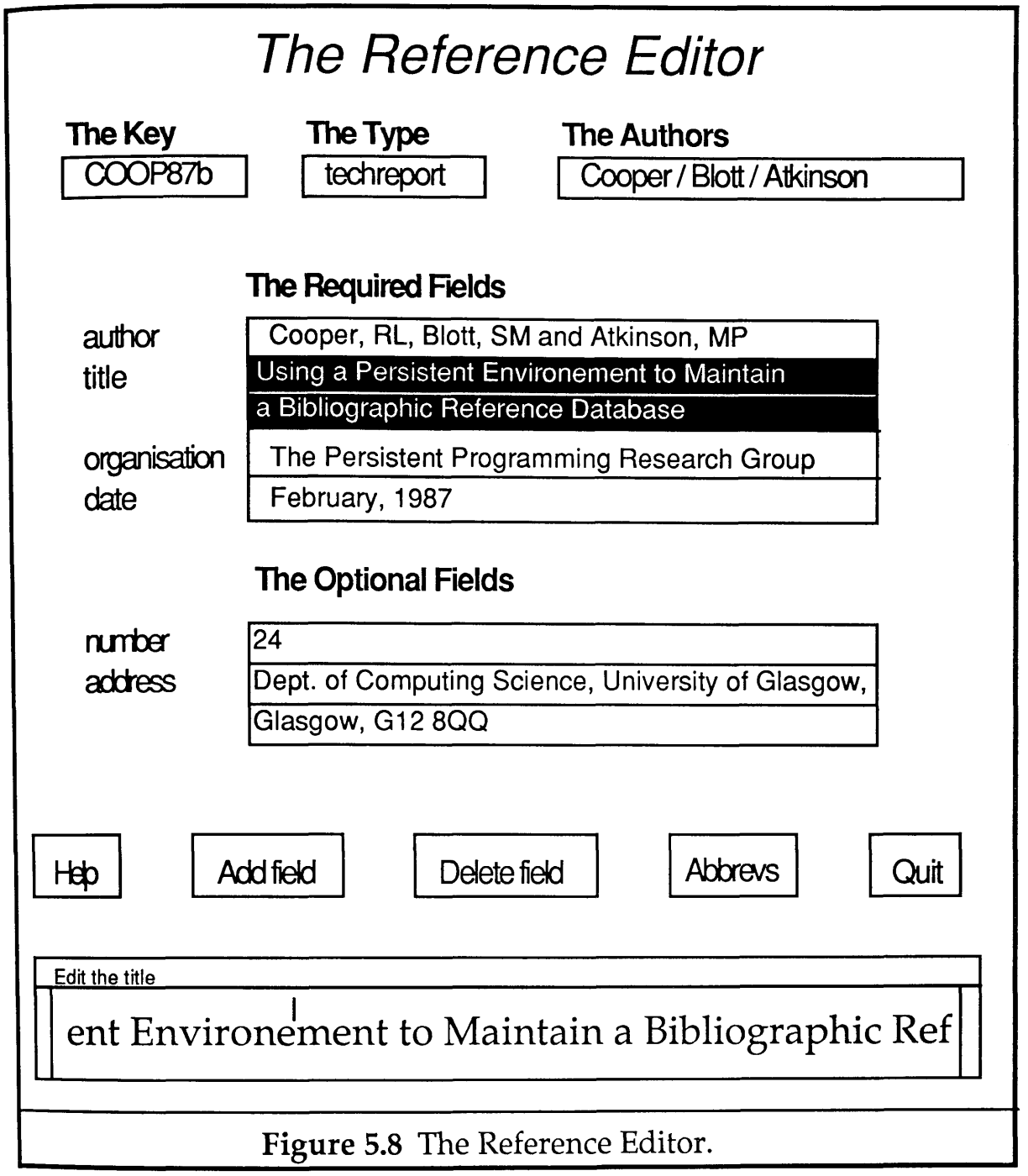


Figure 5.8 The Reference Editor.

Delete field - the field to be deleted is clicked over. Only optional fields can be deleted.

Abbrevs - clicking over this button throws a switch between displaying abbreviated strings in their short form (e.g. "@value[PPRR]) or their long form (e.g. "Persistent Programming Research Report").

5.2.10 Producing A Bibliography.

Having set up the database with all of the required information, using it to produce a bibliography proceeds as follows:

Create a text file containing the paper with all citations entered in the form "@cite[*ckey*]", where *ckey* is the citation key for the reference. The

position of the bibliography should be indicated by a line containing just "@bibliography", to get all the citations from the start to the current point, or "@partbibliography" to get all the citations from the last bibliography to the current point.

Enter the Bibliographic Database System and select "Scan" from the initial menu.

Supply, via the String Editor, a file name for the paper.

Choose a Reference Format from the menu provided.

Finally, supply an Output Medium, also by menu. This will be one of the following:

Screen - this option displays the output on the screen via More, that is, paged with mouse button clicks to "turn" the page;

File - this sends the output to an ASCII file, the name of which is requested via the String Editor;

TEX - this sends the output to a file which formatted for input to a TEX processor.

5.2.11 Finishing Off.

The final option of the initial menu is labelled "Clear Up" and provides facilities for making the changes to the database permanent and for leaving the program. Selecting the option leads to a sub-menu, which includes a "Help" option as well as:

Commit - make any changes to the database permanent and continue within the system;

Quit/Commit - commit the changes and quit the system;

and **Quit/Abandon** - quit the system losing all the changes since the last commit.

5.3 Implementation Decisions.

When starting the implementation several criteria were taken into consideration:

- the need for a consistent user interface;
- the identification of low-level modules, which would be re-usable in later programs;
- the desirability of a coherent structure to manage a large implementation task;

- the provision of a program which was flexible to use;

and
- the identification of a method for managing software modules.

With this in mind, the task was given a modular structure and the software partitioned into six sets of modules:

(i) a database initialisation program;

(ii) a program which starts the system, summoning the top-level menu;

(iii) a set of modules corresponding to each of the major tasks of the program (the browser, the various editors, etc.);

(iv) a set of low-level, but application-specific modules, such as one to return a list of references for a given author;

(v) a set of more generally useful modules, such as the Chooser or the String Editor;

and (vi) sets of parallel versions of the same utilities, such a bulk loaders.

Given this partitioning of the software, an implementation methodology was adopted which specified a location for each of the modules in sets (iii), (iv) and (v). Each of these modules was implemented as a program which stored the module in the persistent store. Such a program consists of three parts: retrieval of any values required by the module from the persistent store; one or more procedures to implement the functions of the module; and storage of the procedure(s) in the persistent store. (For storage, the procedures were packaged into structures.) The values retrieved might include both data and other procedures called by this one.

This meant that the modules could be implemented in any order. If a top-down method was chosen then a calling module, *A* say, could be written which dereferenced a called module, *B* say, after retrieving it from its designated location. This program for *A* would then compile correctly, whether or not *B* was already in place, and moreover the program would run and store *A* in its correct location. Alternatively, if a bottom-up approach seemed more appropriate, the program for *B* could be written and run first, in which case *A* could be tested as soon as it was inserted.

The methodology for managing the modules is not discussed in detail here as it forms the central theme of Chapter 8. Briefly, the three sets of modules were stored in tables whose structure was designed for holding inter-related modules. The structure makes explicit the binding between modules so that, for instance, when a low-level module was replaced, all calling modules would be rebound to the new version.

Another implementation technique was used for those modules for which several parallel versions were created ((vi) above). There were several bulk loading procedures, for instance, for different file formats. The mechanism provided here was for one module representing the Bulk Load operation and a table of loaders for the various formats. The Bulk Load operation used the Chooser to elicit from the user the choice of which loader to use. Moreover, Bulk Load functioned properly as soon as a

new loader was added to the table, always reflecting the contents of the table. These points are elaborated in Section 5.3.2.

The database structure (discussed in more detail in Section 5.3.1) was designed to maintain coherence. All of the data and the two application-specific module sets ((iii) and (iv) above) were kept in a single database. The maintenance in a single space of both the data and the software specific to an application has a considerable simplifying effect on the task of implementing the application. The only other databases which were used were the system-provided database of fonts and a database of the utilities (module set (v) above) designed for this application, but later shared by numerous other programs.

In this section, details of the design and implementation methodology are discussed. The organisation of the underlying database is described to show one way of structuring a persistent store. Then the structure of the program is described, with emphasis on the ease with which a modular structure can be developed incrementally. Next, the value of the PS-algol graphics system in producing a consistent interface is discussed. It was found that the implementation was hampered by the lack of a transaction mechanism. The reasons for this are discussed next and then a mechanism is described which has been designed to overcome the lack of fine grained concurrency control. Finally, a number of minor points concerning object identity are discussed.

5.3.1 The Bibliographic Database Organisation.

The organisation of the database used by the system is shown as Figure 5.9. In this diagram, a table is represented by a vertical line, with horizontal lines extending rightwards from it. These lines represent entries in the table, which are shown as the key joined to its associated object by an arrowed line. The key is either shown literally as a string or as a generic description. In the diagram are shown:

❶ The top level table of the database which is named "Bibliography", with password "Reference".

❷ A table of low-level procedures, accessed via the key "**%$procedures**". The table is organised in the systematic way described in Chapter 8, with each procedure accessed via its name.

❸ A table of the high-level modules, accessed via the key "**%$modules**", also organised in the structure described in Chapter 8.

❹ The help information, accessed via the key "**%$help**". The text for each help screen is stored with a string key which the program uses to find it.

❺ A table of post-processors, accessed via the key "**%$media**".

❻ A vector of all the valid field names known to the system, accessed via the key "**%$fields**".

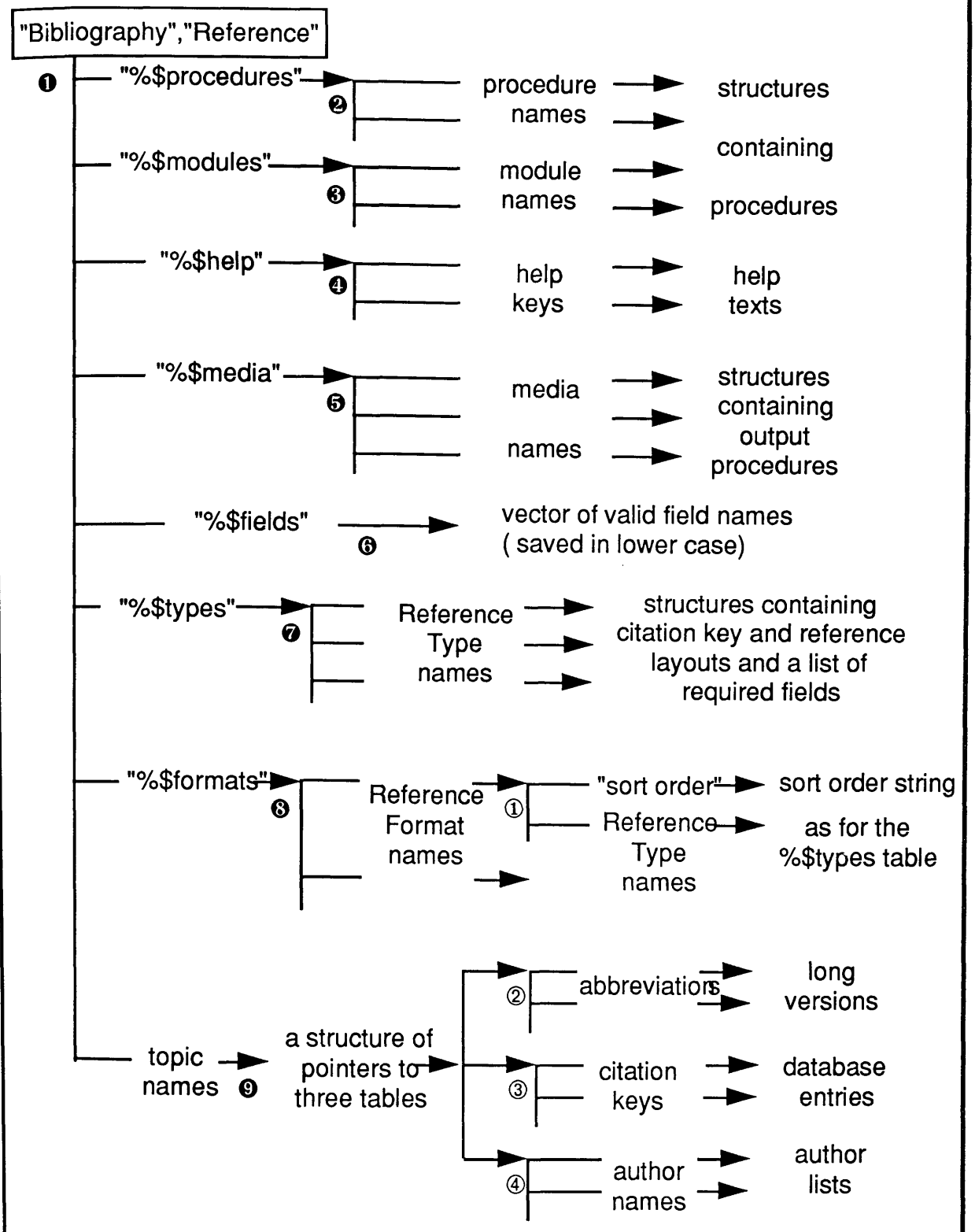


Figure 5.9 The Bibliographic Database Organisation.

❼ A table of all the Reference Types known to the system, accessed via the key "**%$types**". Each entry in this table is accessed via a Type name, e.g. "book", and contains default values for this Type of reference. The information stored is:

- a string which defines the **citation key layout;**

- another string defining the **reference layout**;

and
- a vector of the names of the **required fields**.

❽ A table of all the Reference Formats known to the system, accessed via the key "**%$formats**". Each entry is accessed via a format name, e.g. "IEEE", and points to a table (marked ①), which contains the bibliography sort order, accessed via the key "**sort order**", and an entry for each Reference Type known to this format. These entries have the same structure as those in the "%$types" table (❼). When producing the citation keys and references in the bibliography, the builder looks for its formatting information in the selected Reference Format table(①) and if the type is not there, looks for the default values in the "%$types" table (❼).

❾ The other entries in the top level table are accessed by Topic names and point to a structure containing all the data about a given Topic. This structure consists of pointers to three tables:

② a table of abbreviations which consists of entries accessed by abbreviations pointing to packaged full forms, e.g. "CJ" points to "Computer Journal" packed into a structure with a string field;

③ a table of authors which contains an entry for each author name in the set of references of this topic - the entry points to a list of the references of which he or she is an author (this is used by the browser);

and ④ a table of the references, which uses the citation key as a key for the table.

There are two main points to be observed in this structure. Firstly, the hierarchical structure of the database fits the data, which naturally subdivides into objects of different types. Secondly, the modules of the program can co-exist with the data, thus keeping all the information specific to this task together in the Persistent Store. This simplifies the programmer's conceptual view of the world. It can also be imagined that, in a large store this coupling of program with data will have performance benefits, in that all the objects in a given database may be kept "close together".

As previously mentioned, the program also makes use of the fonts database and the utilities library database. This library is also organised as described in Chapter 8. In general, an application program can expect to use "system" databases to get access to communal facilities, together with one or more "owned" databases.

5.3.2 The Software Modules.

The software is divided into six parts as described above. Here a little more detail is supplied about these six parts.

(i) The program which installs a skeletal database creates the database and provides initial values for the set of **valid fields**; the table of **Reference Types**; and the table of **Reference Formats**. It also sets up empty tables for the procedures, modules, media and help information.

(ii) The startup program must be run to initiate a session with the database. It provides the main menu and calls the high-level modules as requested by the user.

(iii) The main operations of the program, stored in the "**%$modules**" table, are called directly from the startup program. There is one program to insert each of these in the table. It is important to emphasise that it does not matter in which order the modules are written, nor whether they are written before or after the startup program, as each will compile and run separately. If the startup program is to be tested first, the modules can be represented by stubs until they are replaced by the real version. The program implementing them will effect this replacement with no extra effort concerning the startup program.

(iv) The set of low-level utilities which are specific to this program are stored in the "**%$procedures**" table. These are called by the modules or by other low-level procedures. Once again, procedures can be written before or after those which call them. It is noted here that straightforward static binding of one procedure to another is not sufficient, since the called procedure might need to be replaced. Fortunately, PS-algol permits a range of binding strategies including dynamic, static and optional forms of binding, provided that a sufficient structure is available to make the binding explicit. Chapter 8 describes the various techniques for this and the chosen strategy used for this and the other programs described in this thesis.

(v) The utilities in the standard utilities database include the Dialogue Box Package (4.1.5); the Chooser (4.1.4),; the String Editor (4.1.6); the Message Facility (4.1.2) and the More facility (also 4.1.2). These are called from the procedures in sets (iii) and (iv), but do not themselves call any procedures in the application database, since they are designed to be a stand-alone set of utilities.

(vi) There are three **sets** of procedures which provide parallel versions of a given operation - the bulk loaders, the bulk dumpers and the specialised media output packages. Each of these sets is represented as a table and for each different version there is a separate program to put it into its appropriate table. These tables are organised so that new versions of an operation can be installed for immediate use by users without any re-running of other programs. Each version is implemented and installed by a separate program and access to this table is controlled by a procedure, which calls the Chooser to select a version and then loads and uses the selected version. The immediate update of the table shows through to the user because the Chooser dynamically builds its menu.

5.3.3 The User Interface.

The principal goal of the design of the User Interface was to make it uniform. At every phase of the interaction with the system, the user initiates the same sort of action in the same sort of way. Thus, having edited one sort of object, the user will find that to edit any other sort of object will require a similar process. This should speed the familiarisation process and give the user confidence in the program.

Another unifying feature is that display and edit modules are compatible. For instance, the windows to display and edit a reference look very similar. The only apparent differences are a slightly different heading and the presence of light buttons for commands in the edit window. There is, also, a non-apparent difference - the information display items which are passive in the display window become light buttons in the edit window.

Another goal in the design of the user interface is to reduce the amount of information the user needs to provide - the more that is typed, the more errors will be made. It is felt that the user should not have to provide information that the program already has, like the names of objects, nor have to type in commands in a strict syntax. This led to the menu-dominated interface style. All operations are selected by mouse clicks over light buttons of one sort or another. Another consequence of this was the design of the Chooser (4.1.4). This tool allows the user to select objects by menu and not by a name that must be remembered.

Thus a User Interface has been created which is coherent, consistent and easy to use. Some writers have questioned over-reliance on a menu-based style, claiming that for some users or for some situations, chiefly those when the user is very familiar with the domain of the program, a command-input style is faster and less frustrating. For the present task, menus seem to be the best path - although there is a nod in the command-input direction, by providing the keyboard input for the Chooser. It has also been pointed out that the menu-based style was inappropriate for the Sort Order editor. A direct-manipulation style, in which the various ordering attributes were represented by tiles which could be "dragged" into order might very well have been easier to use and could easily have been implemented in PS-algol.

However, the main point to be made is that the presence in PS-algol of sophisticated graphics primitives allows the program designer to make choices between these options without undue cost. Given the system, the customer could rewrite the interface software if that was required and run the rest of the system intact. Indeed, it would be possible to provide a choice of editors for users with different requirements.

5.3.4 The Transaction Mechanism.

The system requires a fine-grained control over object update. That is, there is a need for changes to individual objects to be atomically reversible. In changing a particular object, the user must be able to undo the changes, without undoing changes to other objects. However, it should be recalled from Section 3.2.5 that PS-algol has a very crude notion of object update. In order to change an object in the database, its reachable value must be over-written and then **commit** must be performed. The problem is that there is no way of committing only some of the changes that have been made. Thus if changes are made directly to the database and an error is made, either the error must be accepted or all the other changes since the last commit will be lost. Calling commit after every update would get round this, but then all changes would be made immediately irreversible. It is also a limitation of PS-algol, that changes cannot be abandoned without leaving the program by performing **abort**. It would also be useful to be able to reverse a number of changes after making them.

To summarise this, atomically reversible updates available on all objects, including bulk objects, are required. For these purposes, a system of transactions has been implemented. This will now be illustrated and then discussed.

The process of editing one of the default set of Reference Types is here taken as an illustration of the nested transaction mechanism. The user must follow this procedure:

i) Select the "Edit types" option in the top level menu. This initiates the transaction, "Edit the set of Types".

ii) Select the "Edit" option of the Type Editing Menu to edit a particular Type. This starts a sub-transaction, "Edit a Type".

iii) Edit the Type .

iv) Respond to the question "Do you want to preserve your changes?". If the response is "y", the Type Editor returns a new Type object with the modified values. Otherwise, the editor returns the original object. Thus, after the Type has been modified, the user can abandon the modifications at the end of the "Edit Type" transaction if so desired. If the modifications are kept, then they are held as part of the modifications in the current "Edit the set of Types" transaction.

v) Edit more Types and when no more changes to the set of Types are to be made, respond to another "Do you want to preserve your changes?" question. This again gives the user the option of abandoning all changes done during the transaction by responding "n". If the response is "y" control returns to the initial menu and all the modifications are made to the database itself.

vi) To make the changes permanent, the user must select the Clear Up option and then select the "Commit" option to carry on, or the "Quit/Commit" option to finish.

To support this mechanism, copies of every edited object and set of objects is made. Thus in the above, at step (i), an empty table is created which will hold the set of modified or new Types. At step (ii), a new Type object is created, whose attribute values are the same as the object selected for editing. At step (iii), this new object is modified - not the original. At step (iv), either the old object or the new one is returned by the editor. If it is the new object, this is put into the table of modifications. At step (v), if the user responds "n", this table is thrown away. Otherwise, it is merged into the table of Types. Finally, at step (vi), with the call of commit, the changes are made permanent and irreversible.

This mechanism is cumbersome and has not proved popular with users. In order to make changes permanent, it is necessary to go right back to the initial menu. To continue, the user must then use the menus to return to the data that was being changed. A modification of the system is under way, in which the user has a commit button in every window. This would merge the changes in all transactions of which

the current one is a part and make them permanent. However, the real solution to this problem lies in a better design for object sharing, update and committal, which is a major research issue for the future. It has been shown in this section, however, that whatever transaction mechanisms are proposed it is likely that they can be built on top of the PS-algol primitives.

5.3.5 Object Identity.

One of the minor design issues concerned the editing of lexical identifiers. It was decided that editing an identifier **created a new object, but did not affect the old one.** This method has been chosen so that many objects of the same type and with largely the same values can be created easily. For instance, a new Reference Type can be created which is identical in most cases to an already existing one. Changing the identifier of an object and yet maintaining object identity seems to the author to be an unusual activity and one that should not be directly supported. The mechanism for identifier update is as follows.

The major objects in the database all have an identifying string associated with them. Fields, Types, Formats and Topics have names and the references themselves have a key. When the identifier of an object is changed by use of an editor, this corresponds to creating a new object. If the editor is entered with an object identified as "X", some changes are made to values within the object, the identifier is changed to "Y" and then more changes are made, **a new object identified as "Y" will be created.** This will be a copy of "X" with all the modifications made, whether before or after modifying the identifier. This edit will leave "X" **totally unchanged**. Not even the changes made before the identifier will be made on "X".

This brings up a point about identifiers and persistence. Objects in the persistent store have a unique invariant Persistent Identifier (PID) and thus have no logical need for data dependent identifiers - this is one of the selling points of persistence, with the claim that space is saved. The PID plays a similar role to the surrogates in RM/T [Codd, 1979]. Although the PID represents a sufficient mechanism for the program to keep track of objects, the user also requires a lexical reference to the object. All of the objects have some field which uniquely identifies them and this is used to provide the user with a mnemonic for the object. The author believes that, in most database applications like this one, this kind of identifier will be essential.

5.3.6 Further Work.

The system is operational as specified and its limitations have become apparent. For instance, the system would be more satisfactory if there were an option for the papers themselves to be stored in the persistent store. There is no technical reason why they should not be, but until word processing power has been added to the system, the disadvantages of having two copies of the paper probably outweigh the advantages.

Providing a page make-up system using the persistent store would give processing economy. Most runs of such a system are of iterations of the document, in which the document is only slightly perturbed. Retaining the data structures

describing the layout of the generated pages would yield economies or an accelerated WYSIWYG response.

The larger accumulations of data will warrant better retrieval tools than the browser. These should be built on the basis of current information processing techniques, including some browsing on key words. At the same time, it would be desirable for the user to be able to build up an owned set of references in an *ad hoc* manner and then produce a bibliography using the same output facilities currently used by the automatic bibliography builder. The ability given by persistence to bind new code to existing, highly structured data, held in a strongly typed form, should prove particularly helpful when adding such computationally sophisticated modules.

5.4 Conclusions.

The development of a system for the maintenance of bibliographies has been described. All of the software was developed in a matter of four man-months. The speed of development was due to programming within a persistent environment. The system was developed in an incremental fashion, using fairly small, easily debugged modules. The modules were themselves stored in the same space as the data in the form of data structures containing properly bound first-class procedures. Therefore, it was easy to re-use sections of code to perform similar tasks. It was also easy to replace partially working modules with better ones.

However, the main benefit was the ability to store new modules alongside old ones and then to generate menus to decide which module to use. For instance, initially the only bulk loader available was for Scribe files. As soon as a Refer format file was encountered, a Refer format loader was written, plugged in and was then immediately available since the Chooser permits the selection of a loader by use of a dynamically produced menu.

As the work proceeded, modules were identified which were of more general usefulness than just for this program. These modules, such as the Chooser and the String Editor, were abstracted from the bibliographic database and placed in a database which made them generally available to other applications. These modules will be re-used in later chapters.

Managing such a large set of modules threw up problems concerning the relationships between them. When PS-algol is used "cold", the way in which one module is bound to another is not explicitly available after the binding has happened. Chapter 8 describes a number of ways this binding may be made and some techniques for making the binding explicit. A major conclusion from this chapter is that trying to manage a complex application without such techniques is laborious and would become infeasible for really large-scale applications.

Another finding was that the lack of a transaction management system had a significant effect, but also that a suitable system could be built on top of the available primitives. This is a clear example of the extensibility of PS-algol. If a given feature does not exist, it can usually be supplied on top of the primitives, using the language itself. This seems to be a greatly superior environment than one in which any extension to the system requires delving down into the implementation language

(usually C) and hacking the implementation itself. Many of the features of the system, the menus, the **print** statement, etc., were written in PS-algol itself.

Finally, of course, the application shows a clear advantage of a persistent system in maintaining a single computer model of the application. Figure 5.9 shows the database structure, included in which are the structured objects of the application. The structure in which the objects are conceptualised in the program is exactly the same as the one in which they are conceptualised in the database. There is no mapping between the two and the application never had to concern itself with dissecting up an object to store it. For these reasons it is concluded that PS-algol proved a suitable implementation vehicle for the application.

Chapter 6. Building Database Systems in PS-algol.

Chapter 5 showed how database applications can be built directly in PS-algol. This chapter and the next one move up a level and describe how data modelling systems may themselves be implemented in PS-algol. The implication of this is that the functionality of data models can be added to PS-algol and so any application requiring, say, the facilities of the relational model can make use of a component programmed in the same way as the rest of the application. It is only possible to do this because PS-algol is sufficiently high-level that programming systems can be described in it.

In this chapter, two implementations of the Relational Model are discussed, before describing higher-level models in Chapter 7. The first of these is the RAQUEL system of Pedro Hepp and the other is a relational system constructed by the author and Djamel Abderrahmane which provides improved storage and retrieval methods for relational data.

6.1 A Database Architecture With Several Interfaces.

The first relational database system was implemented at the University of Edinburgh by Pedro Hepp [Hepp, 1983a, Hepp, 1983b, Norrie, 1985]. The goal of this research was the creation of a system which provided a multiplicity of user interfaces to a uniform internal data model. In the system produced by Hepp, a relation is called a table and the columns are typed - each column being of type integer, bool, string, date or time.

The provision of a number of interfaces to the same database gives data access to different classes of user. The Query Languages provided were: TABLES, a screen oriented query and update language for a relational database, similar in style to QBE [Zloof, 1977]; RAQUEL, a relational algebra language, also for querying and updating a relational database; and FQL [Buneman *et al.*, 1982]. It is envisaged that naïve users will use TABLES, which is simple to use but limited, while more sophisticated users will move on to RAQUEL or FQL. There is also a Report Generator - a document producer, which takes in commands to specify page layout, headings, etc.

6.1.1 The TABLES Interface.

The first interface provided is called TABLES. This is a QBE-like interface to the underlying relational model. The queries which can be specified are persistent objects in their own right and are made up of selects, projects and joins. A query is formulated by filling in items in skeletal tables. The commands in TABLES permit the following operations:

- select a table to use for subsequent work;
- traverse the table - commands for this manipulate a cursor (which is initially in the top left cell) and move it left, right, up or down or to the top or bottom row or to a column having a specific value in a specific row;

- bulk load some data;
- manipulate user views;
- update data - includes commands for the insertion and deletion of rows and the modification of individual values;
- define queries, see below;
- output a table or the result of a query.

As has been said, queries are formulated in terms of tables on the screen and uses a now out-of-date character addressable i/o model of interaction. For instance, the query in Figure 6.1 specifies the query which will return the two column relation as an answer to "give the staff numbers and matriculation numbers of all teachers who are over 40 and also students of the science faculty".

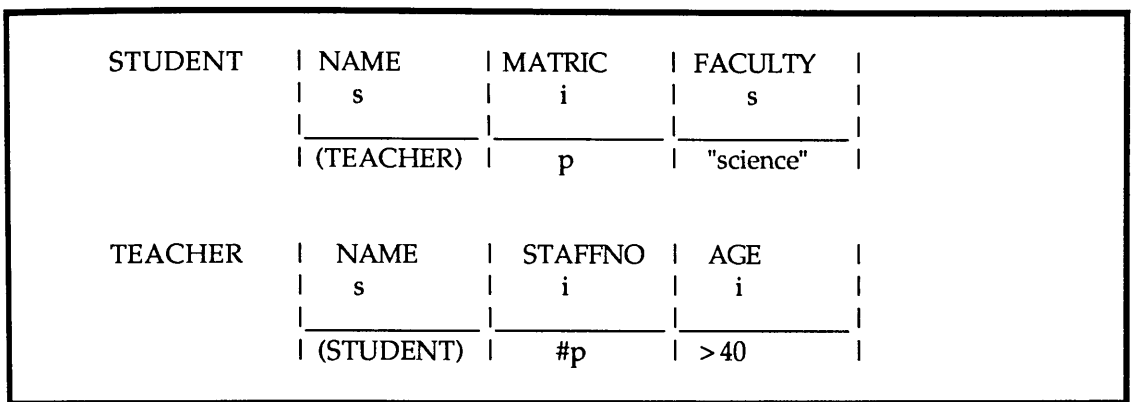


Figure 6.1 A Sample TABLES Query.

In the diagram: the "#" indicates the current cursor position; a "p" indicates include this column in the output (i.e. project); a table name in brackets indicates a join to another table; a constant value indicates a selection to determine rows in the output (the rows must have this value in this column); an expression starts with one of "=", "<", ">", "<=", ">=" or "<>" and indicates a condition for row inclusion in the output (i.e. select). Multiple rows in the same table in the query indicate disjoint alternatives.

To support these queries, the interface has the following operations:

- move the cursor up, down, left or right;
- add or delete tables to the query;
- join two tables;
- insert a "p", a constant value or an expression at the current cursor point;
- delete the item at the current cursor point.

Using TABLES, it is possible to build up complex queries which are made up of selections, projections and joins. It is envisaged that TABLES will provide a good

introduction to relational systems for novice users, who can then migrate to RAQUEL to build up more complex queries.

6.1.2 The RAQUEL Interface.

RAQUEL is a textual relational query language with far more facilities than the TABLES interface. Not only are selection, projection and join available, but a number of other functions which will now be briefly illustrated with reference to the relations given in Figure 6.1.

Projection is achieved by a command of the form:

query *STUDENTNAMES* := *STUDENT* **projected on** *NAME*

Selection is indicated as in:

query *OLDIES* := *TEACHER* **selected on** *AGE* > 40

Systematic data modification can also be done, as in

query *OLDERTEACHERS* := *TEACHER* **modified on**
if *AGE* < 65 **then** *AGE* := *AGE* + 1

Ordering of results can be done:

query *STUDENTSBYMAT* := *STUDENT* **order on** *MATRIC* = **a**

where the order can be either "a", ascending, or "d", descending.

Extending a table can be done:

query *RETIRALS* := *TEACHER extended to*
NAME, STAFFNO, AGE: RETIRAL := **if** *AGE* > 65 **then** "R" **else** ""

which is a projection followed by the creation of a new column.

Grouped column creation is the last of the unary operations, as in:

query *STUDENTSBYFAC* := *STUDENT* **grouped on**
FACULTY: TOTAL := *count*

which projects to a column for *FACULTY* and then adds a second column which uses the system function, *count*, to calculate the number of students in each faculty. Other numerical functions are *min, max, avg* and *sum*, each of which can be followed by a selecting expression. There are also two boolean functions, *all* and *any*, which return **true** if all or any of the contributing rows return **true** for a following expression.

Natural join is performed by:

query *STUDENTTEACHERS* := *STUDENT* **joined by**
NAME =NAME TEACHER

Outer join is similar:

query *ALLPEOPLE* := *STUDENT* **oj** *NAME* = *NAME TEACHER*

but here all rows from both columns are included, with columns that appear in only one column being filled out with nulls.

There are also facilities for **set union, set intersection** and **set difference.**

Using these commands, queries of arbitrary complexity can be built up and RAQUEL becomes a good tool for teaching the richness of the relational algebra.

6.1.3 Functional Query Language.

This is an implementation of Buneman's FQL [Buneman *et al.*, 1982]. In this language, all of the elements of the database are represented by functions. For instance,

Relations are represented by functions which return sequences of objects, such as !STUDENT.

Columns are selected by a dot operator (STUDENT.NAME).

Rows are built up as in ["A student", 12345, "science"].

Operators are also seen as functions. Thus [1, 2] + is a representation of 1 + 2, with the square brackets creating a tuple of integers and the following plus operator summing over it.

Using FQL, complex queries can be built up against the same database as TABLES and RAQUEL. It is interesting to compare this component of Hepp's system with the implementation of the Functional Data Model [Shipman, 1981] described in Chapter 7.

6.1.4 The Report Generator.

This tool, which will not be described in detail, permits the user to create a structure for the output from a table interactively. It includes facilities to define report and page titles, to set the page length, to define the layout characters which separate rows and columns, to provide some basic statistical information such as the averages, maxima and minima of columns, and also allows graphs and histograms to be produced.

The Report Generator was written before the graphical facilities of PS-algol were added and so reports are generated only for character devices, but the resulting reports show how summarising information can easily be derived from databases using PS-algol.

6.2 Implementation Details of the RAQUEL System.

6.2.1 Overview.

The model produced by Hepp contains three components: an Internal Conceptual Schema (ICS), containing meta-data; an Internal Data Manipulation Language (IDML); and an Internal Query Language (IQL). A modified subset of the extended relational model, RM/T [Codd, 1979] with the following architecture was proposed, illustrated in Figure 6.2.

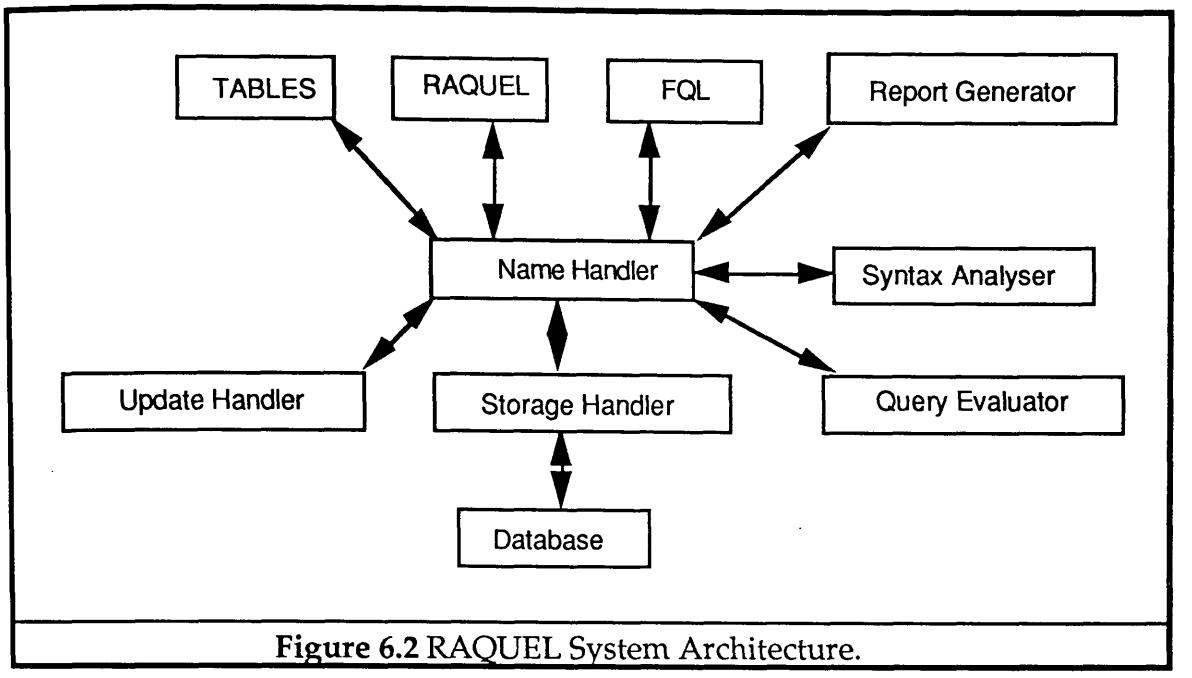


Figure 6.2 RAQUEL System Architecture.

The Query Language interfaces interact with the database only through the Name Handler, which translates names into internal identifiers. It uses the ICS to make all its checks and this itself has been organised as a set of relations, so that as with EFDM, the same procedures can be used to access meta-data and user data. The ICS starts off with two relations, one containing a list of all the relations in the system and one containing a list of attributes. User-defined relations and attributes are gradually added to these. The other components shown include the Storage Handler (SH), the Query Evaluator (QE) and the Update Handler (UH). The SH controls the creation, maintenance and deletion of relational structures, such as relations and tuples. The QE processes queries specified in the IQL and the UH ensures database consistency by monitoring update requests to detect integrity violations.

The program itself consists of a set of 43 procedures held in a single structure, together with an initiating program (which starts RAQUEL up). There is one source module to create this procedure package as a persistent object and five more to insert the evaluator procedures, the storage handler procedures, the name handler procedures, the syntax analyser and a set of utilities into this package. The procedures which provide the various user interfaces form another set of source modules.

As in the Bibliographic Database, all the program and data is stored in a single PS-algol database. The top-level table of this contains 9 entries, one to the packaged procedures and the others to further tables. In these are stored the relations, the columns, the constraints, the queries, evaluated queries, temporary relations, some

global data and the views. The latter are PS-algol tables with the same structure as the top-level table. The user switching views merely switches which table is the current one.

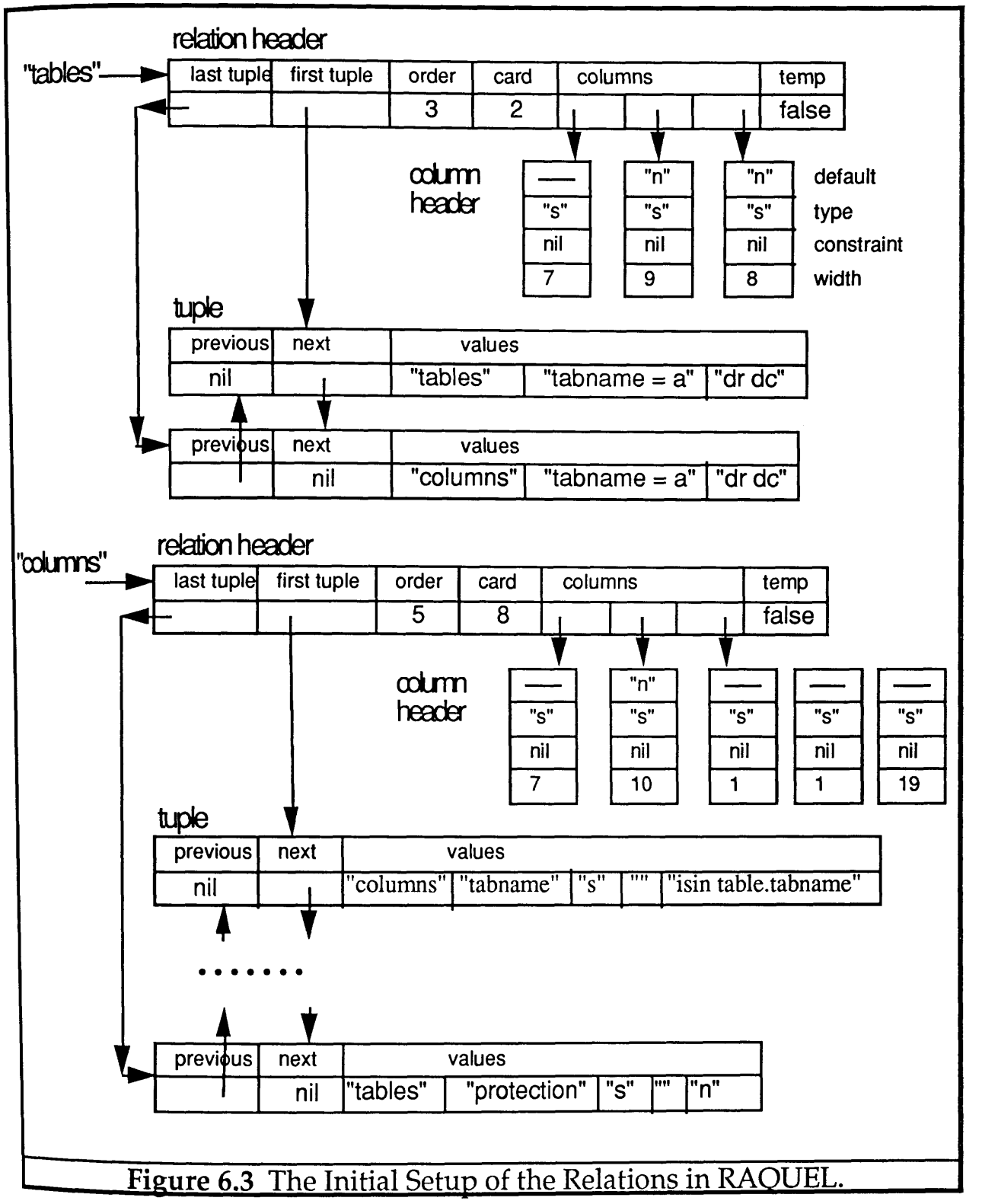


Figure 6.3 The Initial Setup of the Relations in RAQUEL.

All data and meta-data are stored in the form of **relations**. The data for a relation are stored in a doubly linked list of **tuples**, with a **relation header** containing summary information at the head of the list. Included in the summary information is some information about each attribute or column of the relation, stored in a **column header**. Figure 6.3 shows the data structure used to represent two relations, in this case

the two holding the metadata, as these are stored in the same way as ordinary data. The two relations are "tables", which holds one tuple for each relation, and "columns", which holds one tuple for each column of each relation. At start up time, the system contains only these relations, as shown in Figure 6.3. As can be seen, *tables* has 2 tuples, one for each metadata relation, while *columns* has 8, one for each metadata column.

Note that Hepp has used a crude strategy for getting a kind of polymorphism into his program - every value is represented as a string. His program relies on there being a few types of columns and he provides, for each type, a pair of translation procedures - string.to.type and type.to.string. This is a fairly inefficient method of storing data, both in terms of space and in terms of time to search and to dereference data.

6.2.2 The Benefits of PS-algol.

Pedro Hepp used an early version of PS-algol - one that did not have the graphics systems, nor first-class procedures. In his arguments for using PS-algol, Hepp puts forward many of the reasons mentioned in Chapter 3 - uniformity of approach, lack of arbitrary exceptions, relieving the programmer from concern about the physical mapping of data to store, and the simplicity of the language. However, the main benefit he found in using PS-algol is not stated directly, but is implicit in every section of his thesis: the ability to create a program incrementally. He made use of this in four ways (and also reduced compilation time by breaking down source code into smaller modules).

Firstly, he built his system incrementally. At first a very small system was implemented, with crude versions of the modules. Later, he replaced these with more sophisticated versions, using the persistent store to hold the most recent. This enabled him to develop each module separately. As the database access implicitly provided by PS-algol is based on lazy fetching from disc and strict type checking, program construction is performed as necessary by an incremental type-checked linker - the persistent system itself. It is possible for the programmer to arrange to use permanently one particular implementation of the module, or to use the latest version, or one chosen by any other algorithm. Thus the incremental construction depended on the delayed binding supplied by the **pntr** type.

Secondly, once the internal model was put into the persistent store, as many user interfaces as were required could be added, one at a time. In fact, having got the RAQUEL interface working (with all of the modification and debugging of the internal system implied by this), Hepp got the TABLES interface working "in less than a week" and the FQL interface "in approximately one week of work".

Thirdly, in making the decision on which underlying storage structures to use, he could try independently a number of different options before selecting the best one. This was done by replacing the storage handler with a number of variants and testing the resulting system for speed of access, storage requirements and ease of programming. He tested whether to represent a relation by lists or vectors and whether to represent tuples as strings, vectors of strings, vectors of pointers or as a list of pointers. His analysis led him to choose to represent his tuples as a vector of strings.

Fourthly, he used the persistent store to record patterns of usage of the various interfaces and modified them to overcome user problems. For instance, certain inelegancies in the syntax of RAQUEL queries were ironed out after examining the pattern of user errors. Furthermore, an analysis of the frequency of usage of objects in the system revealed that "a small set of columns and relations are used more frequently in query composition than the rest." Clearly this fact could have been used to provide more efficient storage and retrieval methods.

Hepp made no use of the run-time compilation system, not then available. With the advent of the run-time compiler, the analysis of usage, which was performed off-line, could be performed regularly by the system itself. For example, a dæmon, activated at times of low system usage, would carry out some analysis of the usage of each data object, refer to some normative data on usage, and, if necessary, change the storage to be more appropriate for the pattern of usage found. The user would not notice the change in the underlying storage structure, except that response times would be improved. These ideas are similar to those put forward by Stocker [Stocker, 1973], but the freedom to devise and manipulate any data structure would facilitate experiment and implementation. This idea is left undeveloped at the present time, but it is noted that a Persistent Store which can contain programs and data is an ideal implementation environment.

6.3 A Polymorphic Architecture For Relations.

In this section, another improvement due to the run-time compiler is explored. The storage structures for the data, which in RAQUEL are forced to be static, could be created dynamically, according to the nature of the data. This new internal model is called GRAPE (Glasgow Relational Adaptive Persistent Environment) [Cooper *et al.*, 1987c]. The starting point is a data storage model similar to that used by Hepp and uses the universal pointer type to provide a polymorphic storage scheme for the tuples of a relation. The interface is amended to take advantage of PS-algol's facility for producing Abstract Data Types and the storage of the tuple structures is tailored to the form of the relation using the compiler function. This exploits PS-algol's ability to implement polymorphic schemes by use of late or early binding to achieve efficient data representations.

6.3.1 A Static Internal Model for GRAPE.

After some investigation, a storage scheme for a relation was produced, which is structured as shown in a simplified form in Figure 6.4. The header for the relation consists of four fields: the relation name; a pointer to the body, which is a doubly linked list of tuples; a pointer to the primary key header (here shown to be a single column, but in general a list of columns); and a pointer to the rest of the column headers of the relation (also pointed to by the primary key). The column headers are organised into a linked list of structures each containing the column's name and a pointer to an instance of an Abstract Data Type defined on domains. In the initial scheme, each tuple consists of a vector of pointers to value containers.

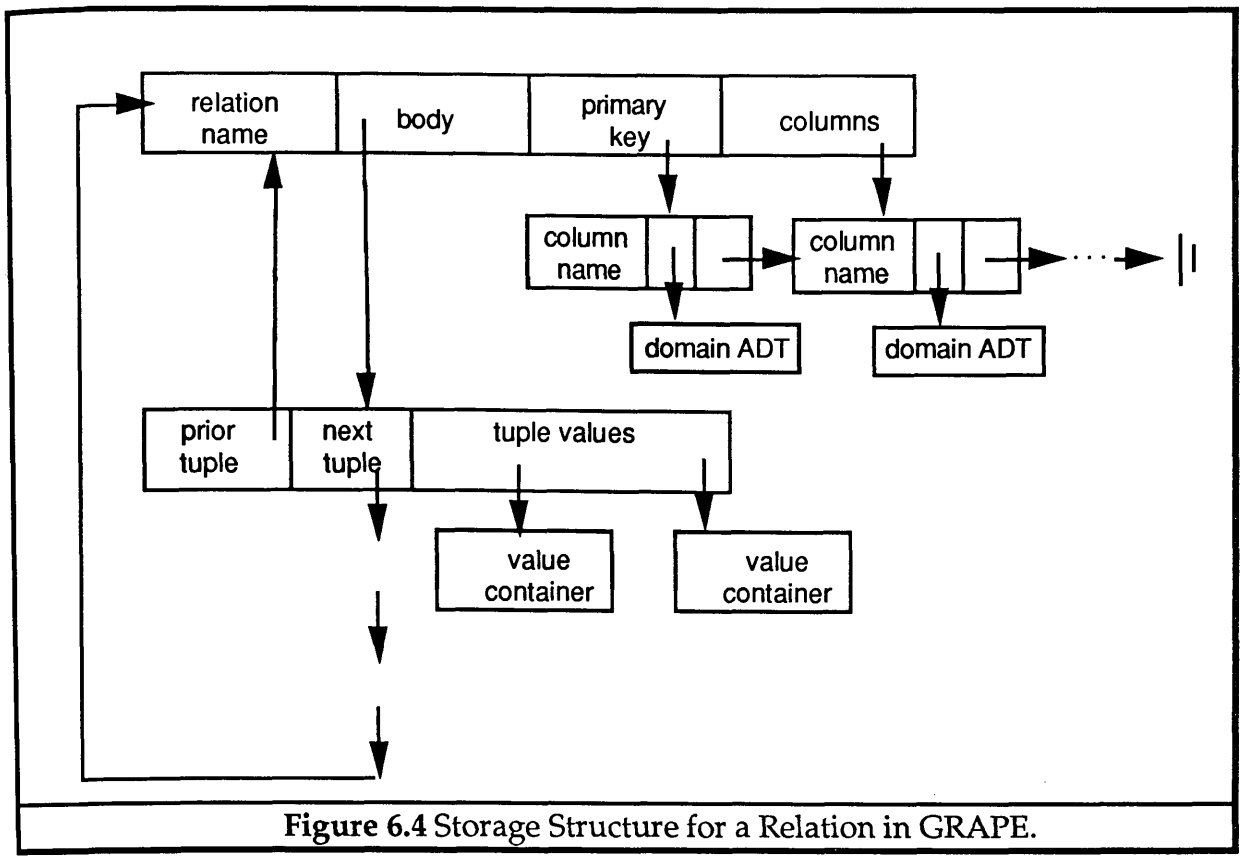


Figure 6.4 Storage Structure for a Relation in GRAPE.

The interfaces provided to both relations and domains are in the form of Abstract Data Types. Domains are represented by an ADT that contains at least the following operations:

```
proc( string -> pntr ) putDomVal        ! package a value
proc( pntr -> string ) getDomVal        ! unpack a value
proc( pntr, pntr -> bool ) compDomVal   ! compare two values
```

Domains are created by calls to a creation procedure by the user interface programs and stored in a table in the persistent store.

Relations are created using the following procedure -

MakeRel = **proc(string** *description* **-> pntr)**

This is given a description of the relation in the form of a string (containing attribute names, attribute domain types and those attributes which are used as the key) and returns a packaged set of procedures which contain all of the operations permitted on this relation, such as adding a tuple, looking up a tuple from the key, traversing the tuples, checking whether or not the relation is empty, etc. Each call of *MakeRel* binds the **same** code bodies to a new instance of data structures with the **same** definition.

Take as an example the relation

ADDRESS(**string** *name* | **int** *house*, **string** *street*)

in which the field *name* is to be used as the primary key. The construction of a

simplified polymorphic representation in PS-algol (corresponding to Figure 6.4) of the tuple "R. Cooper, 73, Bow Rd." is shown in Figure 6.5.

```
structure tuple( pntr last, next; *pntr values )      ! Relation independent tuple and
structure stringContainer( string stringValue )       !   data containers.
structure intContainer( int intValue )
let RC = tuple( ..., ..., @ 1 of pntr [  stringContainer( "R. Cooper" ),
                                         intContainer( 73 ),
                                         stringContainer( "Bow Rd." ) ] )
```

Figure 6.5 Indirect Storage Scheme for an Address

This creates an instance of the tuple structure, *RC*, consisting of pointers to the adjacent tuples in the list and a vector of pointers to the three field values. The 73 would be de-referenced by

RC(values)(2)(intValue)

which first takes the *values* field of *RC*, takes the second element of the vector and then unpacks it - thus the operation requires three levels of indirection.

A version of *MakeRel* using this storage method is shown simplified in Figure 6.6. The procedure constructs all the information it needs from *description* (looking up the domain information from the domain table). It then creates an instance of the relation structure as *ThisRel*. Then it defines operations on *ThisRel*, of which only the *AddTuple* operation is shown. This adds a new tuple to the relation from values supplied by the calling program. Finally, it packages the operation procedures as an ADT for export to the calling program. *AddTuple* merely looks in the body of the relation to find where it should put the tuple, constructs the tuple from the values input and then inserts it. Note that *MakeRel* creates a new instance of the relation structure and then binds a copy of the operation procedures to it.

This version of *MakeRel* can be written once to handle any kind of relation since all the values are stored via pointers. It achieves polymorphism by using the fact that the **pntr** type corresponds to the union of all possible structures and hence of all possible containers.

```
structure RelHead( string rname; pntr body, pkey, columns )
structure ColHead( string cname; pntr domType, nextCol )
structure tuple( pntr prior, next; *pntr values )
let MakeRel = proc( string description -> pntr )
  begin
    let RelName =                                   ! Get these from
    let PkeyName =                                  !   the description
    let PkeyType                                    !   by string
    let ColNames =                                  !   manipulation.
    let PkeyADT = s.lookup( PkeyType, DomainTable ) ! Get domain types of the
    let ColTypes = ...                              !   primary key and the
    let ColADTs = ...                               !   other columns.
    let PkeyComp = PkeyADT ( compDomVal )           ! Get an ordering procedure.

    let TheseCols := nil                            ! Make the relation header.
    for i = 1 to upb( ColNames ) do
            TheseCols := ColHead( ColNames( i ), ColADTs( i ), TheseCols )
    let ThisPkey := ColHead( PkeyName, PkeyADT, TheseCols )
    let ThisRel = RelHead( RelName; nil, ThisPkey, TheseCols )

    let AddTuple = proc( pntr PKVal; *pntr ColVals)  ! Procedure to add a tuple.
      begin
        let before:= ThisRel( body )                ! Find the tuple's place in
        while before ~= ThisRel and                 !   primary key order.
              PkeyComp (before( values )( 1 ), PKVal) do
                  before := before( next )
        let after = before( next )
        let NewTuple  := tuple( before, after, ColVals ) ! Create and insert the
        before( next ) := NewTuple                  !   new tuple.
        after( last ) := NewTuple
      end
    ......                    ! other operations of the ADT

    structure relationADT(proc( pntr,*pntr ) addTuple;
                          .... )                    ! Other procedure holders.
    relationADT( AddTuple, .... )                   ! Return this and other
  end                                               !   operations as an ADT.
```

Figure 6.6 The Simple Form of the *MakeRel* Procedure.

6.3.2 An Adaptive Internal Model for GRAPE.

In the above model, the operation to dereference the "73" field of *RC* required three levels of indirection. The new model proposes to replace the tuple structure given above with one that is more appropriate to the particular relation. It would be preferable to create *RC* by

```
AddressTuple( pntr prior, next; string name; int house; string street )
let RC = AddressTuple( "R.Cooper", 73, "Bow Rd." )
```

and de-reference the 73 by

```
RC( house )
```

but to do this, the *AddressTuple* structure must be bound into the program. When

writing the system, however, the relations the user will create are unknown and yet there must not be any restrictions on the relations that can be created. A mechanism is needed which operates dynamically (as does the original structure) and produces a structure like the above, which has improved access speed and occupies less space. The *MakeRel* procedure therefore has to use a new strategy.

```
let TupleClass = ...                                    ! Get these from the
let FieldTypes = ...                                    !   description by
let PKeyName =                                          !   string manipulation.

let MakeAddTuple =
"proc( pntr TheRel -> proc( pntr, *pntr ) )
   begin
     structure RelHead( ....      ! as above
     structure #TUPLECLASS                    ! Place holder for tuple structure.
     structure intContainer( int intValue )
     .....                                    ! more containers for string, bool, etc.
     let PkeyComp = TheRel(pkey) ( compDomVal )      ! Get an ordering procedure.

     let NewAddTuple = proc( pntr PKVal; *pntr ColVals)
       begin
         let before= ThisRel( body )                 ! Find the tuple's place in
         while before ~= ThisRel and                 !   primary key order.
               PkeyComp (before( #PKEYNAME ), PKVal) do    ! Place holder for
                   before := before( next )                ! key field name.
         let after = before( next )
         let NewTuple  := tuple( before, after, #PKEYVAL,      ! Place holders for
                          #COLLIST )                 ! derefs of Primary Key and
         before( next ) := NewTuple                  ! Column values.
         after( last ) := NewTuple
       end
     NewAddTuple
   end"

replace( MakeAddTuple, "#TUPLECLASS", TupleClass )
replace( MakeAddTuple, "#PKEYNAME", PKeyName )
replace( MakeAddTuple , "#PKEYVAL", "PKVal( "++ FieldTypes(1 ) ++ "Value) "
for i = 1 to upb( FieldTypes) - 1 do
     replaceVector( MakeAddTuple , "#COLLIST",
                    "ColVals(" ++ iformat(i) ++ ")(" ++FieldTypes(i+1)++"Value)"
endVector( "#PKVALLIST" )

structure ProcBox(proc( pntr -> proc(pntr,*pntr) ) Makeproc)
let CompiledForm = compile( MakeAddTuple,
                                  ProcBox( proc( pntr -> proc( pntr, *pntr ) ); nullproc )
let AddTuple = CompiledForm( Makeproc )( ThisRel )
```

Figure 6.7 Part of *MakeRel* Using the Run-time Compiler.

To exploit the efficiency of the second structure and still retain polymorphism, use is made of the technique introduced in the PS-algol Database Browser (Section 4.2). This is to construct all those procedures which make use of the tuple structure **at run-time**. Note that this need not be done for all of the operations of the ADT. For instance, the operation which checks whether a relation is empty can be statically determined. This only references the relation header and this has the same statically determined structure for all relations. In contrast, procedures like *AddTuple* cannot be specified in advance as they make use of the dynamically produced tuple structure.

The parts of *MakeRel* which are concerned with these procedures are rewritten to be generated automatically as shown in Figure 6.7.

In this second version, *AddTuple* cannot be directly specified, since this would not permit the specific structure of the tuples of the relation to be bound into the procedure. Adding references to an object called *ThisRel* into the string defining *AddTuple* will not make them refer to the required object, since *AddTuple* must be compiled separately. Instead a procedure-generating procedure, *MakeAddTuple*, itself constructed as a string, takes in a pointer to *ThisRel* and produces a version of *AddTuple* which operates on *ThisRel*.

MakeRel takes in a pointer to the relation and generates the string containing the tuple structure, *TupleClass*, and the vector of field types, *FieldTypes*, from the input description. Then it constructs the *MakeAddTuple* procedure as a string which varies only in the tuple structure, dereferencing the primary key value in the comparison with *before* and the line of code constructing the tuple. In this line, the values of the fields are unpacked from their containers by dereferencing the field of the container. If the field is an integer field, for instance, it is contained in an *intContainer*, whose field name is *intValue*. Conventionally the fields of a container structure are always of the form *type* ++"Value", and so can be simply created by *MakeAddTuple* . In the case of the address structure above, *MakeAddTuple* would be as shown in Figure 6.8.

```
proc( pntr TheRel -> proc( pntr, *pntr ) )
  begin
    structure RelHead( string rname; pntr body, pkey, columns )
    structure tuple( pntr prior,next; string name; int house; string street )
    structure intContainer( int intValue )
    .....                                          ! more containers for string, bool, etc.
    let PkeyComp  = TheRel(pkey) ( compDomVal )    ! Get an ordering procedure.

    let NewAddTuple = proc( pntr PKVal; *pntr ColVals)
      begin
        let before= ThisRel( body )                ! Find the tuple's place in
        while before ~= ThisRel and                !   primary key order.
              PkeyComp (before( name ), PKVal) do
                    before := before( next )
        let after = before( next )
        let NewTuple  := tuple( before, after, PKVal( stringValue),
                        ColVals( 1 )( intValue ), ColVals( 2 )( stringValue ) )
        before( next ) := NewTuple
        after( last ) := NewTuple
      end
    NewAddTuple
  end
```

Figure 6.8 *AddTuple* Generated for the *address* Structure.

MakeAddTuple is then compiled and run with *ThisRel* as its argument. It returns the appropriate *AddTuple* procedure as its result. It is at this point that the relation structure is bound to the *AddTuple* code to return a procedure which adds a tuple to this relation. This procedure is then packaged as part of the ADT returned by *MakeRel*.

6.3.3 Further Speeding Up By Memo-ising.

There are some overheads when using this method. Relation creation is a more expensive operation as it involves compilation. Although this should be offset by more efficient access to the relation once it has been created, something can be done to cut down on the need to compile every time a relation is created. Again, a technique is used which was introduced in the PS-algol Browser. This is to transform the tuple structure definition into a canonical form involving only the types of the columns. Thus the address structure would be referred to as a **string.int.string** structure and the structure defined in *MakeAddTuple* above would be:

structure *tuple*(**string** *string1*; **int** *int2*; **string** *string3*)

When the address structure is encountered, *MakeRel* refers to a table in the database to find if it has already encountered a structure keyed by "string.int.string". If it has, compiled forms of the procedure generating procedures, like *MakeAddTuple* in the example above, are retrieved from the database and re-used. Otherwise, it will compile new versions and enter them into the table, ready for any other structure, for instance:

structure *student*(**string** *sname*; **int** *snumber*; **string** *class*)

which will be mapped onto the same canonical form and will look up and use the same procedures. Further savings still are achieved by permuting the column types into a canonical order. This method of "memo-ising" a structure is supported by PS-algol tables.

6.4 Conclusions.

This chapter described two relational database systems programmed in PS-algol. An examination of Pedro Hepp's work showed how he used the persistent store to develop his system incrementally. The program was divided into manageable modules, each of which was implemented separately. This allowed him to experiment by trying different versions of modules with compatible interfaces, by dynamically binding them with the unchanged and with extant data. The cost of rebinding and reloading in a less dynamic system should not be underestimated. It also allowed him to provide a number of user interfaces which operate independently of each other. He used the persistent store to record information about system usage, an analysis of which enabled him to make improvements to it. He transformed all of his data types to strings to defer data binding. Notice also the natural way in which meta-data (the table and column information) was stored in the same way as user-defined data. The clarity of the program structure means that this was much easier to do than would normally be the case.

The GRAPE implementation has centred around attempts to increase system efficiency by using a callable version of the compiler to factor out these bindings. The "database engine" was programmed to provide a relation as an Abstract Data Type. The motive for this was an enforced and formal definition of module boundaries,

guaranteeing that module replacement was feasible. Access to a compiler at run-time has enabled the generation of the ADT using a more efficient representation as its internal model. Finally, a method was shown which reduces the cost of creating a relation by using a canonical representation of relations, which enable those with the same types to share code. This work points the way to systems which overcome the objections of Donahue [Donahue, 1987] to the use of persistent environments.

When producing database systems in conventional programming environments, the programmer faces many kinds of problem. The production of the system is significantly simplified if these problems are separated and tackled as different modules. However, in most implementation environments such separation usually involves significant complexity in inter-module communication. Thus the task of organising data on backing store may be provided by a file system, while the user interface is usually in the form of library modules. Therefore effort which should be concentrated on ensuring that the most efficient storage structure is used and providing the interface best suited to the task in hand is diffused into controlling the complexity of the inter-module interfaces.

It is also difficult in conventional environments to provide a flexible system. It is well known that different applications require different storage methods, while different interfaces suit different users' needs. However, providing more than one storage method or user interface will usually create a considerable increase in the complexity of the system.

The provision of a persistent environment allows the programmer to concentrate on the issues of basic functionality and user interface and to leave to others problems of optimising the underlying system. In particular, the programmer will not have to refer to any mechanisms extraneous to the programming language (such as file managers) to handle the storage of data. Persistence by reachability ensures that if the data are relational, storing all the data in a relation is achieved by entering a pointer to the relation's header into the backing store. All of the associated data (tuples, column names, etc.) will then be stored automatically. Furthermore, GRAPE is a demonstration that, given an efficient underlying implementation, the description of the functionality can be both very high-level and efficient.

The particular features of PS-algol which have proved of most value have been the **pntr** type and the callable compiler. By judicious use of the feature that complex objects all share a common type, which is the union of all conceivable PS-algol classes, the program has been modularised in two ways. The functions of the program are split into modules, stored in PS-algol structures and linked through **pntr** references. This permits individual modules to be replaced without the need for any compensatory work on the rest of the system. At the same time, the data have been stored in PS-algol structures and therefore programs which run over an infinite range of classes have been created. For instance, the header of a relation refers to a list of tuples and in GRAPE, the structure of these list nodes varies from relation to relation, without any effect on this header. Therefore, there is a static type check on the type of the relation header and a dynamic type check on the type of the tuple.

This dynamic type check may be performed in two different ways. It may be performed for a limited set of types by a kind of type case statement using the **is** test for class membership. Alternatively, as seen in GRAPE, it may be performed by the run-time compiler. This latter method does the dynamic type checking by binding the type

of the object into the operations required and then compiling these operations. In this way, GRAPE binds the structural information about any relation into the operations required of a relational system.

In summary, this chapter has shown that programming a DBMS in a persistent environment frees the programmer from the time-consuming issues involved in organising backing store and allows concentration on more important problems, such as a more efficient access to data and a more ergonomic user interface. It has also been shown that the programmer should be provided with a range of options on when the binding of data to the program occurs. In particular, it has been shown how the availability of run-time compilation within the implementation language permits storage schemes which are both efficient and type-secure. A further implication is that if many models can be implemented within the same system, they can be made to co-exist. In much the same way as the various bulk loaders co-exist in the BRDP, so a database system in which the user is given a choice of modelling system could be constructed.

Chapter 7. Building Data Models in PS-algol.

Chapter 6 showed how classical data models such as the Relational Model have been implemented in PS-algol. This chapter turns to higher level data modelling tools and shows how they too may be implemented. The background to this work is described in 2.2.1, but to summarise, Semantic Data Models were introduced in the mid-seventies to provide better tools for the design of databases. Initially they were introduced as conceptual off-line tools and only lately have they been available as software tools.

The reasons for this are twofold: to run these tools requires considerable computational power; while to write the software involves complex programming techniques. The former restriction is reduced by the fall in hardware costs. Consequently the computer can be used to perform more and more implementation work, leaving the software engineer to concentrate on the design. This theme unites the provision of high-level data modelling tools for database design and the provision of helpful environments within which to produce software (see the next chapter).

The complexity of programming data modelling tools, on the other hand, is only reduced by the availability of better programming tools, such as PS-algol. Thus the first implementation of the Functional Data Model was Kulkarni's EFDM, which is the first example system considered in this chapter. Written in the same early version of PS-algol as RAQUEL (see Chapter 6), Kulkarni provides a complete implementation of the FDM within which useful database applications can be written.

The second system to be considered is PSRML. This is a Requirements Modelling tool, which manipulates entities and activities involving those entities and enables models of such objects to be populated and tested. The construction of this system relies heavily on the callable compiler and the availability of first-class functions. Object-Oriented systems, such as Smalltalk, in which behaviour is tied to data instances, are too limited to enable all applications to be specified. 'Free-standing units of behaviour' greatly facilitate the production of dynamic models.

The third example is an implementation of Hull and Abiteboul's IFO model carried out by Zhenzhou Qin at the University of Glasgow. This shows how a high-level data model with a good quality user interface may be constructed using PS-algol.

Finally, the implementation of a minimal object-oriented language is described. This language, MINOO, was designed to incorporate the essential features of Object-Oriented programming, while omitting standard and well-understood programming constructs. The purpose of the implementation was to show that the hard problems of implementing such a system are tractable within the language.

The chapter concludes with some thoughts about the production of high level semantic data modelling tools and Object-Oriented systems within which efficient database applications can be constructed.

7.1 EFDM: The Extended Functional Data Model.

EFDM is an implementation of the Functional Data Model (FDM), described by Shipman [Shipman, 1981] and discussed in Section 2.2.1.4. EFDM was constructed by

Krishna Kulkarni at the University of Edinburgh [Kulkarni, 1983, Kulkarni and Atkinson, 1986, Kulkarni and Atkinson, 1987]. The FDM models data, as its name implies, using functions as the basic modelling unit for database design. An entity type is represented by a function which creates objects of that type. Attributes are modelled as functions between one entity type and another. Functions can be single-valued or multi-valued and they can be either base functions, having their data explicitly stored, or derived, having their data implicitly derived from other functions. This latter facility avoids duplicated storage of data and also provides inherent integrity constraints. The FDM comes with DAPLEX, a simple update and query language. The model forms the basis of the ADAPLEX project [Smith *et al.*, 1983].

7.1.1 The Functionality of EFDM.

EFDM is a text-based interface to the Functional Data Model. A database manipulated by the program consists of a schema with associated data, programs and queries. The interface permits the bulk loading from files of the schema or the data, or the input of data or schema by an interactive command language.

Schema definition is supplied by function definitions, such as:

declare *person*( ) ->> **entity**
declare *student*( ) ->> *person*
and **declare** *name*(*person*) -> **string**

which may appear in bulk loaded files or commands to the system.

Data input is either by specially formatted files for bulk loading or by DAPLEX commands, such as:

for a new *s* **in** *student*
    let *cname*(*s*) = 'Moyana'
    let *sname*(*s*) = 'Johns';

or **delete the** *s* **in** *student* **such that** *cnames*(*s*) = 'Moyana' **and**
        sname(*s*) = 'Johns';

Queries are entered in DAPLEX, as in:

for each *s* **in** *student* **print** *cname*(*s*), *sname*(*s*);

DAPLEX commands and queries may be stored as named objects in the database, as in:

program *printFemaleStudents* **is**
    for each *s* **in** *student* **such that** *sex*(*s*) = "F" **print** *cname*(*s*), *sname*(*s*);

such a query can then be retrieved from the database and executed as required.

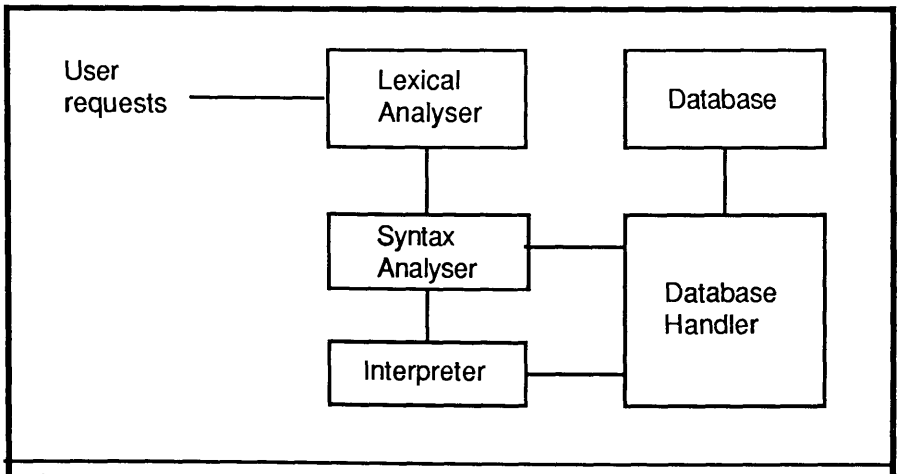


Figure 7.1 A Block Diagram of the EFDM Program Structure.

7.1.2 The Implementation.

The EFDM system consists of a single PS-algol program of some 3000 lines. The layout of the program is shown in Figure 7.1. A user request passes through the Lexical Analyser and the Syntax Analyser, emerging as a syntax tree. Schema modification requests are handled by direct calls to the Database Handler, while data update and retrieval requests pass through the Interpreter, which in turn calls the Database Handler as required. The construction of the system is much simplified by having user data and meta-data stored in the same way, thus allowing the functions of the Database Handler to be used for both.

Three kinds of data object are stored in EFDM:

- **entities;**
- **functions;**

and

- **programs,** which include updates, queries and the code for the body of derived functions.

Programs are stored as binary parse trees, which consist of triples at each node including the operation at this node and pointers to left- and right-subtrees. These trees are interpreted to derive values. The details of this are not relevant here.

The data stored about **functions** (as distinct from the values of the function) are stored in two separate structures, a function structure and an entity structure. The function structure has a number of fields, including:

- a string containing the name;
- a string containing the type of the function;
- a string containing the text specifying the function;
- an integer containing the number of arguments;
- a pointer to its associated entity structure;

and • a pointer to the values of the function, which is either:

- a program, in the case of a derived function;

or - a list of entities, in the case of a base function.

The user has no access to this structure. However, the meta-data is made available to the user in the following way. There is a system function, called *function*, which creates functions in the same way that *person* creates person objects. This system function is stored in the same way as all other functions and in particular has a list of entities which are its values. In this case the entities represent the functions which have been declared. Now the program is simplified, since adding a function to the system is the same operation as adding any other entity. Moreover, this allows functions to be declared which take functions as arguments, such as

declare *name(function)* -> **string**

which returns the name of a function. Eight such functions are automatically inserted into the database when the system is started up. The data for these functions are stored in exactly the same way as for the base functions defined on students, that is, the values are associated with the function entity. For instance, there will be an entity associated with the function *grade* and the entry in its function value vector corresponding to "name" will be a pointer to the string "grade". Therefore, the user can list all the functions in the schema by the query

for each *f* **in** *function* **print** *name(function)*

Entities are stored in lists with the other objects produced by the same entity creating function. Each entity is an instance of the same PS-algol structure which has four fields:

- a pointer to the function which generated it;
- a pointer to the next entity in the list of objects of this type;
- a pointer to a super-type entity (if there is one);

and • a vector of pointers to attribute values.

To illustrate this, Figure 7.2 shows part of the database after the following declarations have been made:

declare *person*() ->> **entity**
declare *name(person)* -> **string**
declare *sex(person)* -> **string**
declare *student*() -> *person*
declare *matric(student)* -> **integer**
declare *courses(student)* ->> **string**

and then two students Bill, male, numbered 503, taking courses "IS1" and "CS2" and Jim, male, 504, taking only "IS1" are introduced.

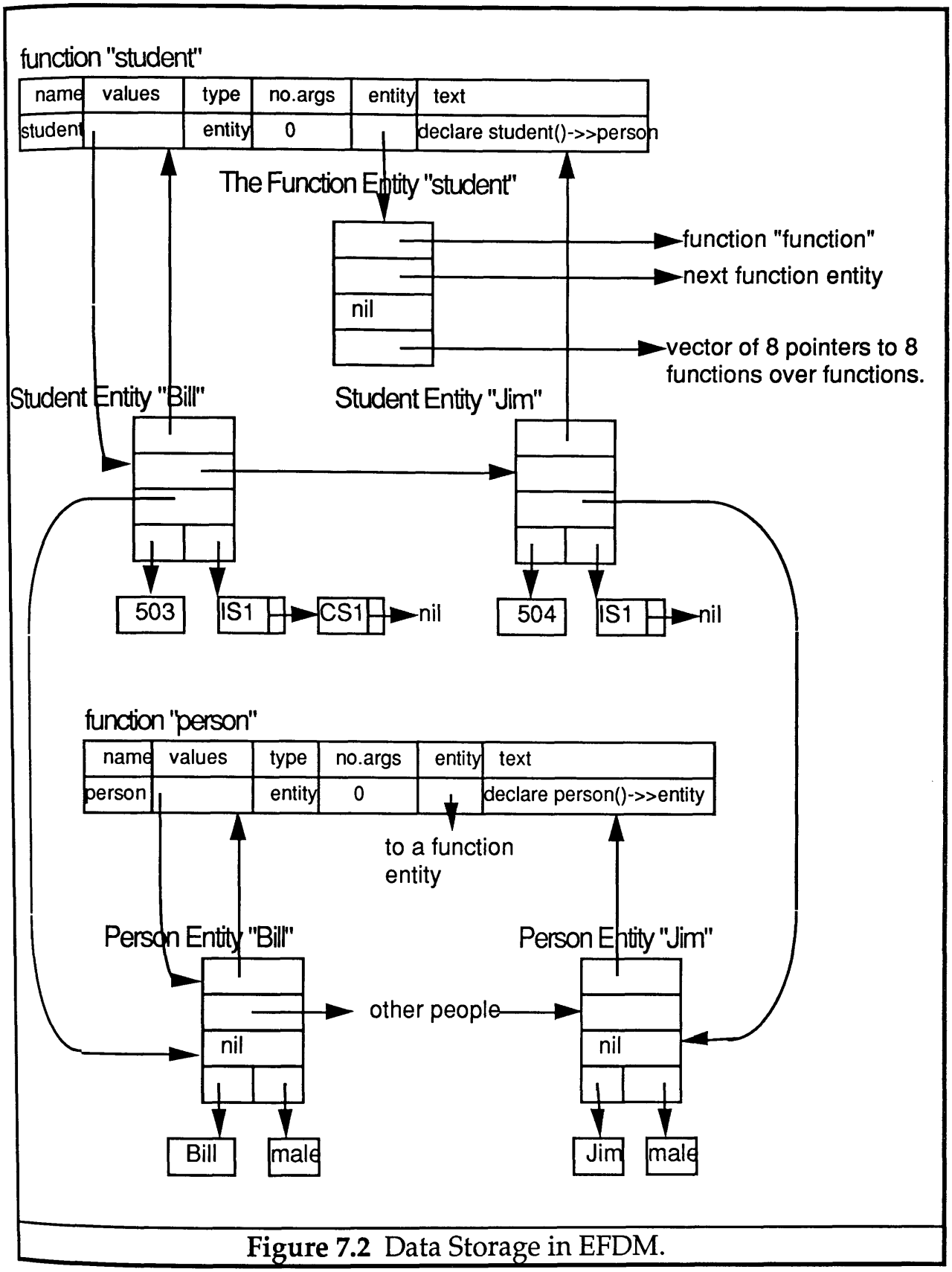


Figure 7.2 Data Storage in EFDM.

The figure shows two function objects and four entity objects, two for each student - one being his *student* entity and one being his associated *person* entity. All of the relevant information is accessible from each object in the system. Thus, from each function, the list of associated entities is accessible. From each entity, the

associated function and other entities of the same type are accessible, as are the base values of attributes (functions defined on this type).

The values may be packaged in various structures - there is one for instance for packaging integers, another for strings, as in GRAPE. The universal pointer type of PS-algol allows references to any of these to be held within a structure of the same type (a vector of pointers) and further allows values of multi-valued functions to be held in the same way. For instance, *name* is a single-valued function whose result is a string, so the pointer to a name will point to a string container. The function, *courses*, on the other hand, is multi-valued, and the pointer to the courses points to a list of values. Thus, a totally polymorphic structure has been imposed on the data and the bulk of the program can handle entities without needing to know which type they are.

Notice that, unlike many DBMS structures, no identifier is stored with the entity. The pointer to the entity is unique and consistent and may be used as the internal identifier for the entity. Wherever the data for the entity actually reside, they will always be referred to by the same pointer value. The only time in writing such a system in which it is necessary to provide such an identifier is when the user requires such an identifier for such tasks as indexing. This was the case for the BRDP, as argued in Section 5.3.5. EFDM only permits scanning over object sets or content-based access and has no need for user identifiers.

The values of multi-argument functions are stored slightly differently, but also as a list of values, each value having associated with it a list of its arguments and a pointer to the result value. Thus, if the following base function is declared:

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

then the data that Moyana Johns got an "A" for course "IS1" is stored as shown in Figure 7.3. The value pointer out of the function structure for *grade* points to a special structure containing three fields:

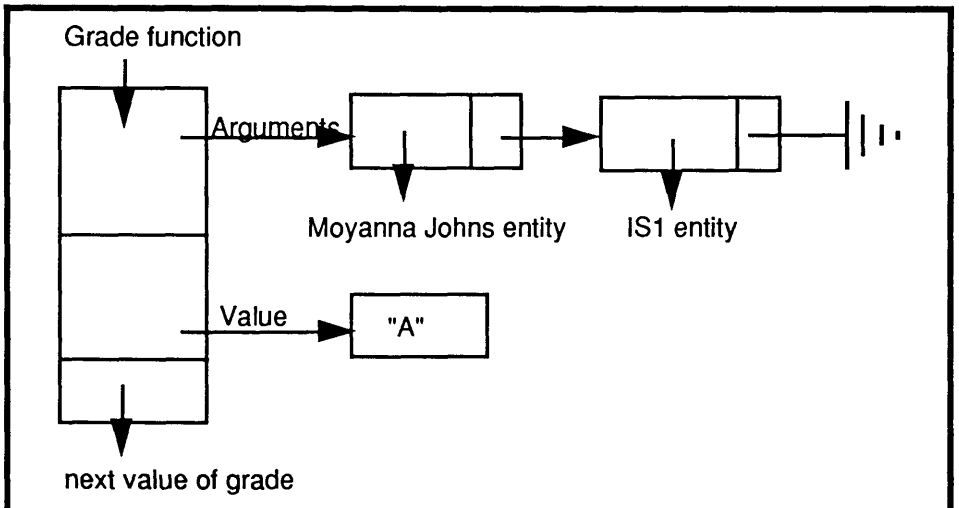


Figure 7.3 Multi Argument Function Storage.

- a pointer to a list of the argument values for one instance of the *grade* function;

- a pointer to the associated value of *grade* for those arguments;

- a pointer to the next instance.

All of this data is stored in the same database. None of the program is stored in the database as it has been written as a single unit, with none of the modularisation described in the next chapter being used. The top-level table of the database points to a number of views, of which only "global" is available initially. Each view is a subsidiary table and this table contains pointers to all of the functions, queries and programs defined in this view - these are all jumbled up together in the same table, which may be inefficient and might confuse maintenance programmers. All data is accessed via these function objects and programs.

7.1.3 The Benefits of PS-algol.

Kulkarni's initial attempt at implementation used the PASCAL language. However, this required interfacing the system to a low-level data management system and when the first version of PS-algol became available, he re-implemented EFDM entirely in PS-algol. There was a reduction in the amount of source code to about a third compared with the earlier PASCAL version. Among the benefits identified by Kulkarni were:

- the organisation of data movement being handled by the system;
- the reduction in data misuse due to type security;
- the ability to organise the data in a uniform way through PS-algol's universal pointer type;
- and an increase in speed of access to database items due to efficient heap management.

The construction of the system is much simplified by having user data and meta-data stored in the same way, thus allowing the facilities of the database handler to be used for both. The fact that function values are referred to via pointer fields, whether they are single base values explicitly stored in container structures, multiple base values stored in a list of values, or derived values stored as a program for retrieving them, greatly simplifies the programming. Kulkarni was in fact able to write a polymorphic system by making his system interpretive.

Kulkarni could have made yet more gains by using two more facilities offered by the PS-algol system. Firstly, the program as it stands is a single unit of about 3000 lines of code. PS-algol offers the ability to break the program into small modules, compile them separately and store them in the database. This means that the program could be developed incrementally, with consequent savings in compilation time and debugging time. Secondly, the code for queries, programs and derived functions is stored as a parsed tree and is then executed by the interpreter. This is an example of deferred binding. The speed of the system is reduced by this. Using the callable compiler (not available in the version of PS-algol he used), EFDM could factor out the binding by compiling the code instead. It could transform the tree into a PS-algol program and then compile it and store it in a form which would have a much faster execution speed.

7.2 A Requirements Modelling Tool.

The history of Computer Science may be viewed as the development of methodologies which enable human beings to specify complex commands to machines with increasing informality and greater levels of abstraction. In the field of Software Engineering, the **Requirements Model** has been described as the first in the series of models with which an informal specification of requirements is transformed into a machine executable form. A desirable development, therefore, would be the production of software which enables Requirements Models to be specified and then executed. This has proved difficult to achieve in conventional programming systems, but with the appearance of Persistent Programming Languages, many of the difficulties, which are arbitrary in nature, disappear.

This section describes the first steps in an attempt to produce an executable form of Greenspan's Requirements Modeling Language (RML) [Greenspan, 1984; Greenspan *et al.*, 1986], introduced briefly in Section 2.2.5.1, in PS-algol. The program, PSRML [Cooper and Atkinson, 1988], provides a graphical interface to a system in which Entities and Activities can be specified. It is planned to extend the program in a variety of ways.

7.2.1 Requirements Modelling and PS-algol.

In his thesis, Greenspan stated that what was needed for Requirements Modelling was a formal language for specifying "a model that reflects the content and structure of the application world" [Greenspan, 1984]. He further stated a number of principles to be adopted when designing such a language:

- it should be **object-centred** - that is, the elements of the language should be objects representing the concepts and entities of the modelled world;
- it should provide mechanisms of **abstraction** to assist the management of complex data;
- of particular usefulness are the well-known abstraction mechanisms -

  aggregation - the creation of entities out of their component parts;

  classification - the collection of similar objects into sets;

  and **generalisation** - the organisation of classes of objects into ISA hierarchies;
- the language should be capable of expressing **assertions, entities** and **activities;**
- there should be **uniformity** in the way in which objects within the language are treated;
- the language should be **formally** expressed;

and
- stress is also put on the value of "box-and-arrow" languages to provide a **visual** bridge between the mental model and the formal one.

To implement such a language has, in the past, proved to be extremely difficult using conventional programming languages. The introduction of Persistent Programming Languages takes a significant step in reducing these difficulties, since the facilities of PS-algol fit well with Greenspan's principles.

As has been stated already, if it is not an "object-oriented" language, PS-algol is certainly a language which orients the programmer towards handling objects.

The abstraction mechanisms are simply implemented in PS-algol as follows:
PS-algol structures are an aggregating facility;
sets can be implemented in a number of ways, by vectors, tables or linked lists, for example;
generalisation hierarchies can be linked by pointer fields as discussed in section 7.5.3.

It is also clear that PS-algol can model all three object types:
structure instances could be a good model for entities;
procedures could be used to model activities;
and procedures with a boolean result could model assertions.

The data-type completeness, orthogonal persistence and presence of graphical types naturally leads to a uniformity of approach.

and The graphical types also facilitate the provision of a "box-and-arrow" language alongside the formal language. Several experiments at the University of Glasgow are in progress, for instance, on the construction of graphical interfaces to standard database systems.

The conclusion is that PS-algol provides a fitting language in which to implement such a system. The RML language used as a basis will now be described and then the implementation details will follow.

7.2.2 The Language RML and Some Descendants.

Requirements Modeling Language (RML) was developed as part of the TAXIS project at the University of Toronto by Sol Greenspan. It is fully specified in his thesis report [Greenspan, 1984], but its essentials are now described.

RML attempts to model three kinds of object: the **entity**, the **activity** and the **assertion**. Classes of each of these are defined by a syntax which makes the three abstraction mechanisms immediately apparent. For instance, a new class of **entity** is introduced by a specification in which the class it is in, the aggregated properties and the classes supertype(s) are explicitly stated as in Figure 7.4. This means that the class of children is in the power set of people, that the class is a subclass of *PERSON*, has *reading_age* as an intrinsic part, has an associated *guardian* entity and the constraint that its guardian is over 21 years of age.

```
CHILD in PERSON_CLASS isa PERSON with
    part reading_age: AGE_VALUE
    association guardian: PERSON
    invariant guardian•age > 21
```

Figure 7.4 An RML Entity Type Definition.

Similarly, **activities** have components and supertypes and are grouped together, as is shown in Figure 7.5, in which the "arguments", "local variables" and "results" of the activity are specified, followed by assertions representing pre- and post-conditions. Finally, the body of the activity is described as an **unordered** set of "calls" to other activities, with values of their parameters being supplied.

```
ADMIT_PATIENT in ACTIVITY_CLASS with
    input p: PERSON
    control     w: HOSPITAL_WARD
                phys: PHYSICIAN
    output pt: PATIENT
    precondition arrvl: ARRIVAL( who:p )
    postcondition admitted?: IN_HOSPITAL( who:p )
    part  check-id: CHECK_ID(p)
                put: CHOOSE_WARD( w, phys )
```

Figure 7.5 An RML Activity Definition.

Finally, **assertions** may be specified, an example being given as Figure 7.6, in which the assertion is presented as a list of arguments followed by an unordered set of sub-specifications, which together constitute the assertion.

```
IN_HOSPITAL in ASSERTION_CLASS with
    argument p: PERSON
    part  patient?:  IN( p, PATIENT )
                present?:  PHYSICALLY_PRESENT( who:p)
```

Figure 7.6 An RML Assertion Definition.

In RML, a **model** consists of a set of definitions of these objects and Greenspan's thesis describes how models are developed by use of the diagrammatic language, SADT [Ross, 1977]. Greenspan lays out plans to provide a consistency checker for models, but nowhere does he set out any thoughts on how to provide an "executable" version. This would be useful in that the behaviour of a model could be examined under various input conditions.

As part of the Alvey project to develop a semi-intelligent IPSE, attention was focussed on Greenspan's ideas, extended to a process model by adding a new object class, the **rôle**, which may be viewed as an entity whose subparts include activities. Again, this was tied to a diagrammatic language, this time via the RAD - "Rôle Activity Diagram".

In order to create a version they could use, the me-too project then created a language which is essentially an executable subset of RML, called Teeny [Bennet and Rowles, 1986]. Teeny is an interactive interpreter, which understands definitions like those given in Figures 7.7 and 7.8.

```
( CHILD
     IN (PERSON_CLASS)
     ISA (PERSON)
     WITH (    ( part ( (reading_age AGE_VALUE) ) )
               ( assoc ( (guardian PERSON ) ) ) ) )
```

Figure 7.7 A Teeny Entity Type Definition.

```
( ADMIT_PATIENT
     IN (ACTIVITY_CLASS)
     ISA (ACTIVITY)
     WITH (    ( input ( ( p PERSON ) )
               ( control ((w HOSPITAL_WARD)
                         (phys PHYSICIAN) ) )
               ( output ( (pt PATIENT ) ) )
               ( precond ( (arrvl ARRIVAL( p to 'who ) ) ) )
               ( postcond ( (admitted? IN_HOSPITAL( 'p who ) ) ) )
               ( apart ( (check-id (exec 'CHECK_ID(makeff( 'p who ) )
                         ( put ( exec 'CHOOSE_WARD( makeff(w ward) )
                         (makeff ('phys doc)) ) ) ) ) ) ) ) ) )
```

Figure 7.8 A Teeny Activity Definition.

Entities can then be instantiated, for instance by:

(inst *'peter* **(makeset** *'CHILD*) 12 **nil)**

which would create a child object, called peter, whose reading age was 12. Activities can then be executed by for instance:

```
(exec 'ADMIT_PATIENT    (makeff( 'peter 'who )
                        ( 'hisward 'ward )
                        ('hisdoc 'doc )        ) )
```

Notice that Teeny has introduced "system" activities, which essentially form a basis on top of which other activities can be defined, including activities to create objects and execute other activities. In order to execute an activity, the sub-activities are queued and executed as their pre-conditions become satisfied. Clearly the syntax of this language leaves a lot to be desired, although this is understandable given the speed with which the language was implemented. The traditional command-line form interface can also be improved upon.

7.2.3 PSRML: The Goals.

PSRML is a version of RML written in PS-algol. It aims to provide a fully working environment within which models may be developed. It will provide:

- a complete well-defined language for the specification of models;
- a graphical interface for the construction of models;
- commands to examine the behaviour of models by instantiating entity classes and executing activities;
- the graphical display of models as they are executed.

It is envisaged that a system with these features would be a powerful tool for the development of software systems. Given the description of PS-algol above, there seems no intrinsic reason why an elegant version of such a system could not be written. There follows a description of a first pass in the development of such a system, describing first of all the PSRML language, then the user interface and finally some implementation details.

7.2.4 PSRML: The Language.

At present, PSRML supports only activities and entities. A full description of PSRML Syntax is given in [Cooper and Atkinson, 1988] and will be illustrated here with examples. Figure 7.9 exemplifies the specification for entity classes. PSRML permits attributes to be specified to be mandatory or not, whereas RML distinguishes intrinsic parts and associated entities. The types of the attributes can either be one of the PS-algol scalar types - **bool, int** or **string** - or a previously defined entity type. Note that throughout PSRML, all elements encountered must already have been defined, except that self-recursive definitions are permitted. This is a weakness of the language that will eventually be removed, since mutually defined types are essential to many designs.

```
entity CHILD isa PERSON with
    required guardian : PERSON
    optional readingAge : int
    modifier changeReadingAge;
```

Figure 7.9 A PSRML Entity Type Definition.

An activity definition is illustrated in Figure 7.10, in which the specification of "variables" has been left essentially as in RML, but the syntax for specifying the body of the activity (which corresponds to *part* in RML) is slightly changed. Each element of the body is a partially specified activation of some other activity. That is, each element of the body consists of:

- an identifier;
- the name of some activity which is either a **base activity** or has been already specified;
- an opening bracket;
- an optional list of parameter values separated by commas which are either literal values or parameter names from which a value will be extracted at "run-time";

- also optionally, an arrow followed by a list of parameter names, which are to receive the output values from the called activity;
- a closing bracket.

```
activity ADMITPATIENT with
    input p: PERSON
    control     w: HOSPITAL_WARD,
                phys: PHYSICIAN
    output pt: PATIENT
    body checkID: CHECK_ID( p ),
                assignDoc: pick( "DOCTOR" -> phys ),
                assignWard: CHOOSEWARD( phys -> w ),
                makePatient: make( "PATIENT", p, phys, w -> pt );
```

Figure 7.10 A PSRML Activity Definition.

Two points need to be noted. Firstly, the parameters have been passed in the specified order for the activity. Secondly, a set of base activities had to be provided in order to give the executing system a bottom. In the example, the two base activities *pick* and *make* are used. The former, given the name of an entity type, provides a menu of all instances of the type for the user to choose between. *make* also takes in a class name together with a list of values for the attributes of the object and returns an instantiation of the class, with the given attribute values. Two other base activities which have so far been provided are: *makenull*, which given the name of a class, returns an instance of the class, with its field values set to null; and *changeField*, which, given an object, the name of an attribute of such an object and a new value for that attribute, sets the field to the value provided.

Clearly this language is still too limited for serious work. Extensions to create a more powerful language are described later.

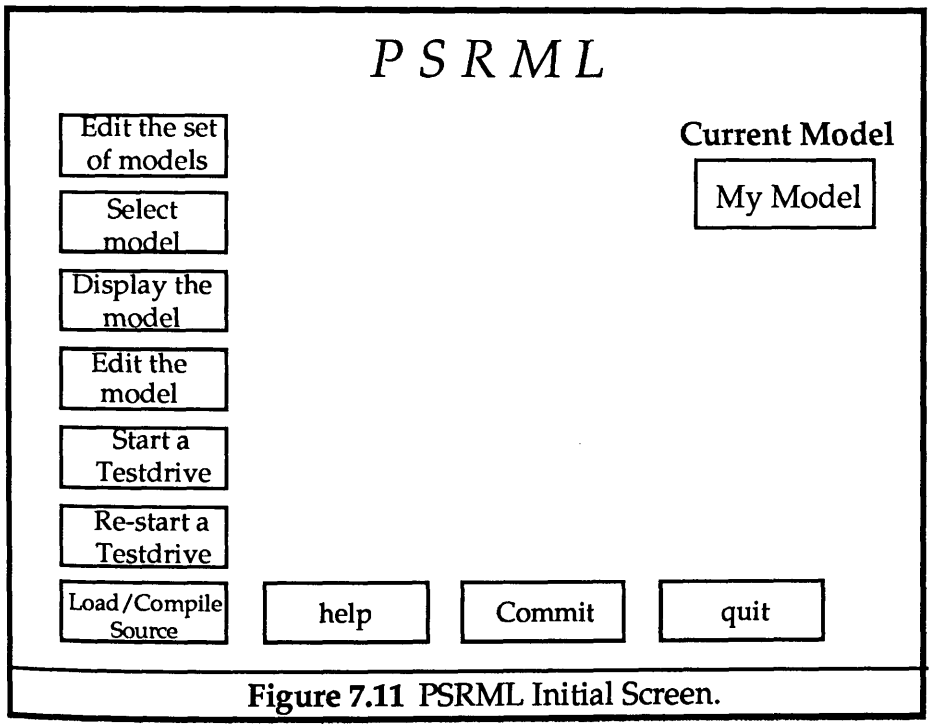


Figure 7.11 PSRML Initial Screen.

7.2.5 PSRML: The User Interface.

PSRML is provided as a menu-based system, re-using many of the same modules as the Bibliographic Reference Database System described in Chapter 5. One of the main factors which accelerated PSRML development was the availability of a re-usable set of modules, planned as part of the BRDP development. The program starts with a menu display as shown in Figure 7.11.

The seven boxes on the left-hand side of the screen are light buttons which provide the following functions:

Edit the set of models: activate a sub-menu in which the user is allowed to view, add to and delete from the list of models, and to display or edit a particular model.

Select model: select, from the set of models, one to be currently active.

Display the model: display the currently selected model.

Edit the model: edit the currently selected model.

Start a Testdrive: initiate an instantiation of the current model.

Re-start a Testdrive: re-enter an instantiation of the current model.

Load / compile source: load a file from backing store and submit it to the compiler, storing compiled objects in the currently active model.

The display module shows a list of all the entity types, activities and "testdrives" which have been specified in the displayed model. Further work will implement the ability to select elements in these lists to view further details. The editor module, when implemented, will provide guided editing of parts of the model in a format compatible with the display module. Again, Chapter 5 describes how such a feature has been implemented in the BRDS system. The source loader is a compromise with a non-persistent world in that it allows bulk-loading of a model from the file store.

The concept of the "testdrive" is that of a mixture of instantiation and execution of the current model. Starting or restarting a testdrive brings up Figure 7.12, which largely obscures the initial display. Within the test drive, five more options become available:

make an entity: The program requests by menu the type of entity which is to be instantiated and then requests, via a simple string editor, an identifier for the new object. Then, by recursively calling a system-supplied set of input procedures, the system requests values for the attributes of the new object. The new object is inserted into the set of objects of the given type. If the entity has supertypes, then values for the inherited properties will also be requested and appropriate structures will be inserted into the sets of objects of those supertypes.

list the entities: A new window is opened which lists all the types of entities together with all instances of them.

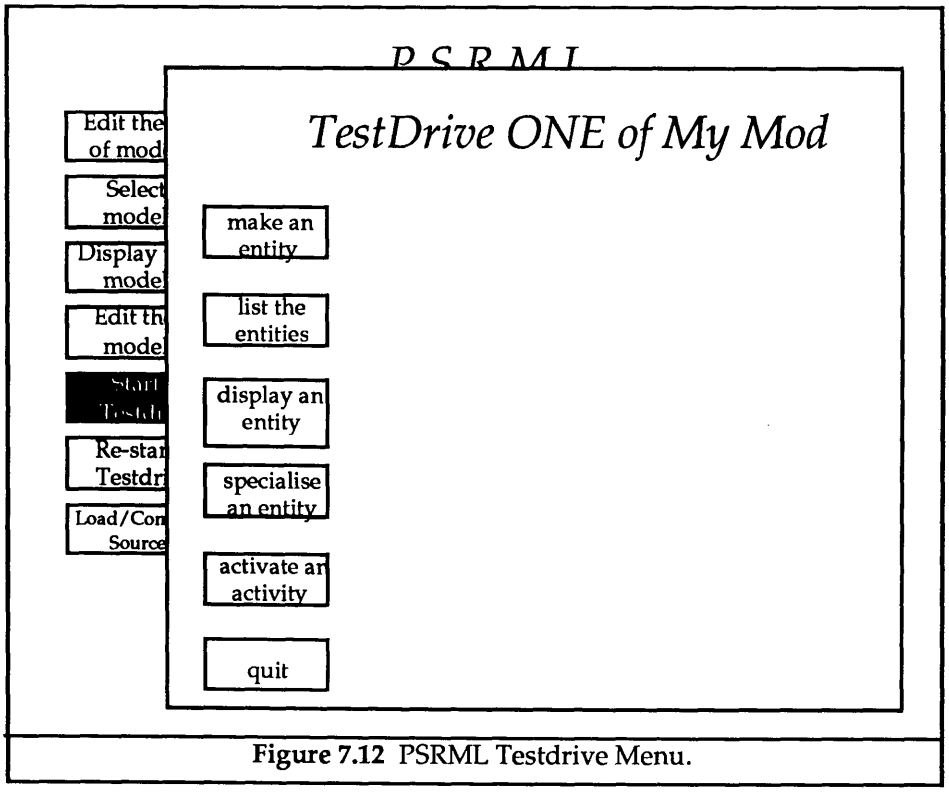


Figure 7.12 PSRML Testdrive Menu.

specialise an entity: Having selected an object of a given type, a menu of all entity types which are sub-types of the current one is given The user selects one of these sub-types and all of the property values required to specialise the current object so that it is of the new type are requested. Note that the converse, generalising an entity, has not been implemented - the inheritance hierarchy has been implemented in only one direction. This again might be seen as a restriction which can later be removed if desirable. There should be no significant difficulty in achieving this.

activate an activity: Values for the input parameters are sought and a symbol table is set up with these values in it. Then all of the sub-activities are put into a queue. Each sub-activity is examined to check if all of its input parameters can be evaluated (that is are they literals or values already on the symbol table). If so, the sub-activity is started and all of its sub-activities are put onto the queue. Sub-activities are recursively queued until a base activity is encountered, whereupon this base activity is executed. When a sub-activity completes, its results are returned to its parent. The parent then updates its symbol table from the values returned. Non-base activities are completed when all of their sub-activities complete. Eventually, either all of the original activity's sub-activities are completed or no more can be initiated - the latter leading to an error message. When an activity is successfully completed, the user is asked for identifiers for each of the activity's results. These are then stored in their appropriate sets in the database. This whole process should eventually be animated for the user's benefit.

display an entity: Again, a type of entity is requested, followed by the identifier for the object to be displayed. Both are requested by menu, following the general principle of preventing the user from making a syntactic error (in this case by having to type in the identifier again). The menus appear as shown in Figure 7.13.

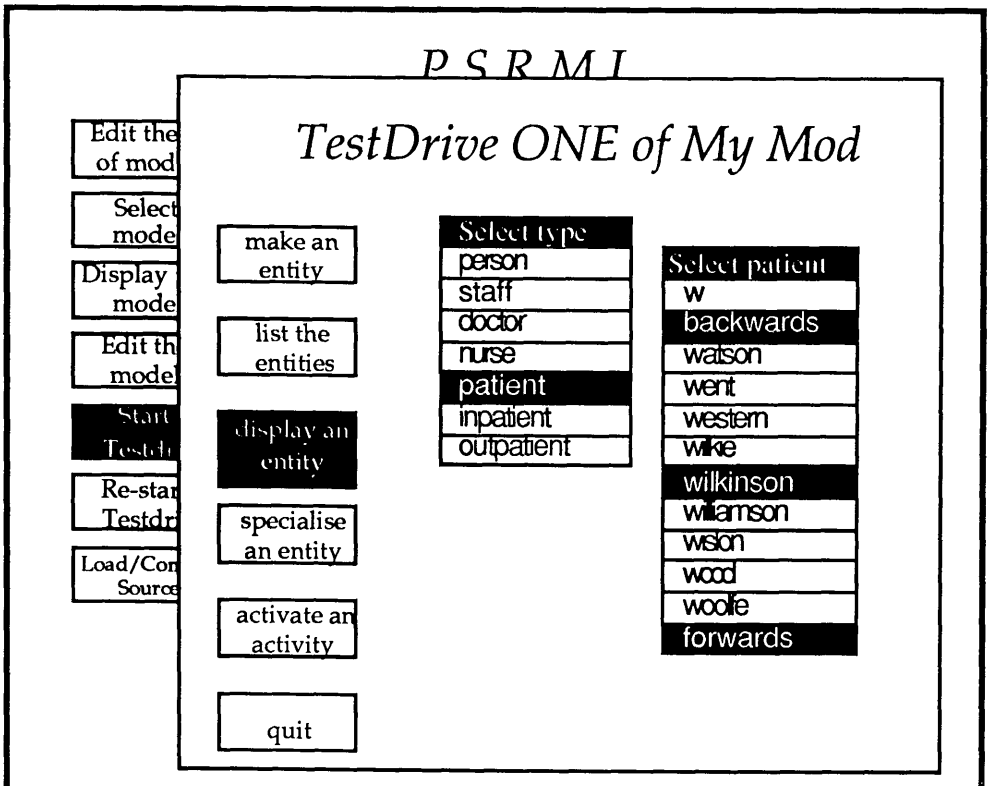


Figure 7.13 PSRML Object Selection.

The object is shown as in the diagram, with all the information for a patient, together with all the specific information inherited from any supertype. Note that if the object, *PAT1*, had been requested from the *PERSON* class, its properties which are relevant to the *PATIENT* class would **not** have been displayed. Figure 7.14 shows the information displayed.

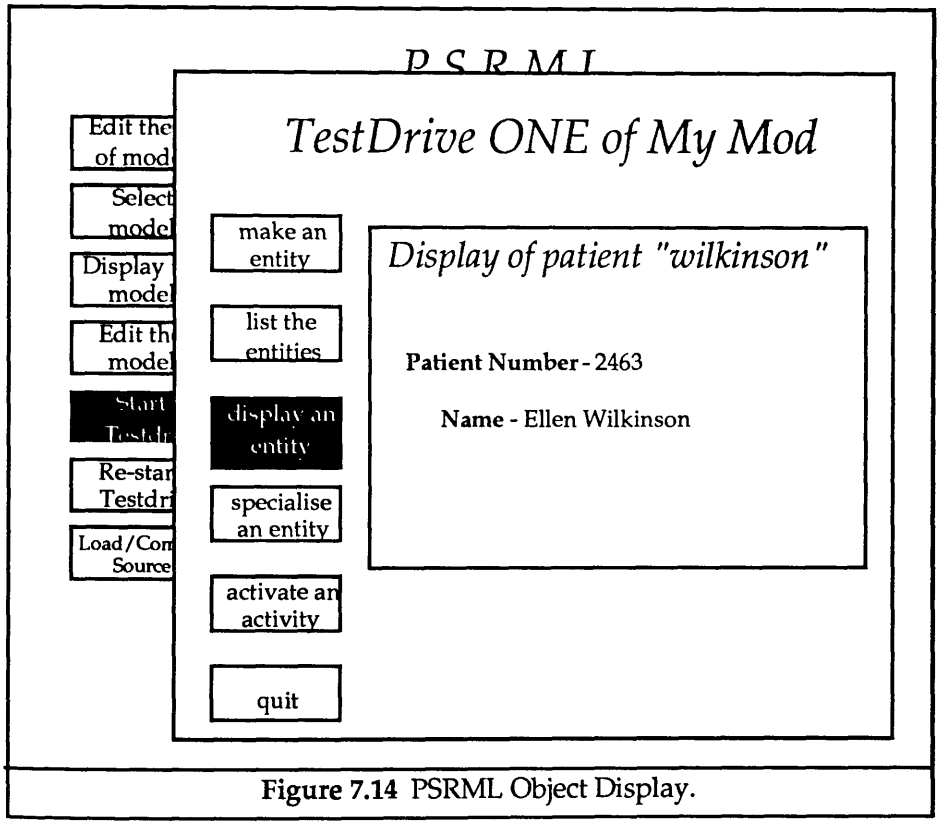


Figure 7.14 PSRML Object Display.

7.2.6 The Implementation of Entities

The implementation makes full use of the persistence mechanism of PS-algol to store the software, the schemas and the instances. The structure of the database is given as Figure 7.15. The top level table contains four system entries and then one entry for each model defined. The four system entries contain: the base activities; i/o procedures for the base types; various software components organised in the manner described in Chapter 8; and help information. Each contains a table from which hang three other tables, one each for the entities, activities and testdrives of the model.

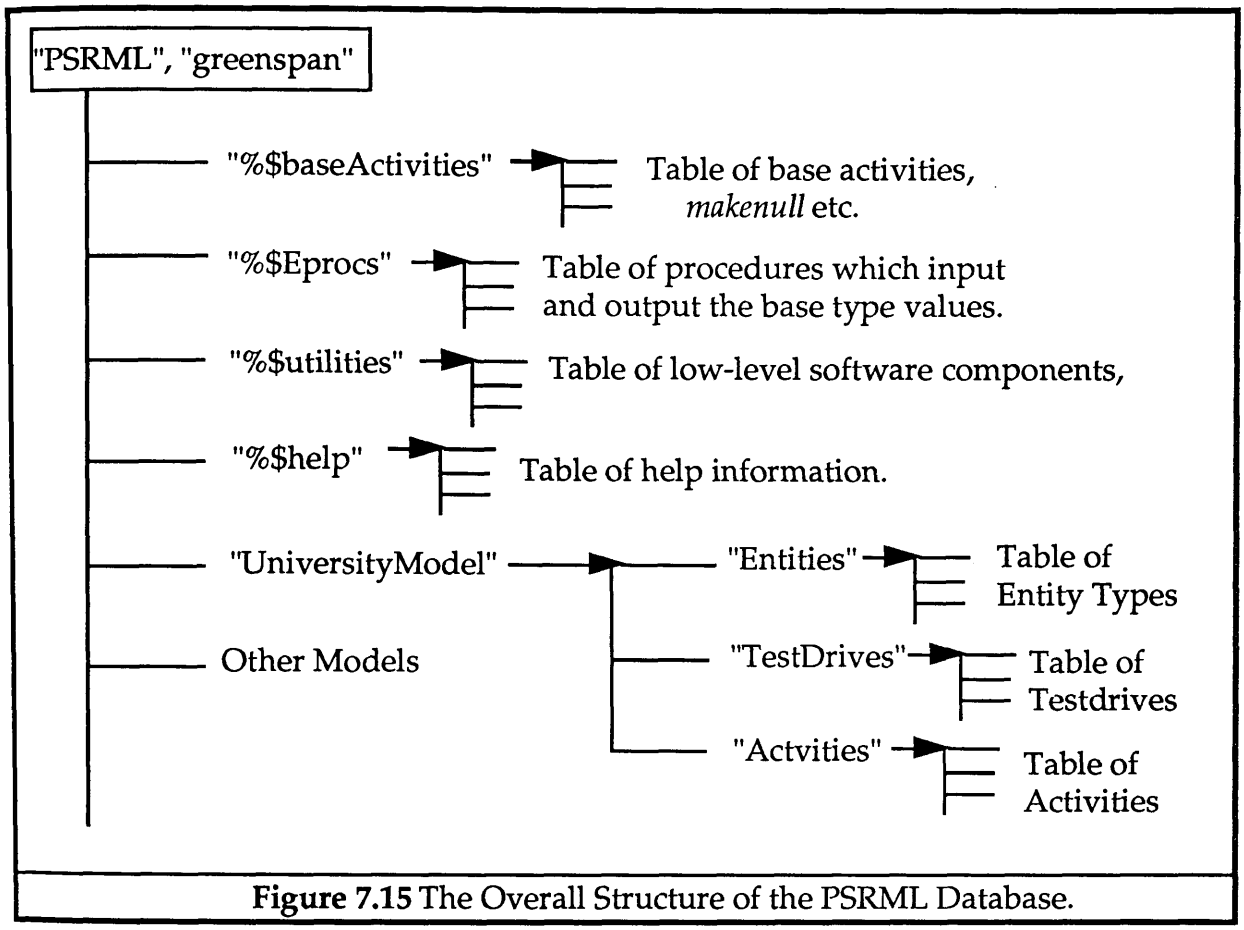


Figure 7.15 The Overall Structure of the PSRML Database.

In implementing the entities, full use is made of the callable compiler to simplify the instantiating program. The implementation uses the same type of technique as has been used in the GRAPE system (see section 6.3). The problem to be overcome is the implementation of polymorphic base operations for the entities. For instance, there is a need for a display procedure which can operate over any kind of object and yet allow the objects to be stored in an efficient structure.

When compiling an entity type definition in PSRML, PS-algol builds a structure and enters it int the *Entities* table. The structure of the Entities and Testdrives part of the database is illustrated in Figure 7.16, for the following example:

entity *PERSON* **with required** *cname, sname*: **string;**
entity *STAFF* **isa** *PERSON* **with required** *staffNo*: **int;**
entity *STUDENT* **isa** *PERSON* **with required** *matricNo*: **int;**
entity *TUTORIAL* **with required** *tutor*: *STAFF*, *tutee*: *STUDENT*;

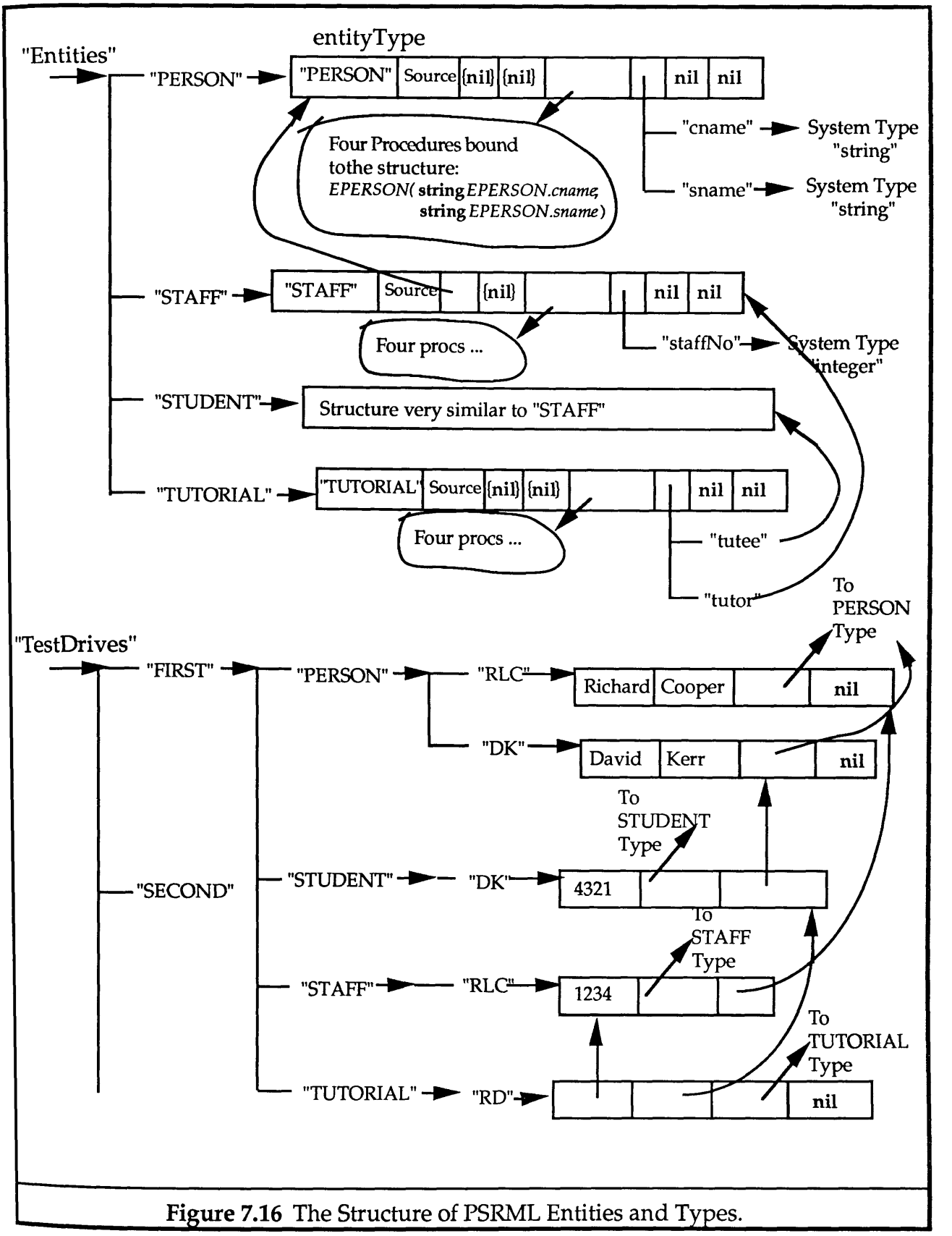


Figure 7.16 The Structure of PSRML Entities and Types.

Each entity type is represented by a structure containing a number of fields, including

- the type name;
- the definition in PSRML source language - so that it may later be edited;
- vectors of pointers to sub- and super-types;
- pointers to lists of required, optional and modifier fields;

and • a group of four procedures which are automatically built to be efficient for the given entity type.

These four procedures form an Abstract Data Type of the operations allowable on an entity. At present, these are:

makenull - returns an empty instance of the class;
makeuser - returns an instance of the class, requesting from the user values for all the fields (including inherited fields);
printer - prints the values of the fields (including inherited fields);
changeField - changes a field of an entity to a given value.

The procedures operate on structures holding entity instances, which are tailored to be efficient representations for this type. For instance, a person would be represented by

structure *EPERSON*(**string** *PERSON.cname*; **string** *PERSON.sname*;
pntr *PERSON.type*; ***pntr** *PERSON.inherits*)

in which all of the information required to construct this structure definition is derived from the type definition by string manipulation. The last two fields are common to all such structures and are omitted from Figure 7.16 for brevity. They point to the type of entity and vectors of inherited information respectively.

```
let makenull = proc( pntr TD, IP -> pntr )
  begin
    structure ECHILD( int CHILD.readingAge; pntr CHILD.guardian;
                      pntr CHILD.type *pntr  CHILD.inherits)
    structure TYPE( string TypeName; ....; *pntr SuperTypes;
                      proc( pntr,pntr->pntr) TYPE.makenull; .... )
    let superVec = TD( SuperTypes)
    let inheritVec = vector lwb(superVec ):: upb( superVec ) of nil
    for i = lwb(superVec )to upb( superVec ) do
                  inheritVec( i) := superVec( i )( TYPE.makenull )( superVec( i ), IP)
    ECHILD( 0, nil, TD, InheritVec )
  end

let changeField = proc( pntr ENT; string Fname; pntr Fvalue )
   begin
    structure ECHILD( int CHILD.readingAge; pntr CHILD.guardian
                      pntr CHILD.type *pntr  CHILD.inherits )
    structure TYPE( string TypeName; ....; *pntr SuperTypes;
                      proc( pntr,pntr->pntr) TYPE.changeField; .... )
    structure intBox( int intValue ) !  ditto for string and bool
    let superVec = ENT( CHILD.type ) (SuperTypes)
    case Fname of
          "readingAge":   ENT( CHILD.readingAge ) := Fvalue( intValue )
          "guardian":     ENT( CHILD.guardian ) := Fvalue
          default:        for i = lwb(superVec )to upb( superVec ) do
       superVec( i )( TYPE.changeField )( ENT(  CHILD.inherits )( i ), Fname, Fvalue )
  end
```

Figure 7.17 Sample Constructed *makenull* and *changeField* Procedures.

The procedures then embed this structure definition into code which constitutes the required algorithm, again by string manipulation. For instance, the *makenull* and *changeField* procedures for the entity class *CHILD* are as shown in

Figure 7.17. The *makenull* procedure gives default values for the two direct fields, *readingAge* and *guardian* and then recursively calls the versions of *makenull* tailored for the supertypes to create null sets for the inherited information.

Note how these procedures are polymorphic over any entity type, using the **pntr** type to provide polymorphism. For instance, the *changeField* procedure has three arguments: a **pntr** to an entity, a string field name and another **pntr** to the new value. The last mentioned will either be an entity in its own right or a canonically packaged base type value. The meat of the procedure is then a case statement, which contains one branch for each field of the structure. If that field is a base type field, the resulting code unpackages the value and puts it into the field. If the field was an entity, the assignment is a direct assignment of **pntr** references. If the field was neither of the two fields specified for a child, any inherited data are searched next by a call to the version of *changeField* tailored to the supertypes.

Similarly, the *makeuser* and *printer* procedures are constructed to make use of the tailored *ECHILD* structure, but are slightly more complex. They have to make calls to two system procedures one of which gets values from the user while the other prints objects of the base types. All four procedures are, of course, constructed using the callable compiler technology seen in the descriptions of the browser (section 3.2), the whole object operations (section 3.3) and GRAPE (section 6.3).

Referring back to Figure 7.16, the lower half of the figure shows a testdrive in which three entities have been introduced: a staff member called "Richard Cooper", number, 1234; a student "David Kerr", with matriculation number, 4321; and a tutorial linking the two. All instances are stored in tables for each type. Note that two entity objects have been created for the student and the staff member, linked by an inheritance pointer. Data are available only from higher up the inheritance chain. Following "DK" from *STUDENT*, all three pieces of information can be reached, while following "DK" from *PERSON*, the matriculation number is inaccessible.

7.2.7 The Implementation of Activities.

Contrary to expectation, the implementation of activities makes no use of the callable compiler. This is essentially because all activities are implemented in the same structure with no tailoring of structure to activity. Each activity is implemented by a structure with the following fields:

- the activity name;
- the definition in PSRML source language;
- vectors of pointers to sub- and super-activities;
- a pointer to the body, a list of sub-activities, used to keep a record of which sub-activities have been completed;
- and pointers to lists of formal input, control and output parameters (these lists are simply specified as a structure with two string fields, containing the field name and type, and two pointer fields to the value and the next parameter on the list).

Each sub-activity is represented by a structure with:

- the sub-activity identifier;
- a pointer to the activity concerned, either user-defined or a base activity;

- a list of the actual input parameters, each containing the parameter name, its type and a value which is either a literal or a variable name;
- a similar list of the actual output parameters, except that all values will be names of variables;
- a progress record, currently either **true** (i.e. done) **or false;**

and
- a pointer to the next sub-activity.

Figure 7.18 shows the activity table after the following activities have been defined:

```
activity SetStNum with
        input stffmem: STAFF, newnum: int;
        body  DoSet: changefield( "staffNo", stffmem,  newnum );

activity pkstStaff5 with
        control aStaff: STAFF
        body  DoPick: pick( "STAFF" -> aStaff )
              DoSet5: SetStNum ( aStaff, 5 );
```

which change the staff number of a staff member, where both the staff member and the new number are input; and instruct the user to pick a staff member and set their staff number to 5.

The process of activating an activity consists of calling the interior procedure which runs all the sub-activities of an activity, setting the input list to **nil**, and then taking any outputs returned from the activity, requesting identifiers from the user and storing them in the persistent store. The interior procedure works as follows:

❶ Take any inputs and put them into the symbol table.

❷ Set all the progress records of all the sub-activities to **false**.

❸ Circle round the set of sub-activities until all have been completed. For each uncompleted sub-activity do the following:
  ① For each required input, either it is a literal, which is the value, or the symbol table is checked to find a value. If any value is unavailable, the sub-activity is not started.
  ② If the sub-activity is a base activity, call it with the supplied inputs augmented by some system information, such as the model and testdrive names.
  ③ If the sub-activity is user-defined, activate it with a recursive call to the interior procedure.
  ④ When the sub-activity completes, add any outputs to the symbol table.

❹ At the end of each cycle of the sub-activities, either:
  there are more sub-activities to complete and some completed this time round - in this case the cycle is repeated;
  there are more sub-activities to complete but none completed this time - the activity will never complete and so an error is signalled;
or
  all the sub-activities are completed.

❺ In the last case, all the required outputs are built into a list and returned to the caller.

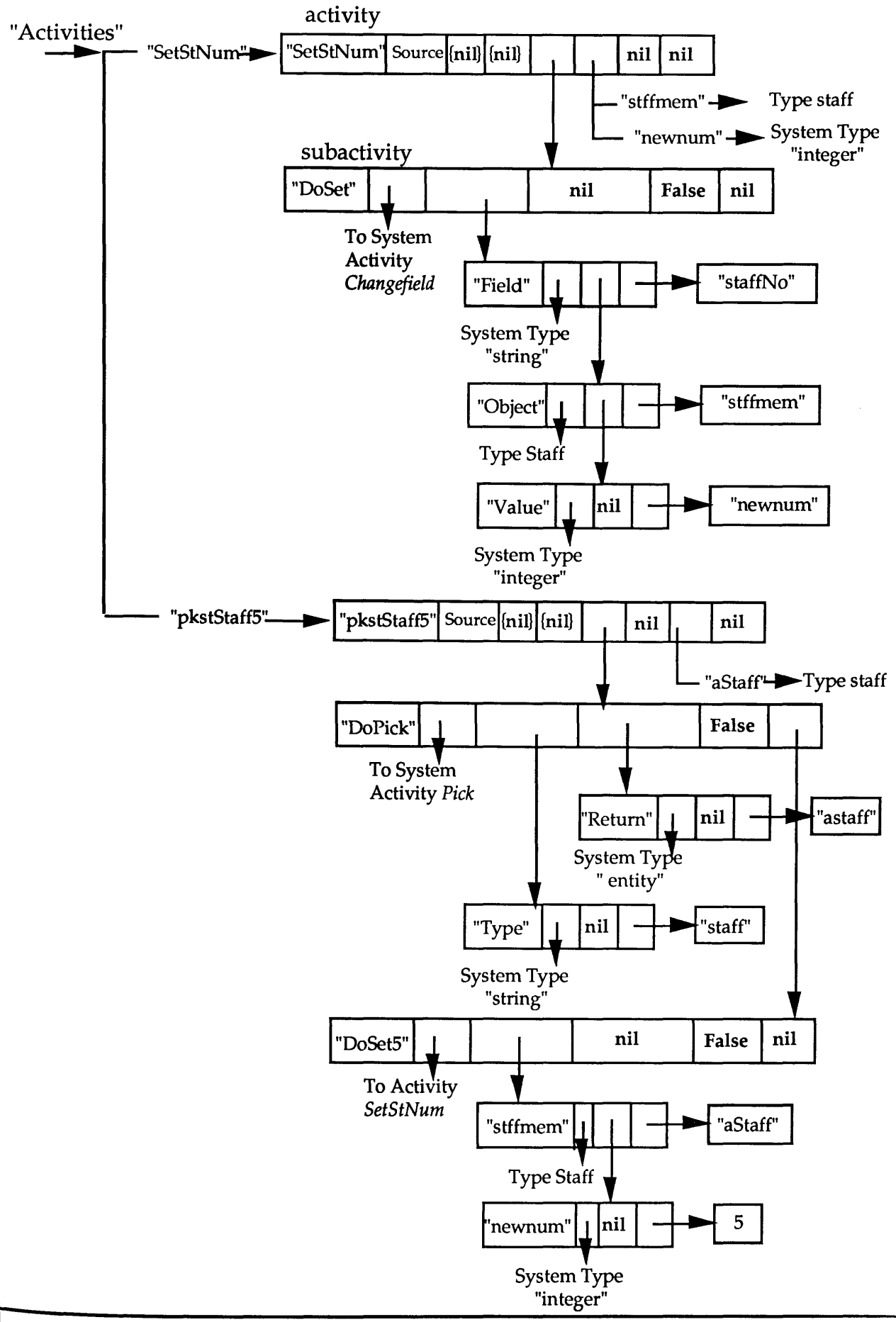


Figure 7.18 The Structure of PSRML Activities.

7.2.8 PSRML Conclusions.

It has been possible to implement a minimal Requirements Modelling System in PS-algol. The programming effort for this task was less than one man-fortnight as the implementation language was so suited to the task in hand. The reasons for this speed may be listed as follows:

- PS-algol structures permit the modelling of complicated objects such as activities with considerable freedom.

- The PS-algol **pntr** type permits the creation of polymorphic code bound to no specific data structure and yet typed within the language.

- The callable compiler further permits this code to be dynamically bound to an infinite variety of structures at run-time.

- First-class procedures greatly simplify the code which executes activities as this code circles a set of unexecuted sub-activities, executing them as soon as possible until no more are left. This code makes use of the ability to store procedures which represent the sub-activities, retrieve them and execute them as appropriate.

- The orthogonal persistence, when coupled with first-class procedures, facilitates the construction of user interface modules in the BRDP, which could be directly re-used in PSRML.

- Furthermore, access to these procedures is simplified by use of the methodology for module control outlined in Section 8.2. No doubt, it would have been furthered aided by use of a version control system such as that presented in Section 8.3.

Further discussion of the implementation strategy is given in Section 7.5, in which several issues concerning the implementation of these systems are discussed.

There has been insufficient experimentation to see to what degree inconsistent models can be created in the system, but it would seem that the system can be made proof against such misuse. It is clear that the system is reasonably powerful and easy to use. PSRML code compiles much more quickly than would have been expected. However, many extensions need to be made.

Firstly, the syntax might be considerably expanded. Some developments in the near future could be:

- a return to Greenspan's *part* and *association* separation of entity properties;
- the ability to specify the constancy of properties;
- the ability to specify sets of objects;
- the inclusion of the whole PS-algol type system as base types, thus allowing the use of pictures as properties, for instance;
- the removal of the need for the definitions to be ordered with only backwards references - this is a real restriction in a requirements modelling language, much more so than in a programming language.

Secondly, **assertions** should be added, together with some notion of **processes**. The value of free standing assertions is doubtful, but as *condition* fields of activities and entities, they are very useful. They can be simply implemented in PS-algol as procedures which return a boolean result.

Finally, an animated display of the activation of an activity might prove informative and helpful.

7.3 The Implementation of the IFO Data Model.

Under the supervision of the author, Mr Zhenzhou Qin created an implementation of Hull and Abiteboul's IFO Model [Abiteboul and Hull, 1987]. This provides a graphical language with which to manipulate a Semantic Data Model and is further described in section 2.2.1.6. The model distinguishes various sorts of relationship between entities and is designed for the analysis of update semantics in a high order data model [Cooper and Qin, 1989].

7.3.1 The PS-algol Interface to IFO.

The program provides a graphical interface to both schema and data manipulation. These are well matched and, as in PSRML, make use of the user interface modules created for the BRDP. Interaction starts with an initial menu with options to edit the schema, entering a graphical schema manipulation mode, and to edit the data, entering a graphical data manipulation mode. These different modes will now be described.

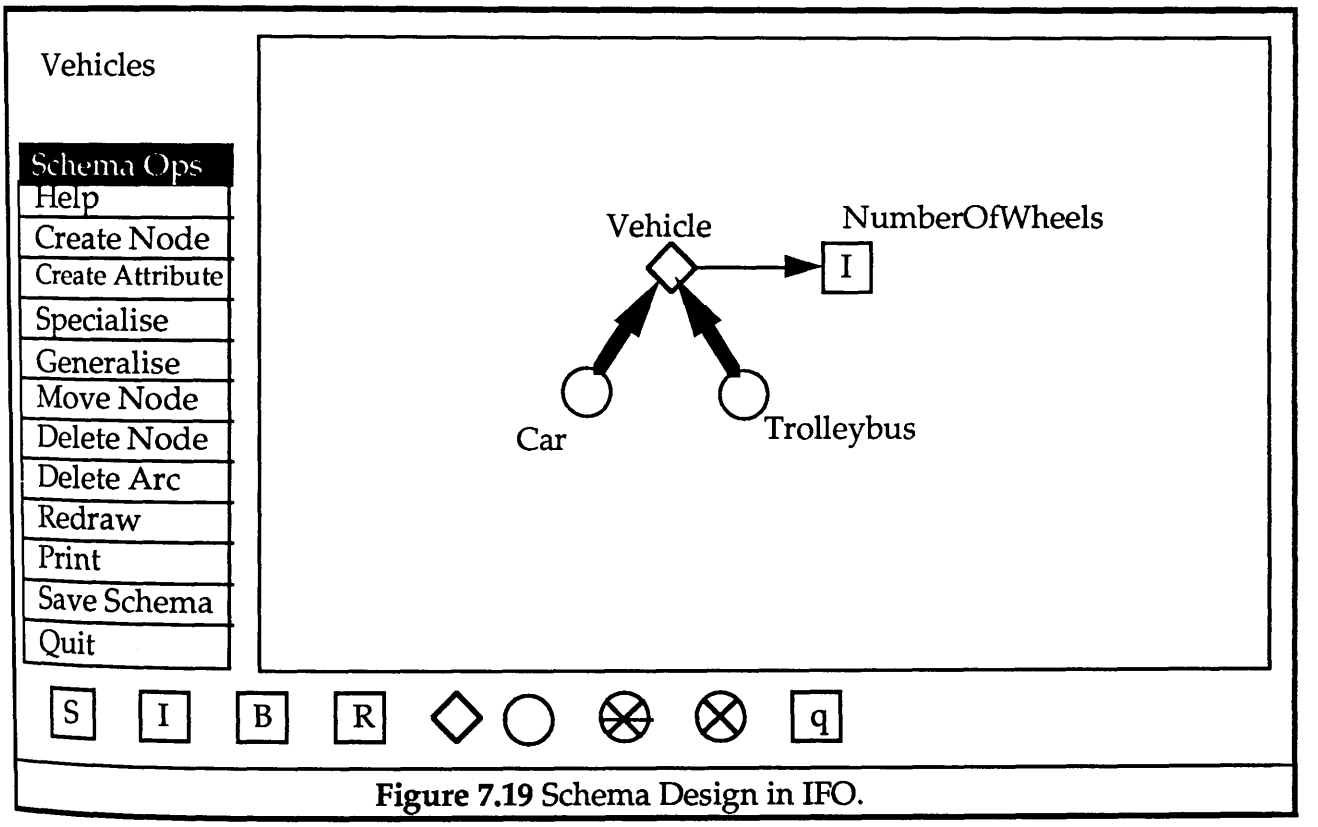


Figure 7.19 Schema Design in IFO.

7.3.2 Schema Definition in PS-algol IFO.

The schema definition interface breaks the screen into three windows as displayed in Figure 7.17. The schema under construction is in the upper right window of the screen. At the bottom is a row of light buttons representing different node types. These include:

square boxes with "S", "I", "B" and "R" for the basic types: string, integer, boolean and real;

a diamond for atomic types;

a circle for "free" types, i.e. types which are specialisations or generalisations of other types;

a circle with a star in it for sets;

a circle with a cross in it for aggregate types;

and a box with a "q" in it.

This last box does not represent a node type, but instead is used to quit the creation of relationships involving more than one object.

On the left of the schema definition window is a menu with a number of options for modifying the schema. Note first that the general method of adding to the schema is to create nodes and then the relationships between them. The options are:

create node - the user must select one of the symbols from the row in the bottom window of the display for the kind of node. Then the position in the graph is selected, by clicking over a point in the window when a name for the node is entered. For set nodes, the user must then select the node of which this one is a set. For aggregate nodes, the user must successively select the component nodes, ending the selection by clicking over the "q" box.

create attribute - the user selects a node and then selects other nodes to be its attributes, ending by selecting the "q" box.

specialise - the user selects a free node on the graph and then selects the node(s) of which it is a specialisation. The creation of specialisation arcs is halted by selecting the "q" box.

generalise - is similar to specialise. Again, a free node is selected and then the nodes of which this one is to be a generalisation are selected.

delete a node - the node is selected. If the type has no data instances and there are no dependent nodes, it will be deleted. Otherwise, the user is warned and, only if a direction to continue is given, does the node and all of its dependent information get deleted.

delete an arc - the user draws the mouse across the arc with the button held down. The relationship is removed from the schema, but if there are data instances which use this relationship, again the user will be warned.

move a node - the user clicks and drags the node to be moved. All the relationship arcs are moved in sympathy.

redraw - sometimes the movement and deletion options cause disturbance to the schema diagram. This option refreshes the diagram on the screen.

print - send the current diagram to be printed by a laser printer.

save the schema - the schema is made persistent and is given an identifying name.

7.3.3 Data Manipulation and Update Semantics

A similar menu is given for data manipulation:

add an instance - not allowable over printable nodes. The user is requested for values of any attributes of the node. For different kinds of node, further information may be demanded:

set nodes - values to go into the set are requested;

specialisation nodes - attributes of the nodes being specialised are also requested.

generalisation nodes - select first which subtype it is and then proceed as if that node had been selected, except that the value will also be added to the set of instances of the generalised node.

Note that each time a value of a given node is created, it is added to the set of instances of the node. Thus, if A is a specialisation of B, creating an A calls for the creation of a B as well. Instances are created both in sets of A and sets of B, with a link between the two instances. Selecting a value for a printable node requires a value to be typed in. Selecting a value for any other type is done by first providing a menu of instances, using the Chooser. The user may select one of these or create a completely new instance.

delete an instance - remove an object selected by menu. Check if there are any dependencies upon it. If there are, warn the user and give the option of: aborting the delete; deleting the dependents; or replacing the dangling reference.

edit an instance - display the instance and use the display as a menu giving a choice of which attribute to edit. Changing a base attribute calls an editor of the appropriate type. Changing a complex attribute calls a menu of all the objects of its type.

display an instance - select a node and then an instance of that node, which will then display the instance with all of its dependent data.

These operations are designed to preserve data integrity. Whenever an instance is created, all of its relationships are given values. Whenever an instance is deleted or edited, or a type is deleted, all of the references to changed or edited objects are kept secure. That is, a referend can only be removed if the reference is transferred to some object of the appropriate type. Maintaining these sorts of integrity constraints was considerably simplified by Hull and Abiteboul's analysis of update semantics.

7.3.4 Implementation Details.

7.3.4.1 The User Interface.

The user interface has been constructed to provide the kind of graphical support for schema and data management that semantic data models potentially facilitate. The interface was constructed easily by using the following:

The tools described in Chapter 4 were used to provide menus of operations (*menu*); error-messages (*error.message*); the entry of integer and string data (*ieditor* and *seditor*); and the automatic generation of menus of objects (*chooser*).

The picture type was used to create the various icons of nodes and arcs.

The raster operations were used to do all of the manipulation of the schema and data graphs. Very little code was required to specify these operations.

7.3.4.2 The Representation of Types.

The types form a graph of type nodes, all of which are stored in a PS-algol table. There are different kinds of nodes, but all are represented with the following common structure:

```
structure Node(     string NodeName;          ! a name for the type
                    string NodeType;          ! whether "set", "aggregate", "string", etc.
                    integer Xc;               ! the X position in the schema graph
                    integer Yc;               ! the Y position in the schema graph
                    pntr setOf;               ! if a set type, the node which it is a set of
                    pntr inSet;               ! the reverse link to the above
                    pntr attr;                ! a list of attribute nodes
                    pntr inAttr;              ! a reverse link to the above
                    pntr aggOf;               !if aggregate type,list of nodes aggregated
                    pntr inAgg;               ! a reverse link to the above
                    pntr speSub;              ! a list of nodes which specialise this one
                    pntr speSuper;            ! a list of nodes specialised by this one
                    pntr genSub;              ! a list of nodes generalised by this one
                    pntr genSuper;            ! a list of nodes which generalise this one
                    pntr instances      )     ! a table of instance objects.
```

Thus, the implementation achieves simplicity at the cost of some redundancy - e.g. every node, whether it is a set node or not, has a slot for a reference to the node of which it is a set.

The operations to create and delete type nodes are simple pieces of code. Creation consists of making a new node and forming the correct kind of bi-directional link to existing nodes and then drawing the new node in the type graph window. Deletion consists of breaking all the links and removing the node from the screen.

7.3.4.3 The Representation of Data.

All data instances also make up a graph, the nodes of which are similarly stored in a common structure. This has been set up to permit the retrieval of the whole of a data object by following pointers. This results in a structure with a lot of one way links so that when retrieving an object, one also can retrieve: all of its attributes; the objects of which it is a set or an aggregate; or the object of which it is a specialisation or generalisation. The structure is as follows:

```
structure Instance(     string InstanceName;    ! the identifier for the instance
                        pntr self;              ! if of a printable object, the packaged value
                        pntr IsetOf;            ! if a set object, a list of objects in the set
                        pntr IattrOf;           ! a list of the object's attributes
                        pntr IaggOf;            ! if an aggregate object, a list of objects aggregated
                        pntr IspeSuper;         ! a list of objects specialised by this one
                        pntr IgenSub;           ! a list of objects generalised by this one
                        pntr genSuper;          ! a list of nodes which generalise this one
                        pntr instances      )   ! a table of instance objects.
```

Again, there is a great deal of redundancy, with a consequent simplification of the code which creates objects and the code for browsing over the data values.

The object creation procedure is recursive, starting from a given node, where it creates a new object with a new identifier, and visiting any subordinate node. When the procedure reaches a new subordinate node, the user is given a menu of all objects of that type to choose between (by a call to the Chooser). If none is chosen, a new object can be created instead. Thus there may be a cascading creation of objects, all handled by a short piece of code.

Data retrieval is similarly straightforward. The user selects a type node and then gets a menu of instances of that type. A recursive procedure then retrieves and displays the subordinate information by traversing the instance graph.

7.3.4.4 The Structure of the Program.

The operations are coded as a set of parameterless procedures, which are all short. The operations are held together by a simple hierarchy of menus and the consequent code comes to about 1000 lines of PS-algol code.

7.3.5 Summary.

The functions of the data model have been provided in a convenient graphical form in a relatively short program. Note that the program has not used any of the complex run-time compilation techniques used in PSRML, since the model does not encompass user-defined polymorphic operations, nor has there been an attempt to provide optimal efficiency. The decision between using interpreted and compiled code

will be discussed in Section 7.5. The model is relatively sophisticated and it may be concluded from this that the implementation of any data model which deals purely with the description of the data structure of objects may be simply programmed in PS-algol, even one providing such a rich set of modelling tools as Hammer and McLeod's SDM [Hammer & McLeod, 1981].

The facilities of PS-algol that have been of most use here are the graphical types; the simple re-use of the tools described in Chapter 4; the ability to use first-class procedures to form hierarchies of menus; and above all the **pntr** type. This allowed the structures shown in section 7.3.4 to be set up simply, without concern for the type of the referend, and permitting the choice of that type to be determined at run-time. These features all contributed to the simplicity of the code and the consequent brevity of the program.

The implementation was tried out on a small group of relatively sophisticated users. The general feeling was that the model was somewhat fussy in its restrictions and that the variety of node types tended to confuse rather than clarify the schema design. Some of this may have been due to implementation and user interface design decisions, but the response seems mainly due to the model itself, which was compared unfavourably with the more familiar Entity Relationship model. On the other hand, some of the aspects of the implementation received favourable comments. The direct manipulation style makes the program easy to learn. The ability to populate the schema with small data sets also proved a useful test of the structure. Most popular of all, however, was the facility that allowed the schema to be printed and taken away for further offline work.

7.4 The Implementation of a Minimal Object-Oriented Language.

One of the outcomes of the interest in Semantic Data Modelling is the development of Object-Oriented programming languages (OOPLs), in which programming in general is eased by the closer correspondence of program elements with the real-world objects being modelled (see section 2.2.2 for further discussion of OOPL's). The elements of an Object-Oriented language are typically: object identity; the classification of objects; inheritance; and encapsulation.

The implementation of an OOPL is clearly a different proposition from the provision of an applicative or even a functional language. Much of the organising work is being done for the programmer by the language environment. This cannot be without cost to those providing that environment. Organising methods, inheritance and encapsulation in a software environment which gives the programmer as little help as writing C under UNIX, for instance, is a daunting prospect. The compiler and run-time support system must be pieced together with the semantic detail of the supporting software being lost in the programming detail.

The crucial difference between PPL's and OOPL's lies in the support for dynamic aspects of the software system in these languages. In an OOPL, these are tied to the objects of the system, whereas in a PPL, data and program co-exist as elements with equal rights. What emerges from this comparison is that the methods of Smalltalk can be programmed in PS-algol, with its first-class procedures, although the converse is difficult to imagine. In this section, methods of building the significant facilities of an OOPL into a small PS-algol program are described. The language, called MINOO

[Cooper, 1989b] (minimal Object-oriented language), has been designed consciously to omit as many as possible of the common and well understood programming language features. There are no computational constructs, expressions or arbitrary length names. There are typed objects, inheritance, the description of operations and attributes, overloading and dynamic binding. All data access could proceed via message passing, although it was decided to provide direct access to the attributes as well, because this was felt to be useful.

The rest of the section will consist of: a description of MINOO; the ways in which the various components of MINOO were implemented in PS-algol; and conclusions about the suitability of PS-algol for the task and for programming languages of the future.

7.4.1 A Minimal Object-Oriented Language

Before discussing MINOO and its implementation, the nomenclature used will be defined, since the nomenclature in the literature is varied. The basic entities of the language are called **objects**. A **type** is the abstract description of a set of objects with common properties. An **attribute** is a passive property of a type. An **operation** is an active property of a type. An object may be referred to as an **instance** of a type.

Each object in the language is an instance of a particular type. A type consists of sets of (notionally private) attributes and (public) operations. The basic commands of the language permit the creation of types, the creation of instances, the assignment of values to objects and the execution of operations. The commands have been provided in the form of old-fashioned single-character named commands (":" for type creation, "I" for object instantiation, etc.) to ease the implementation. Commands are terminated by semi-colons and all layout characters are ignored. There is no benefit in analysing the syntactic quality of MINOO, as its only purpose is to demonstrate how the relevant semantics may be implemented in PS-algol.

7.4.1.1 Type creation

There are three base types, "s" string, "i" integer and "b" boolean. User-defined types can be added by type creation commands. The syntax of type creation allows the user to specify a name for the type, the name of a supertype, a set of attributes and a set of operations. The example:

```
:A      :B
        . k:s, l, m: C
        ! f( q: s ) !$!p(N="k", V=q );      R;      ;,
          r( - C ) I z:C = $!g( N = "l" );  Rz;     ;       ;
```

which is explained as follows:

":" introduces a type definition;
"*A*" is the name of the type (all names are single letter);
":*B*" means *A* is a sub-type of *B* (inherits all *B*'s attributes and operations);
"." introduces the new attributes ;

"*k*:*s*" introduces an attribute of type *s* (a system-defined base type for strings), whose name is *k*;
",*l*,*m*:C" introduces two more attributes *l* and *m* of user-defined type *C*;
"!" introduces the new operations, defined on type *A*;
"*f*(*q*:*s*)" is the signature of the first operation - its name is *f* , it takes in a string parameter *q* and has no result;
"!$!*p*(*N* = "k", *V* = *q*);" is the first command of the operation - it takes object $ (which means "self" in MINOO) and applies the operation *p*, passing in actual parameter values "k" and *q*, for formal parameters *N* and *V*. *p* is a system defined attribute-setting operation (see Section 7.4.1.5) which, in this case, sets the attribute named *k* to value of *q*;
"**R**;;" terminates the operation, returning nothing;
"," means there are more operations;
"*r*(- C)" is the signature of the second operation, which returns an instance of type *C*;
"**I**z:C = $!*g*(*N* = "l");" - this command creates *z*, an instance of type *C*, and initialises it to the result of executing another system operation, *g*, on "self". This operation returns the value of the named attribute in this case *l*. The effect of the command is to create *z* as the value of attribute *l*.
"**R**z;;" terminates the operation, returning the value of *z*;
";" - the last semi-colon finishes the type definition.

So operation *f* sets the value of the *k* attribute to the input parameter, while *r* returns the value of the *l* attribute.

Thus, a type definition consists of its name, an optional supertype, a list of typed attributes and a list of operations. The supertype may be omitted, in which case the supertype is *e* or "entity". An operation has a list of typed arguments of arbitrary length and one or no result types. The body of an operation consists of a sequence of commands separated by semi-colons and these are drawn from instantiation, assignment and operation invocation commands and terminated by an operation return command.

7.4.1.2 Instantiation

Objects are introduced into the system by Instantiation commands. These require the specification of the object's name and type. The values of the attributes of the object may also be specified. For instance, in the command

I *a*:*A* = (*k*="abc", *l*=X);

"**I**" introduces the instantiation command;
"*a*" is the new object identifier;
":*A*" introduces its type;
"=" introduces attribute values - this could be omitted and default values for the attributes would be assumed;
" *k*="abc" " gives *k* a string literal value;
" ,*l* =*X*" gives l the value of object *X*, which must be of type *C*, or one of *C*'s sub-types, of course;
" ; " ends the instantiation - note property *m* takes a default value.

The values of the attributes can be specified as expressions of arbitrary complexity as described below.

7.4.1.3 Assignment

Object assignment is performed by commands like

A *a* = *b* or **A** *a* = *c*!*f*(*q* = "abc")

where the "A" introduces assignment and the right hand side of the assignment may be any expression of the type established for *a* when it was instantiated.

7.4.1.4 Operation Application

The application of an operation of an object is introduced as in

! *c*!*f*(*q* = "xyz")

where *c* is the object name, *f* the operation name, *q* the name of a parameter and "xyz" its input value. At command level, the interpreter prints out the value of the result of applying the operator. Use of the "!" command indirectly within an operation itself is only valid for operations that do not return a result, although other operations can appear as part of an expression.

7.4.1.5 System Provided Operations

Each type needs to have four operations automatically defined on it. These are:

"*c*": create an instance - "*c*" takes in a set of attribute name, value pairs and returns the created object. It is only called by an "I" command.

"*s*": (show) print an object - "*s*" recursively traverses the properties of the object, printing any base-type information it can find. Thus the program:

```
: A. k:s;                          ! Create type A, with one string field, k.
: B. l:s, m:A;                     ! Create type B, with string field, l and a
                                   !       field m of type A.
I a:A = (k ="abc");                ! Create an A called a, field set to "abc".
I b:B = ( l = "def", m = a );      ! Create a B called b, with fields set to
                                   !       "def" and a.
! b!s();                          ! Apply the show operation.

        which will give                    ( "def", ( "abc" ) )
```

"*p*": put an attribute value - "*p*" takes a attribute name as parameter *N*, a string, and a value as parameter *V*, and sets the attribute's value appropriately, as in:

```
! a!p( N = "k", V = "ghi" );
! a!s();                          will give     ( "ghi" )
```

"*g*": retrieve a property value, given the attribute value -

! *a*!*p*(*N* = "k", *V* = *b*!*g*(*N* = "l"));

retrieves property *l* of *b* and sets property *k* of *a* to it. Note that handling these operations "*p*" and "*g*", which have a polymorphic parameter, *V*, in a strongly typed environment may be expected to give some problems.

7.4.1.6 Extra Redundant Syntax

Although the above is sufficient, some extra syntax was added to make the process of testing the interpreter tolerable.

a) The usual dot notation for attributes was added - thus

b.l could replace *b*!*g*(N = "l")

in the above example. The effect, however, is identical, although it does mean that the attributes have been rendered public, which might be desirable anyway.

b) Similarly, a print command was added:

P *a,b;* is the same as ! *a*!*s*(); ! *b*!*s*();

c) A dump facility was added - thus

D;

applies the "*s*" operation to everything in the symbol table.

d) Finally "?" quits the interpreter.

7.4.1.7 Expressions

Given the two notations, an expression handler was built which permits the free mixing of "."'s and "!"s. Thus, imagining type *C*, with an attribute *y* of type *D*, where *D* has an operation *o* which takes a string parameter *p*, then

a!*r*().*y*!*o*(*p*="123")

is an expression which takes *a* of type *A*, applies its *r* operation, returning something of type *C*. This has the *y* attribute dereferenced and the resulting object has its *o* operation applied. The type of the expression is the same as the type of operation *o*.

7.4.2 The Implementation

The first point to be noted was that the interpreter was written without recourse to automatic compiler-generation tools. Secondly, it was decided to implement the

language incrementally, starting with the base types and gradually adding the other features.

7.4.2.1 The Type Structure and Base Types

A simple PS-algol structure was created to hold types:

```
structure type(     string tname;
                    pntr subtypes;          ! a table of subtypes
                    pntr supertype;         ! a single supertype
                    pntr properties;        ! a table of name -> type
                    pntr operations;        !       ditto
                    pntr class )            ! a table of instances
```

Two base structures, a table of types and the symbol table were set up at this point. Then a most general type, "*e*" was created to act as the bottom of the type hierarchy. This looks like:

```
let eType = type( "e", table(), nil, table(), table(), table() )
```

and is unique in having **nil** in its super-type field.

The three base types were then set up. They required, first of all, PS-algol structures to hold their instances. These were:

```
structure stringBox( string stringVal )
structure intBox( int intVal )
structure boolBox( bool boolVal)
```

Then the three type structures were created. For strings, the type looked like:

```
let sType = type( "s", nil, eType, table(), table(), table() )
```

That is, there are no sub-types of a base-type, the super-type is *eType* and the three final fields all initially contain empty tables. The attributes table will remain empty, the operations table will have the four basic operations ("*c*", "*s*", "*p*" and "*g*") inserted, and the *class* table will have strings inserted as they are created. After *sType* is created, a reference to it in the sub-types field of *eType* is made.

The basic operations for these base types are fairly trivial. The "*c*" operation creates a new instance of a *stringBox* structure, for instance. The "*s*" operation will unpackage a *stringBox* structure and print the contents. The "*g*" and "*p*" operations, on the other hand, do nothing as these base-type structures have no attributes to manipulate.

7.4.2.2 The Interpreter and Expression Evaluation

The basic operation of the interpreter consists of a single control loop which seeks command characters and then continues to interpret a command of this type. The commands it expects are:":", "I", "A", "!", "P", "D" and "?".

The commands "I", "A", "!" and "P" all involve calls to an evaluator for expressions as described in Section 7.4.1.7. Expressions can appear in two places in the language. Here, they are to be directly evaluated with their results being used immediately by the command. They can also, however, appear in operation definitions, in which case they are to be stored for later evaluation. Therefore, the expression evaluator comes in a double form. It may be called either to evaluate the expression or to return a piece of PS-algol code, which when executed will perform the evaluation. The evaluator can handle base-type literals or complex expressions. The evaluation of complex expressions is a mixture of look-ups to get attribute values and executions of operations. This mixture is illustrated with the expression

a!*r*().*y*!*o*(*p*="123")

already encountered.

In command mode, this finds the object *a*; finds its type; finds the *r* operation and applies it; it then finds the resulting type and calls the *g* operation, with input *N* ="y"; this results in another object whose type is found; operation *o* is then found and applied with *p* set to "123". The result of this operation is the result of the whole expression.

If the expression appears in the definition of an operation, the code illustrated in Figure 7.20 is created.

```
let typeA := s.lookup( "A", T )                    ! Type A looked up when a
                                                   !   was instantiated.
let parameters1 := table()                         ! The parameters for r
let parameters2 := table()                         ! The parameters for g, which
s.enter( "N", parameters2, stringBox( "y" ))       !   was inferred from the dot.
let typeY := s.lookup( "Y", T )                    ! The type of the y attribute.
let parameters3 := table()                         ! The parameters for o found
s.enter( "p", parameters3, stringBox( "123" ))     !   explicitly.
let typeO := s.lookup( "O", T )                    ! The type of the O operation.

stringBox( s.lookup( "o", typeO(operations) )( ocode )(
    T, ST, s.lookup( "g", typeY(operations) )( ocode )(
        T, ST, s.lookup( "r", typeA(operations) )( ocode )(
            T, ST, a, parameters1 ), parameters2 ),
                parameters3 )( stringVal ) )
```

Figure 7.20 Generated Code for Expression Evaluation.

That is, the program sets up references to tables for the parameters to the three operations and then does look-ups for the operations and applies them. The applications are performed from the innermost outwards. That is, the code for operation *r* of type *A* is applied to an empty parameter set (*parameters1*). Then the *g* operation of type *Y* is retrieved and applied with the parameter *N* set to "y" (*parameters2*). Finally, the *o* operation of type *O* is retrieved and applied to the result of this, with parameter *p* set to "123" (*parameters3*). The result is packaged into a *stringBox*.

7.4.2.3 Type Creation

This part of the interpreter proceeds as follows:

a) Create a new type structure, with the name in it and all other fields empty.

b) Scan the input for a supertype. If there is one, put a reference to it in the relevant field, otherwise use the default, *etype*. At the same time, put a reference to the new type in the *subtypes* field of the supertype.

c) Read the attribute names and types and insert them into the *properties* table, using the attribute name as the key and the type as the value.

d) Automatically generate the "*c*", "*s*", "*p*" and "*g*" operations and insert them into the *operations* table - see next section.

e) Read and parse the operation specifications and create operations for insertion into the *operations* table.

f) Insert the type into the table of types and quit.

Steps (a) - (c) and (f) need no further elaboration, but the generation of the system and user-defined operations are complex tasks and will be described in the next two sections.

7.4.2.4 Automatic Generation of the System Operations

This technique will be described with respect to the type:

: *B.l*:*s*, *m*:*A*.

where *A* is some previously defined type. From this input, the five lexemes *B*, *l*, *s*, *m* and *A* are available, where *s* and *A* have been checked to be valid types.

These can be used to build an appropriate PS-algol structure for *B*, which is

structure *TypeB*(**string** *B.l*; **pntr** *B.m*; **pntr** *B.super*, *B.type*)

(where the last two fields point to inherited attributes of the object and the object's type) and around this are built the four procedures. It was decided to create a procedure type sufficient to cater for user-defined operations as well and this is

proc(**pntr** *theTypes*, *SymbolTable*, *self*, *params* -> **pntr**)

where the first two parameters are required to import the local MINOO environment into a procedure so that the procedure can be compiled independently. In order to store these operations in tables, they need to be packaged into PS-algol structures. The following was chosen for the purpose:

```
structure operation(    string oname;
                        proc( pntr, pntr, pntr, pntr -> pntr ) ocode;
                        pntr arguments, resultype               )
```

in which the four fields hold the name, the compiled code, a table of argument name, type pairs, and a pointer to the type of the operation's result type.

The operation building technique is illustrated with respect to the "s" operation which prints out an instance, as shown in Figure 7.21. The purpose of this operation is to print out the attributes of an object separated by commas and surrounded by parentheses. If the attribute is complex, this in turn is printed in a further set of parentheses. Inherited attributes are also printed, preceded by a colon.

The operation receives a pointer to the object as its third parameter, *OBJECT*. The first and second parameters import the type and symbol tables from the current environment, while the fourth parameter is a dummy since *s* does not take any parameters. Following some initialisation, including building in the structure for this type and the general purpose structures for types and operations, the operation finds the object's type. Then it starts the printing with a "(", before directly printing field *l*, which it can do immediately as it is a base type. Printing the other field is more complicated. The operation must look up the field's type (*A* in this case) and then find the "s" operation for type *A*, which can then be called with the value of field *m* as its principal (third) parameter. The inherited information is printed out in the same way as for complex attributes. The "s" operation is found from the type's supertype and it prints the structure containing the inherited attributes.

```
proc( pntr T, S, OBJECT, dummy -> pntr )
  begin
    structure B( string B.l; pntr B.m; pntr B.super, B.type )
    structure type( string tname, tstruct;
                    pntr subtypes, supertype, properties, operations, class )
    structure operation( string oname; proc( pntr, pntr, pntr, pntr -> pntr ) ocode;
                    pntr arguments, resultype )
    let thetype = OBJECT( B.type )
    let TT := nil;                                  ! The type of an attribute.
    let subshow := nil                              ! The s operation of TT.
    let dummyResult := nil                          ! Receives dummy result form
    write "( "                                      !    recursive calls.
    let commas := ""                                ! Flag to omit comma before
    write commas                                    !    first field.
    commas := ", "                                  ! Print commas from now on.
    write """,OBJECT( B.l ),""""
    write commas
    TT := s.lookup( "A", T )                        ! Lookup type A.
    subshow := s.lookup( "s", TT( operations ) )    ! Get A's s operation.
    dummyResult := subshow(ocode)( T, S, OBJECT( B.m ), nil )      ! Print attribute.
    let superT = thetype( supertype )
    if superT( tname ) ~= "e" do                    ! Print inherited data if there are
      begin                                         !    any, i.e. the supertype is not "e".
        write ": "
        dummyResult := s.lookup( "s",superT( operations ) )(ocode )
                          (T, S, OBJECT( B.super ), nil )   ! Print inherited values.
      end
    write " )"
    nil                                             ! A dummy result.
  end
```

Figure 7.21 The Automatically Generated Operation *s*.

This procedure was built entirely by string manipulation with only the parts underlined being derived from the type information. This string is passed to the compiler and the resulting compiled code is put into an *operation* structure, with name "s", a blank table of arguments and a **nil** for result type. This structure is then inserted into the *operations* field of the type being created.

7.4.2.5 User-Defined Operations

The interpretation of user-defined operations will be illustrated for the *r* operation of type *A* given above. This results in the creation of the PS-algol procedure given in Figure 7.22.

```
proc( pntr T, ST, OBJECT, PARAMS -> pntr )
  begin
    structure type( string tname, tstruct;
                    pntr subtypes, supertype, properties, operations, instances )
    structure operation( string oname; proc( pntr, pntr, pntr, pntr -> pntr ) ocode;
                    pntr arguments, resultype )
    structure plist( string pname; pntr pvalue, pnext )
    structure A( pntr A.l; pntr A.m; string A.k; pntr A.super, A.type )
    structure stringBox( string stringVal )
    structure intBox( int intVal )
    structure boolBox( bool boolVal )
    let vobject := nil;                          let initials := nil
    let parameters := nil;                       let dummy := nil
    let ztype := s.lookup( "C", T )
    let parameters1 := table()
    s.enter( "N", parameters1, stringBox( "1" ))
    let typeA := s.lookup( "A", T )
    let z := s.lookup( "g", typeA(operations) )( ocode )
                                    ( T, ST, OBJECT, parameters1 )
    z
  end
```

Figure 7.22 A User Defined Operation in MINOO.

This procedure takes in the table of types, the symbol table, the object and a table of the input parameters. It starts of by defining all of the structures it needs, including its own type structure. Certain dummy variables are declared, not all of which are used in the context of this particular procedure. Then the command "I z:C = $!g(N= "1")" is transformed into code which does the following:

looks up the type of z as *ztype* - this is not used here but would have been used if the initialisation had involved creating a new object rather than copying one;

sets up a table of actual parameter values, with one entry pairing *N* and "1";

finds operation "g" of type *A* and applies it to the current object, *OBJECT* - the result is z.

Lastly, the "Rz" command is compiled by placing a reference to z as the last line of the procedure -this has the effect of returning the pointer to the object z.

Thus a general purpose compilation of MINOO "methods" into PS-algol procedures has proved possible. The basic method of this compilation is to turn strings in MINOO into strings of PS-algol code, noting any objects which need to be looked up from the environment. Those look-ups are then inserted into the procedure before the code which uses them.

7.4.2.6 Polymorphic Operations

It has been noted above that the system operations deal polymorphically with attributes of any type. There is no trouble in providing "s" (as shown in Section 7.4..2.3) and "p", since the calling program will know the type of the object which it sends. In the case of "g", however, there is a problem, since it returns an object of unknown type. This point may be illustrated by looking at the procedure for "g" of type *A* in Figure 7.23. Here the procedure picks up the property name from the parameter table and checks which of the three attributes of *A* it is. If it does not find the attribute in the current type, it refers to the super-type to find it. Whichever it is, it dereferences the field value from the structure and in the case of the string attribute packages it up in order to return it as a structure. What also happens, however, is that the type of the result is inserted into the slot for the result type of the operation. This means that the calling program can use this field to check the type of the result in the same way as is done for non-polymorphic procedures. The procedure is given as Figure 7.23.

```
proc( pntr T, ST, OBJECT, PARAMS -> pntr )
  begin
     structure type( string tname, tstruct;
                     pntr subtypes, supertype, properties, operations, instances )
     structure operation( string oname; proc( pntr, pntr, pntr, pntr -> pntr ) ocode;
                     pntr arguments, resultype )
     structure plist( string pname; pntr pvalue, pnext )
     structure A( pntr A.l; pntr A.m; string A.k; pntr A.super, A.type )
     structure stringBox( string stringVal )
     structure intBox( int intVal )
     structure boolBox( bool boolVal )
     let propname = s.lookup( "N", PARAMS )( stringVal )
     let theT = s.lookup( "A", T )
     let theO = s.lookup( "g", theT( operations ) )
     case propname of
          "l":      { theO( resultype ) := s.lookup( "C", T ); OBJECT( A.l ) }
          "m":      { theO( resultype ) := s.lookup( "C", T ); OBJECT( A.m ) }
          "k":      {theO( resultype ) := s.lookup( "s",T );stringBox( OBJECT( A.k ) ) }
          default:
            begin
               let superGet =s.lookup( "g", theT(supertype)( operations ) )
               let temp =superGet( ocode )( T, S, OBJECT( A.super ), PARAMS )
               theO( resultype ) := superGet ( resultype )
               temp
            end
end
```

Figure 7.23 The Polymorphic Automatically Generated Operation, *g*.

This is an instance of a generally applicable technique installed directly into this procedure by the system. However, the technique could be extended by replacing the

result type field of the operation structure with two fields, one for its expected type and another for the returned type, which would be a sub-type of the expected type. In the case of "g" the operation expects to return an entity object, but would actually return some other type which it would report in the returned type field. Using techniques of this kind, the various sorts of polymorphism and overriding discussed in Section 2.2.2. are implemented.

7.4.2.7 Object Instantiation

The instantiation command depends heavily on the system-generated "*c*" operations. The command passes to the appropriate "*c*", the type table and symbol table, the type of the new object and a list of attribute name and initial value pairs. The values are derived by calls to the expression evaluator. The "*c*" operation then does the following:

an instance of the type structure is created with a pointer to its type;
if the type has a super-type, the "*c*" function of that type is called and a pointer to the resulting object is put in the super field;
the list of initial values is scanned and fields of the structure filled in as appropriate;
the new object is returned.

The returned object is then put into the class of the type and into the symbol table, which completes the functions of the "I" command.

7.4.2.8 Assignment and Operation Execution

The "A" command finds an object to be assigned to and then calls the expression evaluator to provide the new value. Operation execution is also a very short piece of code, which receives an operation, builds a table of input parameter values and calls the operation. If the operation has a result, this is printed by a call to the appropriate "s" operation.

7.4.2.9 Inheritance

Inheritance is achieved by pointer links. The type structure has two-way links between sub- and super-type. The instances have links from sub- to super-type values. When an attribute for an object is requested by a call to the "g" operation, the operation passes the request up to the super-type if the attribute is not defined in this type. When an operation is requested, again the search for the operation starts at the current type and is passed up the inheritance tree.

7.4.2.10 Summary.

In this section, some of the detail of the implementation has been described. The diagram in Figure 7.24 shows the layout of the underlying data structures. The user-defined types are shown in rectangular boxes split into five parts (the base types appear in single boxes). The type hierarchy is shown as diagonal lines. The five compartments represent the name, the sub-types, the operations, the attributes and the

instances. Instances are shown in rounded boxes, attributes in rectangular boxes with two compartments (the name and the type) and operations are shown as lozenges containing the name, the table of arguments and the result type. Using this simple structure, together with two system objects containing tables of all the types and all the instances, all of the object-oriented data has been represented.

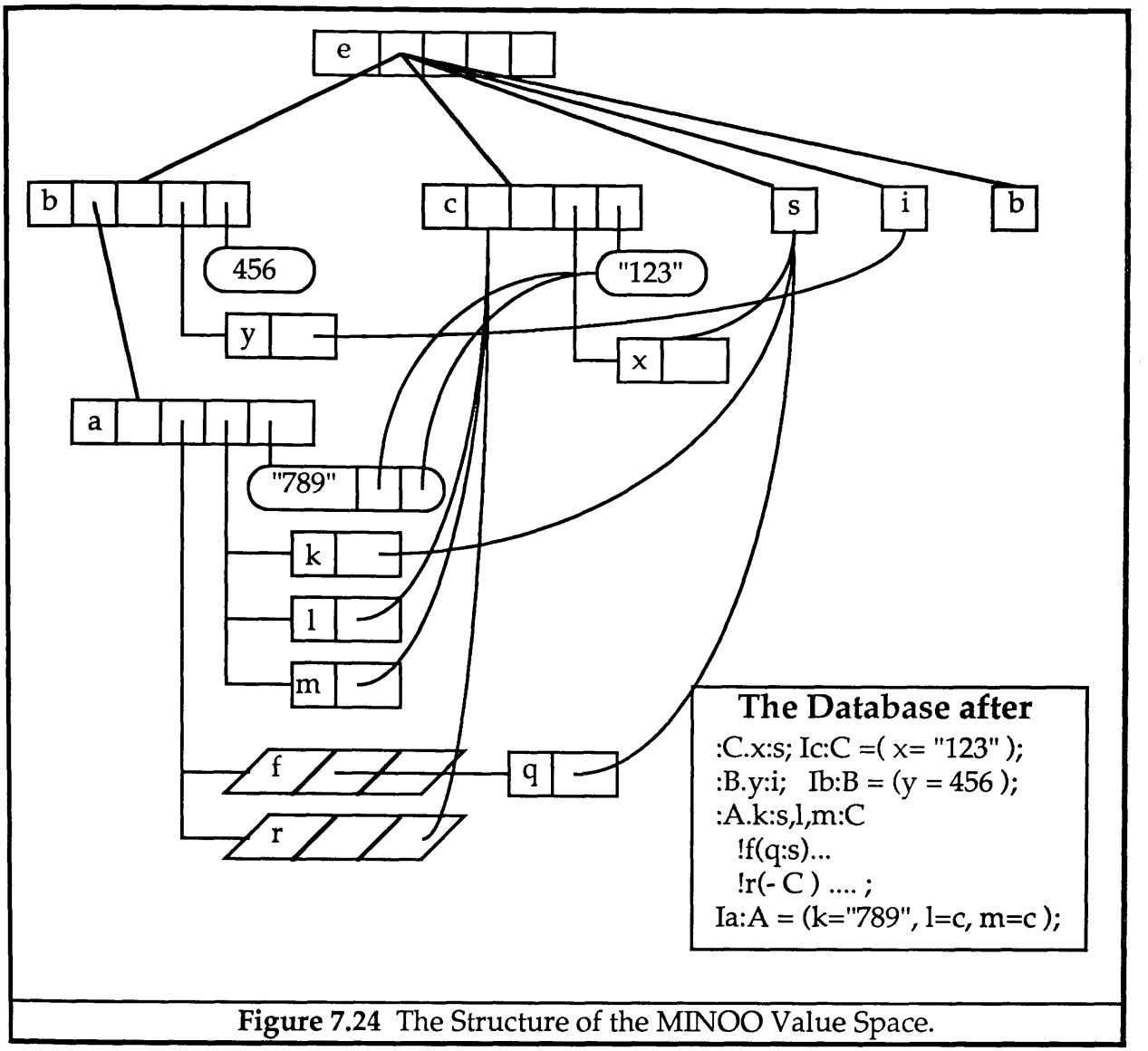


Figure 7.24 The Structure of the MINOO Value Space.

7.4.3 Conclusions Regarding the MINOO Interpreter.

This section has shown how an interpreter for a minimal Object-oriented Programming Language has been implemented in PS-algol. While not tackling some of the well-understood problems of compiler construction, this work demonstrates how implementations of the critical parts of such a language can be achieved. Types and objects are represented by PS-algol structures. Inheritance is achieved by following pointer chains. Methods or operations are compiled into PS-algol procedures which, being first-class elements of the language, may be manipulated freely and stored, retrieved and applied as required. Access to objects can be restricted to procedures stored as part of their associated type.

This work could proceed by developing the language to include further base types, multi-valued types, computational constructs and expressions, although it is

believed that this would require no new technology. Experiments on the representation of types as objects in the system, on the other hand, may provide added simplicity in the resulting structure. More interestingly, multiple inheritance could be included as a method for investigating the semantic problems of inheriting from more than one super-type. The inclusion of parameterised or generic types is a significantly more difficult problem, but would permit experimentation with the various notions involved. The language could be given a sophisticated user interface, using the graphical tools described previously, and including software development tools like syntax-directed editors. Finally, the language could be made persistent. This implementation was never designed to include persistence, which was covered in the previous experiments. From these, it is known that adding persistence when implementing in PS-algol is a very small task.

The constructs of PS-algol have been exercised and found sufficiently robust to handle the demands of the implementation. The simplicity of the language is a great assistance in developing programs quickly. The power of language structures and types which manifest many of the important features of object-orientation (notably object identity and polymorphism) enabled a straightforward implementation of the interpreter. The ability to write general purpose procedures using the **pntr** type and the callable compiler simplified the program, within the security of a strongly typed system and without the environment becoming full of type-ambiguous objects.

The main conclusion to be drawn from this experiment is that an implementation language like PS-algol greatly simplifies the task of implementing an Object-Oriented language. The components of such a language are naturally modelled in PS-algol and so the implementation is kept to a reasonable length (about 1200 lines of code for MINOO). For the implementation of real OOOPLs, such as Smalltalk or Eiffel, the reduction in complexity of the task makes it possible to begin to tackle efficiency issues - for instance, by using the run-time compiler to tailor efficient code. Furthermore, compilers for dynamically typed languages such as Smalltalk can be written which extract statically inferrable types wherever they can and impose them with a resulting efficiency gain. Implementation in a language which simplifies the overall compiler construction task renders the task of doing this more tractable.

7.5 Issues in the Implementation of Semantic Data Models.

This chapter has presented four examples of implementations of high-level data models and programming languages and now, in the concluding section, some general principles concerning their implementation are extracted. The suitability of a Persistent Programming Language like PS-algol is demonstrated.

Before discussing the implementation techniques used for these models, the nomenclature used is defined:

the basic entities in a database will be called **objects;**

a **type** is an invariant abstract description of an infinite set of potential objects with common properties;

an **attribute** is a passive property of a type;

an **operation** is an active property of a type;

a **class** is a collection of objects of the same type;

an object may be referred to as an **instance** of a type or a **member** of a class.

Here the meaning of type is restricted to an intension, with classes playing the rôle of the extension - although in some models there may be more than one class to a type. A schema in a given data model will in general be a graph of types, with a database compliant with that schema being a graph of objects which are instances of those types.

Firstly the rôle of the interface is mentioned and then methods for implementing types, objects and operations in PS-algol are described. Then follows a discussion of the implementation of first-class "active" objects, i.e. objects exhibiting behaviour, such as processes. Finally, it is shown how the schema itself may be brought into the database, giving a desirable reduction in code and improved access to metadata.

7.5.1 The Human Computer Interface.

The principal benefit intended from the use of Semantic Data Models is the formation of sufficiently abstract definitions of the application world for communication during requirements analysis and design. From this should follow a clear definition for implementation and operation, although this aspect was often neglected in early SDM's. The model should be accompanied by effective tools for constructing a schema using appropriate HCI techniques. Many SDM's were only initially envisaged as off-line diagrammatic tools e.g. [Chen, 1976]. These would be used only to design the database schema and this would then be manually translated into some traditional form. With the emergence of good quality terminals and much more powerful hardware, it has become possible to provide graphically-based SDM's as on-line database design tools. Once, the abstract meta-data has been captured and refined in this way, further tools can be used to assist in producing a corresponding implementation [Borgida *et al.*, 1989], once again requiring appropriate HCI tools.

The IFO implementation showed how the graphical facilities of PS-algol supported the diagrammatic "languages" of an SDM as an interface for the database designer and user. Appropriate diagrams can easily be constructed using PS-algol and, more crucially, as the language is data-type complete, the correspondence between an object and its graphical representation can be internally recorded by explicit references. A given type or object can be associated with an icon representing it. Furthermore, animation of data manipulation may also be provided. Whereas all of these facilities can, and indeed are, being implemented using other technologies, the direct manipulation of graphical values in PS-algol eases the task considerably.

7.5.2 The Representation of Types.

The graph of types should be represented in such a way that the relationships between types are revealed. This requires two constructs that will represent the nodes and arcs of the type graph. The nature of these arcs will vary from model to model.

For instance, IFO recognises several different relationships between types: attribution; aggregation; set inclusion; generalisation; and specialisation. There is a further complication in IFO in that there are different kinds of type node and, in general, they cannot all take part in all the different kinds of relationship. An implementation environment should provide powerful enough constructs to avoid baroque code to handle the various kinds of types and relationships.

In all of the PS-algol implementations discussed, a type has been represented by a single instance of a PS-algol structure, while relationships between types have been represented by **pntr** fields within those structures. This is a fairly obvious way of representing a graph, no matter what the graph represents. In creating one of these type graphs, there is one more design choice. Either a common structure can be used for all nodes of the graph, or there can be different structures for each kind of node. All of the examples described have taken the first of these routes, as this seems to simplify the code involved. The COMANDOS type model has been implemented in such a way as to have different structures for each kind of type (unparamaterised, parameterised and instantiated) [Cooper *et al.*, 1989]. This method has demonstrated no clear benefits, but has produced rather more elaborate code which performs efficiently those operations, such as type checking by compilers, which are not frequently carried out.

The type node structures contain essentially three groups of fields: some containing descriptive information, such as the name of the type; some containing the **pntr**s representing the type relationships; and, in some models in which the database and schema have been bound together, a single **pntr** which points to the set of instances of the type. Section 7.5.4 will show that a fourth kind of field is also needed to hold the operations of the type, as they are normally factored out of the instances of the type. This will be a pointer to a set of operations defined for this type. This analysis begins to suggest how to develop a meta-model for describing the types of any data modelling system.

7.5.3 The Representation of Instances.

The discussion is illustrated with objects which represent *PERSON*s, with attributes *name*, *sex* and *age* and *STUDENT*s, which represent *PERSON*s with an extra attribute, *matriculation number*. The first decision in representing instances is whether all the data including inherited information for a given object are to be kept together or not. That is, will the representation for a *STUDENT* keep the four things known about the student together (as in Simula, most Object-Oriented systems and in variants of Pascal); or will there be one structure for the three *PERSON* attributes and another linked structure for the *matriculation number* ?

The latter choice seems preferable for a number of reasons associated with behavioural modelling. Firstly, all *PERSON* objects can easily be scanned since they will be kept together in a set. Secondly, if an object is specialised (for instance a *PERSON* becoming a *STUDENT*) or generalised (the reverse) there is no need to move data around. Thirdly, there is also no need to duplicate data in the case where an object is specialised in the two ways simultaneously. For instance, if the student also becomes a member of staff, there will now be two copies of the *PERSON* data, which might lead to inconsistency.

This decision implies the loss of object-identity - there is not one program object which represents the real world object. In fact, there is no difficulty in tying the sections of data for a given object together via a ring of pointer fields. Although this representation is intended to facilitate access to objects via the extent of their type, there is no difficulty in accessing other information about it using this ring structure.

A general structure is needed for the fragment of a particular instance, one which holds the values of the attributes of one of its types and connects them to instances of other types as appropriate. The general structure for fragments of instances will look something like:

```
structure fragment(   pntr theType,        ! Points to the associated type structure.
                      .....                ! A reference to a set of attribute values.
                      .....                ! References to inherited values.
                      .....          )     ! References to inheriting values.
```

in which the second line holds some representation of the attributes specific to this type and the third and fourth lines point to one or more *fragment* structures. Each object of the system will have one *instance* structure for each type to which it can be considered to belong. The third line will be either a pointer to a single *fragment*, if the model supports single inheritance, or a vector of pointers to *fragments*, if the model supports multiple inheritance. The fourth will be a vector of pointers, if access to inheriting data is permitted. (It is not in PSRML or MINOO, for instance.)

In representing the attributes, there are two basic strategies:

a) provide a general structure for instances of every type;

or b) use the callable compiler to tailor instances for each type.

In the examples above, EFDM and IFO do (a), while PSRML and MINOO do (b). In (a) there is a single field, *theValues*, which points to a set of values - either packaged base values or other *fragment* structures. Implementations of this set may be ordered, such as a list or vector of pointers, or could use a table keyed on attribute name pointing to the values. This technique, which is essentially interpretive, is a little slower and takes much more space, but slightly eases the writing of general programs to traverse the graph of values. A version of *fragment* which provides single inheritance, access to inheriting values and uses method (a) for attributes looks like:

```
structure fragment(   pntr theType,        ! Points to the associated type structure.
                      pntr theValues;      ! A reference to a set of attribute values.
                      pntr super;          ! References to inherited values.
                      *pntr subs     )     ! References to inheriting values.
```

The alternative technique adds a set of fields specific to each type, for instance

string *PERSONname*; **bool** *PERSONsex*; **int** *PERSONage*

by string manipulation and the callable compiler. This leads to much more efficient storage and retrieval, but seems to prohibit the use of general purpose traversal and implies the existence of type specific, tailored access procedures. However, several demonstrations of the power of the run-time compiler have been given, which show how to provide general purpose traversal, as in the browser (Section 3.2), and tailored

access, in GRAPE (Section 6.3). Using these techniques, *fragment* can be made to look like:

```
structure fragment(     pntr theType,           ! Points to the associated type structure.
                        string PERSONname;      ! Three fields to hold
                        bool PERSONsex;         !       the set of
                        int PERSONage           !               attribute values.
                        *pntr super             ! Multiple Inheritance.
                                )               ! No access to inheriting information.
```

which also shows the alternative choices for the inheritance links (a vector for the inherited link, and no inheriting link). This is more efficient and yet can be traversed with the automatically generated procedures.

Note how, within this general framework, type evolution fits quite comfortably. At any time, an object is associated with a number of types. For an object to change type means that it drops some type associations (and their attributes) and gains other type associations (picking up attribute values from the user or from defaults).

In fact, the question of how objects get their attribute data has a number of possible answers. In every system, there will be some code to create objects. In systems like EFDM and IFO, this code exists as a single statically determinable piece of PS-algol which proceeds in an interpretative way. In the tailored systems like PSRML and MINOO, the system generates operations for the creation of instances. In either case, this code can provide null values when creating the objects; ask the user for initial values; or allow values to be provided by any calling code. PSRML provides both the first two options and MINOO provides the third.

Updating attributes is similar to creating objects and there will be pieces of code similar to the creation code for changing the value of an attribute. Again, this is a single procedure in EFDM and IFO, but an operation tailored to the type for PSRML and MINOO. Note that these tailored operations, like a statically written operation, can be written within a single program section. Note also that persistence allows the tailored code to be remembered for future use with similar types, rather than having to be regenerated.

7.5.4 The Representation of Operations.

The previous section shows that even those modelling systems which do not permit the user to define operations themselves, do require operations (such as *createObject*) to be stored. In systems like EFDM and IFO, these operations are provided as schema independent programs. Such programs examine the meta-data and behave interpretively. Owoso refers to these as Universal Programs [Owoso, 1984].

In implementations like PSRML and MINOO, the operations are particularised for each type and so each type has references to the operations available on it. In an implementation like MINOO this provides a convenient place to store user-defined operations as well. The implementation of MINOO shows how user-defined operations may be processed. A compiler for the language in which the operations are written must be constructed. This generates PS-algol procedures to represent the operations and stores the procedures, as described above. MINOO also shows how such operations can be used. An interpreter for calls to the operations was written

which allowed the user to invoke the operation by name or indirectly via the execution of other operations.

Methods have been demonstrated with which systems containing typed objects can be constructed, where the types contain attributes and operations. Instances of the types can be generated, where only the operations can be accessed outside the object or where the attributes may be manipulated directly. Again, this seems to point the way to an analysis of the relative merits of encapsulated and "open" systems.

7.5.5 Active Objects.

Initially, Semantic Data Models could only refer to passive objects. Later models, such as RML [Greenspan 1984] and the Event Model [King and McLeod, 1984], extended the notion of an object to cover "active" entities such as activities, processes, etc. The need for this has arisen in such fields as Office Automation, in which there is a requirement to refer to certain activities as entities in their own right. Thus, the job "Produce the Departmental Booklet" is an activity with a certain structure, including constituent sub-activities and associated static objects (people, information, etc.). Note that this differs significantly from the Object-Oriented approach in that OO data models subordinate activities to some data type. It is expected that, in the future, languages which do not have the facility to describe processes in their own right will lose favour in the production of complex systems. Process modelling, in which processes are described at a very high level of abstraction, is the basis for a design methodology for such a system.

In order to analyse the production of models with active objects, a firm understanding of the nature of such an object is required. Here the view is taken that an active object is an aggregate object, whose components are calls to other active objects, and which also has attributes which are the parameters and variables associated with the object. It is then possible to specify active objects in the abstract manner of RML, so that all of its references (parameters and calls) can be checked for correctness. Thus a model could be checked for consistent references between its constituents before implementation. This was all that was intended in the RML work.

If, on the other hand, there is a desire to go further and check whether the active object behaves as expected, something more is required. In order to "execute" an active object, there needs also to be a mechanism for launching the object and there also need to be some objects which will execute when called, rather than refer to further sub-objects. These objects form the bottom of the lattice of sub-object references and will be called **base active objects**. In PSRML, the base activities are supplied by the system to perform low-level functions, such as creating an instance. It would not be difficult to provide user-defined base active objects, which would be specified in the implementation language rather than the modelling language. PSRML could permit users to specify PS-algol procedures, which become installed as base activities, although this would require some care in designing an appropriate environment within which to compile them.

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

There are essentially two strategies for implementing activities, both of which rely on storing each active object in a common structure, for instance:

structure *activity*(**string** *activityName*; **pntr** *doActivity*; .....)

in which the *doActivity* field points to some representation of the functionality. For base activities this will be the procedure packaged, while for the other activities, a choice of representations is available, either

store a list of pointers to the component sub-activity calls, which contain a binding of the sub-activities to the literals and variables, which will constitute their input when they are called;

or compile a PS-algol procedure, which includes calls to the component procedures.

In PSRML, the former strategy was chosen for a number of reasons. One "Universal Program" will work, since all activities have the same structure, as compared with entities which all have different structures. This also made it easier to enable component sub-activities to be unordered. These were implemented as a list together with an activity execution procedure. This procedure cycled through the list, attempting activities until no more could be executed or all were completed.

However, the latter could be achieved by string manipulation of the object specification and would have the particular merits of unifying the representation of activities and base-activities and speeding the execution, by moving from an interpretive style to a compiled one. For completeness, therefore, there follows a sketch of what such a compiled activity would contain:

i) A signature including the activity name and the input parameters and output parameter type. The input parameters are transformed into the PS-algol type system, i.e. all complex types become **pntr**. If the language only permits single results then the output parameter type is similarly transformed into a PS-algol type. If the language permits multiple results (like PSRML) the output parameter type is **pntr** and the output is a structure designed to hold all the outputs in a consistent manner.

ii) A set of declarations of any local variables and output parameters.

iii) The set of sub-activity calls. If ordered sub-activity calls are specified, then this will consist of a list of calls, generated from the descriptions of the sub-activities. If unordered calls are required, then these are embedded in a systematically constructed framework for cycling through the sub-activities.

iv) The procedure ends either with the result variable, or, if there is no result, it ends with **nil**. For multiple results, these are bound into a systematically generated structure.

Although the callable compiler has not been used in the production of active objects, the **pntr** type and the first-class procedures have been used to provide a common structure for base and complex activities, and this re-affirms that a language

with first-class procedures greatly facilitates the production of systems with active objects in them.

7.5.6 Meta-data Access.

It has generally been regarded as useful if a Data Model can describe itself, i.e. hold meta-data in the same form as ordinary data. This means that the user can query the schema in the same way as the database, thus simplifying the use of the program. RAQUEL demonstrated that the Relational Model can be programmed to hold meta-data (Section 6.1). There, two relations were set up to hold the relations and the columns of the schema. In EFDM, the functions were implemented as entities and so could be queried just like any other object in the database.

The provision in PS-algol of data-type completeness and structures greatly facilitates implementations which support a common view of data and metadata. Having been able to store types in general purpose structures, as described in the previous sections, it becomes possible to distinguish one of the nodes in the graph of types as representing types themselves. This will have a single-valued attribute for the names of types and multi-valued attributes for the relationships and operations. Instances of this type will carry descriptions of the type information.

One word of warning is required here. It may appear that this approach has circumvented the necessary distinction between types and objects in programming languages, which avoids theoretically hard problems of having a type "type". All that has been achieved is that some of the descriptive material about types has been put into the database, using the same structures and the same access methods as the descriptive material about the modelled objects. These structures no more contain types than they do people or students.

7.5.7 Discussion.

The above approach to the creation of data model implementations relies on the following facilities:

- an extensible union type to allow some parts of the program to ignore the type of parts of the data not being processed;
- a mechanism for optionally deferring the binding of program to data and the type-checking of the data as long as necessary;
- a callable compiler so that procedures can be generated and compiled at run-time but whose types are statically determined.

These facilities permit the modular development of programs which are efficient, type-secure and yet extensible to data whose types are subsequently defined.

This chapter has described how to use these mechanisms to provide implementations of object-based database systems. These have included the ability to represent:

- the passive data about the objects of the database in either a general structure which is manipulated interpretatively or in a structure tailored to the object's type;

- the active information about the database, i.e. the operations which are available on it;

- an encapsulated, secure interface to objects;

- a method of storing "active" objects in their own right and not as operations of other objects.

Within this framework, the rapid implementation of a number of data models is possible and from this, a taxonomy of data models and a data model for describing data models within which the models may be compared should emerge.

Chapter 8. Supporting Software Development.

This chapter deals with an issue which arises from two directions at once. From the software engineering point of view, the central problem in the construction of large software systems is keeping track of the software modules. This includes notions of version control and configuration management, the implementation of which seems to become amenable to database techniques if software modules are viewed as data objects. Many systems have been proposed (see section 2.2.4) which use different techniques to manage the problem. The implementation of these proposals has been beset with difficulties, largely due to inappropriate development systems [Nestor, 1986].

The other point of view is that of the PS-algol programmer. When developing application programs in the Persistent Programming Language, PS-algol, the programmer makes considerable use of the availability of first-class procedures in order to develop his program in a modular fashion. There is little support within the language, however, for providing a consistent environment within which to develop program modules. In general, it is hard to find what modules have been put into the Persistent Store, if you do not know where to look. Moreover, if one module calls another, there is no explicit record of the dependency, nor any way of finding what calls what.

The solution of the problem which faces the PS-algol programmer is the creation of a database of procedures, in which the links are explicitly kept together with documentation and version control. In solving this "local" problem, however, techniques have been developed which are relevant to the global problem of software maintenance. The first-class procedures provide precisely the ability to "view software modules as data objects".

This chapter starts by describing in more detail the problem encountered in Chapter 5 - i.e that of maintaining a large PS-algol program. Then a small experiment is described in which a systematically organised database of PS-algol procedures was created to solve this problem. Finally, a version control system is described which, while being unsophisticated, indicates how a more complex system might easily be implemented in PS-algol. Similar version control and naming techniques would be applicable in many CAD/CAM applications. Persistence and the **pntr** type would allow the code prototyped here to be re-used in any of these applications.

8.1 Modular Program Construction in PS-algol.

The provision of first-class procedures in PS-algol encourages the modular design and construction of programs. Essentially, the program is analysed into units of functionality and each is implemented as a PS-algol procedure. The procedure is then stored in the persistent store for later retrieval by calling modules. For instance, a minimum function might be implemented and stored in the "program"/"Library" database by the program given as Figure 8.1. Here, the fifth line of code introduces a structure containing a single field, a procedure which takes two integers as parameters and returns an integer as its result. The sixth line then creates an instance of this structure, the value of the field being the *min* procedure. This is now an object of type pntr and so may be inserted into the table which contains the library of procedures by using an *s.enter* command.

```
let min = proc( int a,b -> int )                    ! A procedure to return the
  begin                                             !   minimum of two integers.
      if a>b then a else b
  end

structure minpack( proc(int,int -> int) minproc )   ! A structure for the proc.
let packedProc = minpack( min )                     ! Package the procedure.

let lib = open.database( "Procedures", "Library", "write" )    ! Find the library.
s.enter( "min", lib , packedProc )          ! Enter the packaged proc into the library.

if commit() = nil                                   ! Commit the change to the library.
            then write "Procedure min stored'n"
            else write "Commit fails'n"
```

Figure 8.1 Storing a Procedure in the Persistent Store.

The procedure is subsequently available for use by other modules, such as a procedure which provides the minimum of a vector of integers. This can be linked to *min* as shown in Figure 8.2. Here, the packaged *min* procedure is retrieved using *s.lookup* and the procedure is unpacked by a field dereference. It is then available for calling by any subsequent part of the program. The body of *minvec* contains such a call and this creates a static binding between *minvec* and the version of *min* found in the library when this program is run to store *minvec*. *minvec* is then entered into the library in the same way as *min*. Note that the declaration and body of *minvec* is entirely statically type checked, the only dynamic check occurring when *min* is unpacked from *minPacked*.

```
structure minpack( proc(int,int -> int) minproc )   ! Retrieve the packaged
let minPacked = s.lookup( "min", lib )              !   procedure and unpack it
let min = minPacked( minproc )                      !   for use.

let minvec = proc( *int V -> int )                  ! A procedure to produce the
  begin                                             !   minimum of a vector of
      let smallest := V( lwb(V) )                   !   integers.
      for i = lwb(V) to upb(V) do smallest := min( smallest, V(i) )
      smallest
  end

structure minvecpack( proc( *int -> int ) minvecproc )  ! A structure for the proc.
let lib = open.database( "Procedures", "Library", "write" )    ! Find the library.
s.enter( "minvec", lib , minvecpack(minvec))        ! Package and store minvec in
                                                    !   one line.

if commit()=nil                                     ! Commit the change to the library.
            then write "Procedure minvec stored'n"
            else write "Commit fails'n"
```

Figure 8.2 Calling a Stored Procedure.

One advantage of this approach is that the *min* procedure, once stored in the database, is accessible for use by any other program that knows of its existence in a database called "Procedures", with password "Library". However, the mechanism is deficient in a number of ways:

- no record of the dependency of one module on another is kept;
- there is no support for the creation of new versions;
- there is no support for documentation;
- the code to store and retrieve modules is unnecessarily complex.

These points will be taken in order. There is an underlying graph of module dependencies (illustrated later in Figure 8.4) which this mechanism obscures. After the two programs above have been run, the fact that *minvec* calls *min* is no longer visible. The link between the two exists only within the closure of *minvec* and this is inaccessible. So, no general purpose program can be written, such as a display facility, which when given a pointer to the "root" module of an application would traverse the module dependency graph (MDG).

The provision of version control is even more important and is totally lacking here. Imagine the effect of making a change to *min*, removing the error that exists in it - yes the error was deliberate! If the source file is edited, re-compiled and re-run, a correct version of *min* will be stored in the database as required. However, no effect will be felt by *minvec*, as this is still bound to the faulty version of *min*, as the binding was static. The module containing *minvec* will also have to be re-run (although not re-compiled), in order to bind it to the new version. The *minvec* module can be modified by moving the lookup of *min* inside the body of *minvec* and this will have the effect of dynamically binding *min* into *minvec* every time the latter is run, thus ensuring that it always uses the latest version. It would be better, however, to provide more sophisticated control over this binding to reflect the variety of reasons for providing new versions. Furthermore, as there is no access to the MDG, when *min* is mended, there is no way of discovering which modules call it, nor which programs need to be re-run to bind these modules to the new version. Finally, the system should have the capability of retaining a history of versions, as in the system described in [Davison and Zdonik, 1986].

Perhaps the lack of any encouragement for documentation may be less serious, but help with this as an application is produced should improve quality and productivity. The approach above permits, if not encourages, the programmer who requires a facility to hack in a quick procedure, dump it in the database, make a call from the module which is under consideration and then forget about it. Allied to this point is the lack of any method of helping the programmer find modules as they are needed. It might be guessed that the minimum procedure is called *min*, but what type are its parameters? At another point in the program, a sorting procedure might be needed. What versions are there and how are they accessed?

The final point is that it is tedious to produce the amount of code required to package and unpack procedures in PS-algol. The language is designed to simplify code and reduce the programming overheads. The overheads seen in the above programs are a consequence of features which provide immense benefits in other areas, but there is no reason why some automatic methods to save some programming effort should not be added. The overheads are made worse by the requirement that every name, including the structure and field names of the procedure packaging, must be unique. A less cumbersome syntax is sought for introducing the simple notion: "this module uses version V of module M in program library L" and "store this module in

the program library L - this is a new module or a new version of M introduced for such-and-such a reason".

To achieve all of these aims, a systematic intermediate packaging structure is introduced with which to represent the nodes of the MDG. This will include pointers to represent dependencies and versions; text fields for documentation and program source; a time field to capture module creation time; other fields for author name; and of course the procedure itself.

8.2 A Simple Library of Utility Procedures.

This section describes a systematically organised library of utility procedures [Cooper *et al.*, 1987a] which was used in most of the experiments described in Chapters 5, 6 and 7, and should assist in the faster development of PS-algol programs. Most of the procedures in the library are those for user interface management described in Chapter 4. The organisation of the library begins to tackle the problems of module dependency and documentation. The library has a coherent structure and some programs which manipulate it have been written. These do the following:

For each procedure in the database, maintain a list of all the procedures dependent on it; a short description of the function of the program; and the date and time of its insertion into the library.

When a new version of a procedure is entered, the user is reminded of any dependent procedures which must be rebound to use the new version.

Display the list of the procedures in the database, with their information.

The section will proceed by describing the organisation of the database which holds the library and then the programs which maintain the library.

8.2.1 The Structure of the Library.

The library is created in the form of a PS-algol table. There is one entry in the table for each procedure in the library, keyed by the procedure name. When a library is created, the table is set up with two system procedures, *prcget* and *prcput*, which retrieve and store procedures respectively (the function of these procedures will be described in section 8.2.2). All the procedures, except *prcget*, are contained in structures of the form:

```
structure intermed( pntr procpaki;     !points to the procedure packaged as below
                    *string depended;  ! a list of the procs which call this one
                    string datestamp;  ! time of insertion
                    string descriptor ) ! short description from prcput
```

The **pntr** field *procpaki* points to an object with one field containing the packaged procedure. The structure of this packaging is conventionally of the form:

```
structure procpak( proc( ...parameter types specific to proc...) xproc )
```

so that the structure name and field name is common to all the procedures and only the argument and result types vary. This packaging strategy has been used to simplify the storing and retrieving programs.

It is envisaged that any large scale application will use two sets of procedures, one a multi-user utilities library, which will be shared between applications, and one a set of procedures specific to the application, which will be either a separate database or a table in the application's own database. The structure outlined above is a general one, which can be used both for the utilities library and for an application-specific library. The latter can be created using the *makelib* procedure described below. In larger projects and larger organisations there may be many such libraries.

8.2.2 Software Support for these Structures.

8.2.2.1 The Initialising Program - dbmaker.

There are two forms of this program. One is a stand-alone program, called *dbmaker*, which sets up a database to contain a system library of procedures, initially containing *prcget* and *prcput* (see next section). The other is itself a procedure in this system library, called *makelib*. This procedure takes a string parameter, *XX*, say, and returns a table of procedures, initially containing *XXprcget* and *XXprcput*. This table can then be stored in the user's database, to create an application-specific library.

Apart from the location of the library and the names of the get and put procedures, *dbmaker* and *makelib* are essentially the same program, which proceeds as follows:

i) A new database or table is created.

ii) The procedure which enters procedures, *prcput*, is created.

iii) The *prcget* procedure is created.

iv) *prcget* and *prcput* are entered in the library - *prcget* as a simple packaged procedure and *prcput* as packaged in the *intermed* structure.

8.2.2.2 Retrieving Procedures - prcget.

When the database or table is set up, it contains two procedures: *prcget*, which retrieves procedures from the database, and *prcput*, which stores them in the database. The function of these two procedures will now be described.

prcget is a procedure which retrieves procedures. As it needs to be retrieved before any other procedures, it cannot retrieve itself and so is stored directly as a *procpak* structure and not via an *intermed* structure. It is retrieved by the following code fragment, which looks it up and unpackages it:

```
let procsdb:=open.database("rutilities","friend","read")
if procsdb is error.record do
    {write "No utilities database - do pdbmaker first'n"; abort}
let prcget=
    begin
        structure procpak(proc(string -> pntr) xproc)
        s.lookup("prcget",procsdb)(xproc)
    end
```

Figure 8.3 Retrieving the Retrieval Procedure.

The procedure is then used by code fragments like:

```
let prcput={ structure procpak(proc(string,pntr,*string,string)xproc)
                prcget( "prcput")( xproc ) }
```

which retrieves *prcput*, described below. The first point to be noted about this clause is that the only parts which vary from procedure to procedure are: the procedure variable name; the key to the procedure in the library; and the type description of the procedure. The first two are conventionally the same as each other, but the fact that the type will vary from procedure to procedure means that the procedures are actually being stored as different *procpak* structures. This would normally mean that in order for more than one retrieval to occur in the same module, *procpak* and *xproc* would have to be named differently for each procedure type. This would complicate matters, so instead {..}‡ have been used to surround a block in which the structuring information appears. By the scope rules of PS-algol, the declaration of that *procpak* structure disappears at the "}" and so the retrieval of *prcput* may be followed by a similar clause which retrieves another, differently typed, procedure.

The function of *prcget* is quite simple: it looks up the procedure name in the library, dereferences the *procpaki* field and returns it. The procedure itself is then dereferenced by looking up the *xproc* field as shown above for *prcput*.

8.2.2.3 Storing Procedures - prcput.

prcput itself is a somewhat more complex procedure. It is retrieved, as above, in the same way as any of the utilities in the library. It is then used to store procedures as in the following fragment:

```
{structure procpak(proc(string,int -> string) xproc)
 prcput("fillstring",procpak(fillstring), vector 0::0 of "",
      "Fill out a string with spaces")}
```

This uses a similar technique of delimiting the scope with {...} in order to localise the packaging which uses the same name for different structures. It contains a call to *prcput* with the following parameters:

string *procname* - the name of the procedure;

‡Note that {...} and **begin** ... **end** are equivalent.

pntr *procpntr* - a pointer to the procedure which is packaged in a structure which is conventionally called *procpak* with a single field called *xproc*;
***string** *dependson* - a list of the procedures which this one calls;
string *desc* - a short description of the procedure.

The procedure *prcput* stores the procedure with following steps:

(i) An empty set of dependencies (procedures which call this one) is set up - this will be filled by the subsequent insertion of procedures which call it.

(ii) Check if the procedure already exists - if not enter it.

(iii) If it does exist, check that it is the same type - if not, the user is given the option of not entering the new procedure, as a procedure may be destroyed unintentionally - if it is of the same type, it is assumed that this is a genuine update, although this too could be made optional.

(iv) The new version of the procedure is entered.

(v) A list of all the dependent procedures is printed, with a recommendation to update them as well.

(vi) The list of procedures which this one calls is scanned and the name of the procedure is entered into their lists of dependencies.

(vii) The procedure is committed.

8.2.2.4 The Library Lister - dblister.

This program lists the library information in a tabular form. It scans the table of procedures, extracting type information and documentation and uses various formatting procedures to create a table which may be displayed, printed or stored in a file.

8.2.3 Discussion.

A small-scale maintenance system for a database of procedures has been described which incorporates a check on procedure dependencies and a small amount of documentation. This was an *ad hoc* system created due to a local need by PS-algol programmers. It became clear that this was insufficient in some respects and that a more ambitious support package could be provided. The next section describes an experiment which takes a step towards that.

8.3 A Simple Module Management System With Version Control.

Producing an application program consists of a number of steps. From systems analysis, there arises a set of modules which will comprise the final program. There

then should be a search for existing versions of any of these modules in the environment within which the program is to be created – clearly this search may influence the first step. Subsequently, there will be modifications to any of these which only approximate to the requirement as well as the production of any module not found. Finally the modules need to be "glued" together.

The diagram in Figure 8.4 represents a multi-application environment. In the centre is a 'system' library of modules of general applicability. Surrounding it are application programs which contain modules specific to the application, as well as calls to system library modules. Each application has a single root module from which it is started and consists of a graph of modules drawn from its own library and the system library.

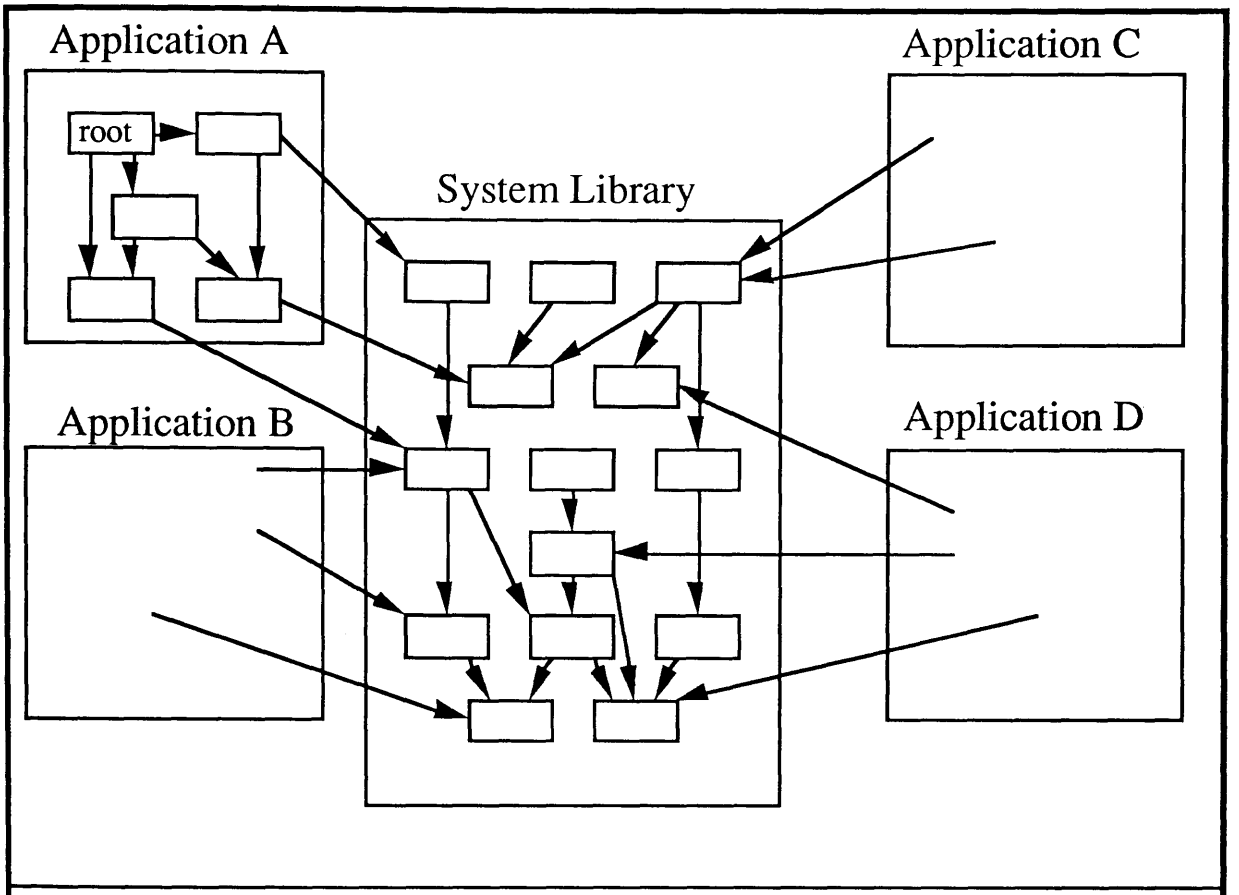


Figure 8.4 Module Dependency Graphs for a System with Four Applications.

Given this view of an application, here is a list of some facilities to be provided:

- a display mechanism for the graph, with facilities for traversing it and homing in on details of individual nodes;
- methods for discovering what modules exist;
- support for versions of modules;
- the ability to establish links between modules and determine the nature of these links;
- assistance in painlessly propagating any changes to low-level modules through the DAG;

- aid in the production of good quality on-line and off-line program documentation.

This section shows how to provide these facilities, in particular concentrating on version control and the way in which modules are bound to their caller. Support is required for the variety of reasons for which new versions of modules are produced. In particular, a new version may be: the removal of a program error; the extension of the power of the module; or the provision of an alternative to the already existing versions (for instance, an implementation of a new sorting algorithm, applicable in different circumstances than those already existing).

There is also a need to support a variety of binding styles. This may include once-and-for-all static binding, dynamic binding to get the latest version of the lower-level module or menu-driven selection between alternatives. In the latter case, the application programmer may make the selection at the time of building the program or pass the decision to the user, who can then, for instance, choose an interface style when starting the program. These different styles can be provided in a language like PS-algol, as it permits a range of times at which programs are bound to data.

The rest of the section describes a set of programs which provide some of the facilities above. This is not intended to be a design for a software management system - that is the business of software engineers. For instance, there is no reference to version merging or configuration management. Instead, the section shows how some of the difficult problems in implementing such systems are eased by using a persistent programming language. The section concludes by describing a small experiment in implementing some of these facilities.

8.3.1 System Requirements.

As stated above, there is no claim that this is a sufficiently powerful version control system for regular use, merely that it illustrates the elements typically found in such systems which are difficult to implement. Version control in a software library contains two components - the storage of a new version and the retrieval of a specified version. There follows an outline of the kind of facilities required for carrying out both of these activities, firstly listing the requirements and then showing how to satisfy them.

When a new version of a software module is created this can reflect a number of different intentions on the part of the programmer:

(i) a bug-fix - the programmer intends a complete replacement of the old version by the new because the old version is faulty;

(ii) an "extension" - the new version is more powerful or of wider applicability than the old, although the old may still be adequate for some users;

(iii) a new alternative version - this is not intended to replace the old version, but to offer the user a choice (for instance of using a new editor).

The programmer who wishes to use a module in a library may wish to choose:

(a) to retain the original version irrespective of whatever versions are offered;

(b) all changes to permeate through to the application;

(c) to specify a particular version when writing the code;

(d) to accept only bug-fix updates, but otherwise retain the original version;

(e) to choose the version when the calling module is stored;

or (f) to allow the user to choose the version when the application is started.

The system will cater for all of these requirements. Neither of these lists can be seen as complete, but a demonstration of these options generates confidence that the system could be extended to cater for any other storage or retrieval alternatives.

8.3.2 The Storage of Modules.

There follows an outline of a system which provides all of these facilities. To start with, consider a two-dimensional taxonomy of versions in which each module in the system can exist as a number of **alternatives**, which are essentially different methods of implementing the module. Each of these alternatives may exist in a number of sequential **generations**, which are different procedures attempting to implement the same module in the same way. In the following the word "version" is used informally, while alternative and generation always mean the above. The following insertion strategy is adopted, in which one of three conditions can hold:

a) it is a new **module** - a new entry in the library is created;

b) it is a new **alternative** - a new version node is created, which is inserted into a list of alternatives;

c) it is a new **generation** of an already existing alternative - the new generation is placed at the head of the list of generations of this alternative.

The simplest PS-algol code to do each of these is given in the three parts of Figure 8.6 for the case in which a string editor is to be inserted. All three would be preceded by the code given in Figure 8.5.

```
let editor = proc( string Xin -> string )
        .....                                   ! Code which implements the editor.
structure edPack( proc(string -> string) anEditor)   ! Packaging for an editor.
structure modVersion( string Aname, Gnumb, Reason;      ! Packaging for a version of
                      pntr thisVers, lastAlt, lastGen )    !  some unspecified object.
let lib = open.database( "Procedures", "Library", "write" ) ! Find the library.
```

Figure 8.5 Initial Code for Creating a Module Instance.

All three approaches use two structures: *edPack* is a package for an editing procedure, similar to the *minpack* structure already seen; *modVersion* is a structure to hold a single version of an object in the two-dimensional object space. This structure

contains an alternative name and a generation number as identifiers, a reason for the new version, a pointer to the current packaged procedure, *thisVers*, and pointers to the next node in the lists of alternatives, *lastAlt*, and generations, *lastGen*. Note that this same structure could be used for version management of any kind of object as it does not define the type of the object pointed to by the *thisVers* field. This means that the version management code is sufficiently polymorphic that it could be used to manage CAD data, for example. Such polymorphism is a direct consequence of the curtailment of eager type matching on encountering an object of type, **pntr**. This allows systems to be composed out of independent components and libraries of those components to include such things as a choice of version managers.

```
let firstAlt =                                  ! Create a first alternative.
    modVersion("first", "1", edPack(editor), nil, nil )
s.enter( "editor", firstAlt, lib )              ! Store the first alternative.
if commit() =nil                                ! Commit the change to the library.
            then write "Procedure edit stored'n"
            else write "Commit fails'n"
```

(a) Inserting the first version of a module.

```
let oldAlt = s.lookup( "editor", lib )          ! Find the last inserted editor alternative.
let newAlt =                                    ! Create new alternative with a link to the old one.
        modVersion("second",  "1", edPack(editor), oldAlt, nil )
s.enter( "editor", newAlt, lib )                ! Replace the table reference with new one.
if commit() = nil                               ! Commit the change to the library.
            then write "Procedure edit stored'n"
            else write "Commit fails'n"
```

(b) Inserting the second alternative of a module.

```
let oldGen:= s.lookup( "editor", lib )          ! Find the latest version of "second".
while oldGen( Aname ) ~= "second" do oldGen := oldGen( lastAlt )
let newGen =modVersion( "second",               ! Create new version of "second" with
            succ( oldGen( Gnumb) ),             !   next generation number, the new
            edPack(editor),                     !   generation, a link to the same last
            oldGen ( lastAlt ),                 !   version and a link to the last
            oldGen )                            !   generation.
if oldGen= s.lookup( "editor", lib )
        then s.enter( "editor", newGen, lib )   ! Was first in list.
        else .....                              ! List processing to put it in place.
if commit() = nil                               ! Commit the change to the library.
            then write "Procedure edit stored'n"
            else write "Commit fails'n"
```

(c) Inserting a new generation of the second alternative of a module.

Figure 8.6 Three Different Ways to Create Module Versions.

Figure 8.6(a) shows a new module being created as a *modVersion* structure containing the packaged procedure, an alternative name, "first", a generation number, 1, and **nil** pointers indicating that there are no other alternatives or generations. This is then inserted into the system library. Note that while the diagrams show a simple scheme in which the "root" version of the module has the same structure as all the others, in the implementation proper there is another structure which will contain a "generic" version of the module.

Figure 8.6(b) shows the insertion of a new alternative, named "second". The last alternative is retrieved from the system library and, when the new *modVersion* object is created, it includes a link to the old alternative as one of its fields. This links the alternatives together in a list.

Figure 8.6(c) shows the insertion of a new generation of "second". Now the procedure must not only retrieve the first version, but also scan down the list of alternatives to find the "second" alternative. The new module object now has references to both the last alternative and to the last generation and is inserted in its correct place in the list of alternatives.

What has been demonstrated here is a general purpose module version insertion strategy. It could be used to store versions of any data structure, but here, because procedures are first-class objects and because the **pntr** type can be used to delay type-checking, the strategy is used to make insertions into a software library containing procedures of a variety of types.

8.3.3 The Retrieval of Modules.

Turning to the method of binding to stored procedures, Figures 8.8-8.13 indicate six of the many ways to do this, for the case making a call to a string editor in procedure called *caller*. These examples must be preceded by the code in Figure 8.7 and may be succeeded by some code to store the *caller* procedure (similar to the examples above) or by immediate use of *caller*.

```
structure edPack( proc(string -> string) anEditor
structure modVersion( string Aname, Gnumb, Reason;
                              pntr thisVers, lastAlt, lastGen )
let lib = open.database( "Procedures", "Library", "write" )
```

Figure 8.7 Initial Code for Retrieving a Procedure.

Retaining the original version is simple to do in a persistent system (see Figure 8.8). The latest editor will be bound into *caller* and the dependency created can never be broken by anyone other than the application programmer, by re-running this program. The creation of new versions of editors will not affect this binding. Not even "deleting" the version of the editor from the database would remove the called procedure or the link to it. It would instead be retained as it is still reachable, because of the accessibility rules of PS-algol

```
let myEdPack = s.lookup( "editor", lib )
let myEditor =myEdPack(thisVers)(anEditor)      ! Retrieval performed once only.
let caller = proc( ... )
    begin
        .....
        newString := myEditor( oldString )      ! Usage.
        .....
    end
```

Figure 8.8 Always use the first version found.

To make changes to the implementation of the editor permeate through, it is necessary to move the *lookup* of the editor into the body of *caller*, as in Figure 8.9. Now the editor is bound into *caller* each time it is called. This has the effect of always using the latest version inserted into the library.

```
let caller = proc( ... )
 begin
     let myEdPack = s.lookup( "editor", lib )
     let myEditor=myEdPack(thisVers)(anEditor) ! Retrieval performed every time.
     .......
     newString := myEditor( oldString )              ! Usage.
     .......
 end
```

Figure 8.9 Always use the latest version.

To specify the alternative required, a utility *findNamedAlt* could be provided, which, given the module name and alternative name, could look up the latest generation of that alternative. Figure 8.10 shows how this utility could be used to specify the alternative once and for all. The call to *findNamedAlt* could be moved inside *caller* to bind dynamically to the latest generation of the named alternative.

```
let myEdPack=findNamedAlt("editor","myver")
let myEditor=myEdPack(thisVers)( anEditor )      ! Retrieval performed only once.
let caller = proc( ... )
 begin
     .......
     newString := myEditor( oldString )      ! Usage.
     .......
 end
```

Figure 8.10 Using the alternative "myver".

A more complex retrieval is to fix the alternative required, but only to take later versions if they are bug-fixes. This is shown in Figure 8.11. Here, an initial alternative of the editor is selected when the procedure is entered. Each time it is run, the latest generation of that alternative is retrieved and replaces the original if it is a bug-fix. The unpacking is now also done inside the procedure, in case there has been a change.

```
let myEdPack = s.lookup( "editor", lib )                ! Get original version and
let oldName = myEdPack( Aname )                         !     store which alternative.
let caller = proc( ... )
 begin
     let newEdPack = findNamedAlt( "editor", oldName ) ! Retrieve latest generation.
     if newEdPack(Reason) = "bugfix" do                 ! Replace if bug-fix, but not
                          myEdPack := newEdPack         !    otherwise.
     let myEditor =myEdPack(thisVers )(anEditor )       ! Unpacking.
     .......
     newString := myEditor( oldString )                 ! Usage.
     .......
 end
```

Figure 8.11 Using a bug-fix of original version.

To leave the choice of which alternative of the editor to use until the procedure *caller* is stored, there is a generic utility, *paraChoice*, which is called instead of *s.lookup*. If there is only one alternative, it returns the latest generation of it, but if there is more than one alternative it builds a menu of the alternative names and gets the programmer to choose one, as shown in Figure 8.12.

```
let myEdPack = paraChoice( "editor" )          ! Provides menu of alternatives.
let myEditor=myEdPack( thisVers )( anEditor )  ! Retrieval.
let caller = proc( ... )
 begin
     .......
     newString := myEditor( oldString )        ! Usage.
     .......
 end
```

Figure 8.12 Choose alternative at commit time.

The same utility could also be used to pass the choice onto the user by moving the call inside the *caller* procedure. Figure 8.13 shows a retrieval which gives the user a menu of alternatives every time *caller* is called. Another alternative is to insert a little more code which presents the menu only for the first call after the application is started up.

```
let caller = proc( ... )
 begin
     let myEdPack = paraChoice( "editor" )     ! Provides menu of alternatives
                                               !    whenever called.
     let myEditor=myEdPack(thisVers)(anEditor)
     ......
     newString := myEditor( oldString )        ! Usage.
     .......
 end
```

Figure 8.13 Let the user choose the alternative.

These constitute some of the ways a called module may be bound into a caller in PS-algol. There are many others which are essentially combinations of the above and provide no more difficulty for the implementer. Use has been made of the ability to manipulate first-class procedures again, but also of the fact that the moment at which the two procedures are bound together can be controlled. Further, use has been made of the ability to write procedures like *paraChoice*, which can be bound to dynamically changing data - in this case the list of alternatives of a given module. Finally the strong typing of PS-algol has been used. Despite the degree of flexibility achieved, type unsafe operations are not permitted anywhere. When a retrieved procedure is used in Figs. 8.8 to 8.13, the programmer can be absolutely sure that it is of type **proc(string-> string)** and not some other type.

8.3.4 Language Extensions to Simplify Version Management.

The flexibility of PS-algol is, however, somewhat offset by the nature of the code required to generate the different binding styles. It is somewhat verbose and, in the

previous examples, obscures the programmer's intention. The proposed system makes the programmer's requirements explicit and allows the programmer to create module source files which have clear instructions to the system on what binding is required. Such a source file is taken and translated into a pure PS-algol program of the forms shown in Figures 8.5 to 8.13, which is then submitted to the compiler.

In the case of storing a module, the programmer needs to specify: where the module is to be stored (i.e. in the system library on in the space associated with some application); some identification of the module and the version; and which of the three operations in Figure 8.6 is required. To do this the programmer will be allowed the syntax:

```
"save"  ( "systemlib" | applicationname ) modulename
                versionname ("new" | "newversion" | "newgeneration")
```

A line of this form will be placed at the end of the module and this will be transformed into the code shown in Figures 8.5 and 8.6. The three examples are written:

(a) **save** *systemlib editor first* **new**
(b) **save** *systemlib editor second* **newversion**
(c) **save** *systemlib editor second* **newgeneration**

For retrieval, far more can be specified. The programmer needs to specify: where the module is stored; which module; and which version. The last of these will specify which of the binding styles is used. It can be done with a line of the form:

```
"retrieve" ( "systemlib" | applicationname ) modulename
        ( "fixed" | " latest" | "bugfix" | "preferred" | "Ichoose" |
        "userchoose" ( "firsttime" | "everytime" ) |
        ( "/" versionname ( "fixed" | "latest" | "bugfix" | "preferred" |
                        ( "/" generationnumber ) ) ) )
```

at the start of the source file which will be transformed into the code shown in the Figures 8.8 to 8.13, which can now be written, respectively, as:

8.8: **retrieve** *systemlib editor* **fixed**
8.9: **retrieve** *systemlib editor* **latest**
8.10: **retrieve** *systemlib editor /myver* **fixed**
8.11: **retrieve** *systemlib editor* **bugfix**
8.12: **retrieve** *systemlib editor* **Ichoose**
8.13: **retrieve** *systemlib editor* **userchoose everytime**

The techniques used in the version control system can be incorporated in a set of programs to give general support to the applications programmer.

8.3.5 System Implementation - the Objects.

The objects manipulated by the system and their implementations are described in detail. A **module** contains the following: a name by which it can be referenced; a

description of its purpose; and a reference to a list of versions. This is represented by the following structure:

```
structure module(       string moduleName, description ;
                        pntr dynamicCallers, versions )
```

where the *dynamicCallers* field points to modules which are dynamically bound to this one and so call no specific instance.

A **module version** consists of:

- an alternative name;
- a link to all other alternatives which exist;
- a generation number, the range of which is left unspecified and may vary from implementation to implementation;
- a link to all other generations of this alternative;
- the reason for storing this version;
- a set of references to the module versions this one calls;
- a set of references to other module versions statically bound to this one;
- the names of the types of the arguments of the version;
- the name of the type of the result of the version (following PS-algol in only permitting a single result for the purposes of this discussion - extending to the more general case introduces no new problems);
- the program source which defines the version;
- the procedure which implements the version; and
- sundry documentation material.

Then the structure to contain a module version has the form:

```
structure moduleVersion(
        string versionName;
        pntr lastAlternative;           ! versions held in a
        pntr nextAlternative;           !    doubly-linked list
        string geneNumber;              ! string preferred to int
        pntr lastGeneration;            ! generations also held in a
        pntr nextGeneration;            !    doubly-linked list
        pntr called;                    ! a list of modules called by this one
        pntr staticCallers;             ! a list of modules statically bound
                                        !    to this one
        string imports;                 ! e.g. "int a,b"
        string export;                  ! the type of the result
        string theSource;
        integer storeTime;              ! time the module was stored
        string author;
        string updateReason;            ! e.g. "bugfix"
        pntr packagedProc )             ! points to packaged proc.
```

The last pointer will be to a structure which contains a single field to hold the procedure. The names of the structure and the field can be generated automatically from the parameter types of the procedure, for instance *minvec* would use the following:

```
structure VintTintprocpack(proc(*int->int) VintTintproc )
```

A **module source** will be a string, which in the case of *minvec*, would be as shown in Figure 8.14.

```
retrieve systemlib min latest
let minvec = proc( *int V -> int )
 begin
     let smallest := V( lwb(V) )
     for i = lwb(V)+1 to upb(V) do smallest := min( smallest, V(i) )
     smallest
 end
save systemlib minvec firstgo new
Richard Cooper
Returns the smallest of a vector of integers.
```

Figure 8.14 A Source Module for *minvec*.

More formally, the module source consists of:

- one or more lines retrieving versions of modules to be used from the system library or from the application library;
- the procedure body implementing the module version being defined;
- a line saving the module version in the system library or application;
- a line with the author's name; and
- one or more lines of description.

The **system library** is a set of modules, none of which calls modules outside of the system library. This is implemented as a PS-algol database with name "Procedures" and password "Library". New modules will be entered in the top-level table of the database. Updates will be linked via the various fields of the module structures.

An **application** is a set of modules and data held together. None of the modules call modules which are not in the application itself or the system library. There is one distinguished **root** module, of type **proc()**, which is not called by other modules but by directly invoking the application. The application is also stored in a PS-algol database. One entry in the top-level table will point to a table containing the application library, which will be organised in the same way as the system library. The root module is contained in this table along with the others, and is keyed in the table with the name "root". This contrasts with the usual PS-algol approach in which the root module is a compiled PS-algol program which initiates the application. All the data connected with the application will be stored in structures reachable from the other entries in the top-level table.

8.3.6 System Implementation - the Operations.

There follows an outline of the implementation of two operations provided by the system, one of which runs an application and the other stores a module version. There is also a description of the design of three others as yet unimplemented.

To **run an application**, issue the command:

runapp *applicationName*

and this runs the simple program shown in Figure 8.15.

```
let AppDB = open.database( applicationName, "friend", "read" )
let lib = s.lookup( "library", AppDB )
structure voidprocpack( proc() voidproc )
let theApplication = s.lookup( "root", lib )( voidproc )
theApplication()
```

Figure 8.15 The *runapp* facility.

It finds and unpacks the root procedure and then executes it.

To **store a module**, use the following command:

store *moduleSource applicationName*

where *applicationName* can be "systemlibrary". This will take in the module source and translate it into a PS-algol program similar to those shown for storing *min* and *minvec*. The program that implements this lies at the heart of the system and deserves close attention.

It relies on an extension of the PS-algol parser, which analyses the source module in the form presented above. Using a system like that described in section 4.4, such a parser can be created, which returns a tree of lexeme, token pairs. This is used as follows:

1) Divide the module source into six parts: the retrievals; the procedure definition; the body of the procedure; the storage line; the author's name; and the description.

2) Scan the retrievals and build *open.database* calls to the system library and application databases as required, as a string *ODB*, say.

3) For each retrieval, build the appropriate code to load the correct module. This code relies on: a packaging structure which can be built from the type names found in the *imports* and *export* fields; calls to standard procedures, like *findNamedAlt* and *paraChoice* - these will be already in the system library and so must be themselves retrieved; the final parameter which will determine whether the retrieval should occur outside or inside the procedure - two strings *staticCalls* and *dynamicCalls* will be built up for these two cases.

4) The required PS-algol source to this point is *ODB* followed by *staticCalls* followed by the procedure definition followed by *dynamicCalls* and the procedure body.

5) The list of modules called by this one is created. The list contains a pointer to the called module if the bind is dynamic or to a module instance if it is static. The list is scanned to create backwards links - a general function, *makeBackwardLinks*, does this systematically.

6) a) If this is a new module, build a module version object with **nil** fields for the version and generation lists and "new" for the *updateReason* field. Get the initial value of the generation number from a system function, *firstGen* - this allows different systems to have different generation numbering mechanisms. Create a new module object, make it point to the version object and put it into the table.

b) If it is a new alternative, look up the module object and create the new version object so that it is linked into the list of alternatives.

c) If it is a new generation, lookup the module and then find the alternative. Create a new module instance to represent the new generation and then link it into the list of generations - the generation number will be determined by using another system function *succGen*, to generate it.

7) Fill in the other fields of the module version from the appropriate sources.

8) Put the **commit** line in.

```
let lib = open.database( "Procedure", "Library", "write" )
structure intintTintprocpack( proc( int,int -> int ) intintTintproc )
let minvec = proc( *int V -> int )
 begin
     let min =s.lookup( "min", lib )(versions ) (intintTintproc )     ! Latest version always.
     let smallest := V( lwb(V) )
     for i = lwb(V) to upb(V) do smallest := min( smallest, V(i) )
     smallest
 end

! show that minvec calls min
structure callList(pntr call; string bindType ; pntr callNext )
let calledProcs =callList(s.lookup( "min", lib ), "latest", nil)
 makeBackwardLinks( minvec, calledProcs )

structure VintTintprocpack( proc( *int -> int ) VintTintproc )
structure module( ....
structure moduleVersion(....
let version = moduleVersion(          "firstgo", nil, nil, 1, nil, nil,
                                      calledProcs, nil, "*int V", "int",
                                      "retrieve .....integers.",      !i.e. the whole source.
                                      date(),                         ! System function.
                                      "Richard Cooper", "new",
                                      VintTintprocpack( minvec ) )
let Module = module ( "minvec", "Returns smallest of a vector of integers.", version )
s.enter( "minvec", lib, Module )
if commit() = nil      then write "New module minvec stored ok'n"
                       else write "Commit of new module minvec failed'n"
```

Figure 8.16 Automatically Generated Module Storage.

This is now a complete PS-algol program to place the module appropriately. The PS-algol compiler can be called to compile this and then the program can be run.

As a final illustration, the module source given in Figure 8.14 for *minvec* would be translated into the pure PS-algol code given in Figure 8.16.

Display an application/the system library. The graph of modules in an application will be shown to an arbitrary degree of detail. Among the facilities that may be provided are: scrolling about the MDG; traversing the versions of the modules; and zooming in on the imports/exports which flow along the dependencies or on the details of the modules as required.

Retrieve module details. Given a search string, any modules having that string as part of its name, version name or some subset of the documentation fields will be returned for examination. The search can be confined either to the system library or to an application together with the system library.

Produce program documentation. The application MDG will be processed in a systematic way and the documentation fields will be transformed into a documenting text.

8.4 Conclusions.

This chapter has shown how to build a set of tools to manipulate a procedure database in order to assist in the construction of application programs. This has been possible in a persistent language because the fine detail of storage has been removed from the programmer's concern. The provision of first-class procedures and the extensible union type **pntr** have been the major aids. Programs could be written which manipulate the procedures which instantiate modules with the same ease as any other data item and there is access to the compiler at run-time. Further, by making references to the stored procedures via **pntr** fields, generic programs could be written which handle any kind of procedure, although use of any of these will be fully type-checked. Where specific types were required, these were automatically constructed in systematically named structure classes. Furthermore, the ability to control binding times has been used to create a module version control system in which the bindings between modules can be of a number of different forms.

There is no attempt to justify the nature of the tools constructed - it is the business of software engineers to design the kind of tools which they require. However, when the tools have been designed, they can be implemented in a language like PS-algol. The approach here has some similarity with the object-oriented approaches of [Dittrich *et al.*, 1986; Zdonik, 1986], rather than the traditional file-oriented [Rochkind, 1976; Tichy, 1985] or record-oriented [Ecklund *et al.*, 1987] approaches. This demonstration begins to justify the expectations of [Nestor, 1986] in the use of Persistent Object Stores and to overcome many of the problems encountered - for instance, Zdonik's statement: "There is hardly a programming language to perform the operations".

In particular, this is a system which is both integrated (all of the modules live in the very consistent regime of the Persistent Store) and open [Nestor, 1986]. Openness is provided both in the sense that new modules of any type can be added to the system indefinitely without compromising system integrity and in the sense that new tools can be provided to manipulate the module base, without invalidating existing tools.

There are a number of restrictive assumptions in the system as described. For a start, it is assumed that each module source produces exactly one procedure. There is no intrinsic reason why a module should not store more than one procedure. Nor is there any reason not to use the same mechanism to store other kinds of object. The data-type completeness of PS-algol means that any type of object can be manipulated with the same ease as any other. Thus future work is envisaged implementing systems like those described in [Katz and Chang, 1987]. Restricting the number of procedure exports to one has already been referred to – this was merely to ease the transformation into PS-algol. All of these restrictions were made to simplify the work by excluding extraneous issues.

Work is already under way to implement this system. The implementation is taking two paths. A direct implementation in PS-algol of an Automated Interactive Module Management System, which provides the facilities described here [Kerr and Cooper, 1989]. The storage and retrieval of modules containing any number of objects of any type has been implemented, but the implementation of the document and display facilities has yet to be tackled. There is also to be an indirect implementation using the requirements modelling language, PSRML, see section 7.2. In this implementation, the objects and operations will be specified in a very high level language which should automatically generate the software which implements it.

Chapter 9. A Methodology for Persistent Programming.

This chapter pulls together the implementation techniques used in the preceding chapters. The concentration is on an overall methodology for developing programs within the PS-algol language, outlining strategies for generating suites of application programs using higher level models of the application domain and describing the way in which specific features of the language are used. This methodology will support an incremental development of applications and follow iterative cycles of design and implementation (as shown in Figure 9.1), rather than a purely linear sequence of design and implementation phases, rarely usable in practice. The figure illustrates a design process with three cycles, an inner cycle depicting the trouble shooting of a particular level of code, a second cycle depicting a successive refinement and addition of detail and finally the process of re-assessing the requirements of a piece of software once it "works". The steps in the methodology described in this chapter are also subject to these cycles of design and implementation, so that although the steps are numbered sequentially, it is a particular strength of the persistence paradigm that they can be taken in any order and freely mixed.

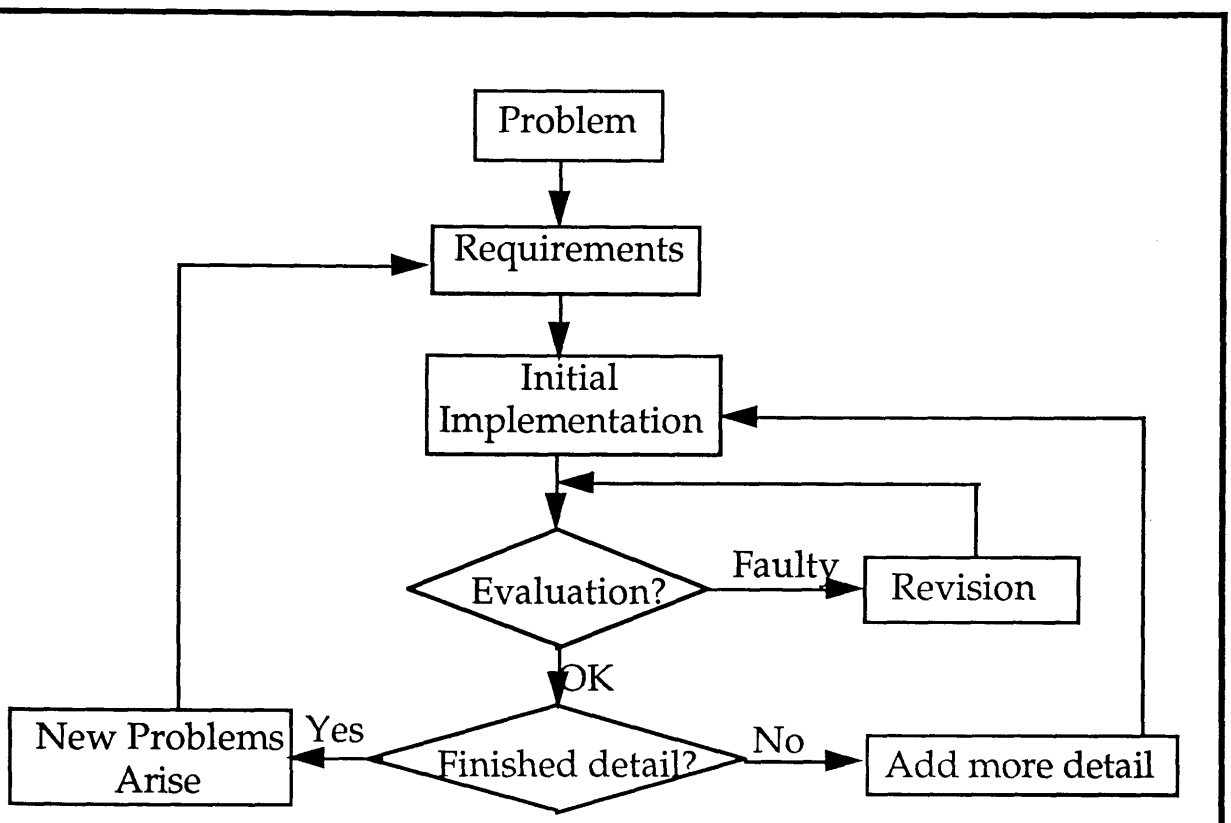


Figure 9.1 An Incremental Design Process.

The basic building blocks of the methodology will be the following features of PS-algol:

- **orthogonal persistence** and the ability to store the values of the application domain in the same form as the structures used during processes;

- PS-algol **structures** and their ability to model objects of arbitrary complexity with fields of any type;

- the **pntr** type and its use to defer the binding of program to data and to delay the type checking of general purpose code until run-time;

- the **run-time compiler** and its use to tailor code appropriate to a wide variety of data classes;

- the **graphics** primitives and their use for creating the user interface.

Perhaps the most important point to be made at the outset of this chapter, however, is that throughout this chapter, there will appear alternative development paths. This is because the ethos behind PS-algol is to facilitate all kinds of programming technique and not to impose a particular programming style. For instance, there is an assignment statement in PS-algol, but no attempt to force the programmer to use it. As PS-algol has first-class procedures, one alternative is to use a purely functional style of programming. Conversely, if one wishes to use a purely object-oriented style, this can be achieved without departing from PS-algol [Philbrow *et al.*, 1989]. In short, PS-algol does not make decisions for the programmer, but provides sufficient primitives to enable different approaches to the problem.

The first part of the methodology is to specify the data structures of the objects used in the program using the PS-algol structures, as described in section 9.1. The step includes specifying attributes of the objects and then other kinds of inter-object relationships. Next, in section 9.2, the program design is discussed. It is argued that in general, all code modules can be split into three sets: operations on object classes, as supported in Object-Oriented systems; modules which are themselves the components of objects (recall the action procedure associated with a light button); and modules which stand outside of the object space, for instance the module which starts an application. Following this, section 9.3 describes polymorphic programming; 9.4 describes organising the persistent store; and 9.5 describes organising the user interface. The chapter concludes with some discussion of the weaknesses of PS-algol as a programming language, with suggestions for improvement.

9.1 Program Specification and Data Modelling.

9.1.1 Modelling Simple Data Attributes.

The first step in developing a new application suite is to specify the kinds of object which it will manipulate. This step, akin to data modelling, is of general applicability to all large-scale software development whether it be data intensive or "systems" programming, such as compiler writing. The **structure** constructor of PS-algol, which permits objects of any type to be components of a complex object, provides a good device with which to express the objects of the application domain.

Thus, in developing the Bibliographic Database System (Chapter 5), a structure type was introduced for each of the following: references, reference types, output media, formats and abbreviations. In a compiler, structures can be created for lexemes, lexical analysers, nodes in a parse tree, and so on. A large range of the objects of the

system from the trivial to the complex can be expressed in a consistent way, providing a complete description of the data structure of the application domain. Moreover, the structures may include procedures as components and so they can also provide a specification of the dynamic features of the world, which are specific to a particular class of objects. An output medium in the BRDB, for instance, contains procedures to process a set of references for output in particular ways. The structure definition of an output medium specifies the types of the procedures involved.

However, initial concentration will be on determining the simple attributes of each type. These will include any textual, graphical, numerical or boolean attributes, including vectors of any of these and a structure definition will be constructed which contains these fields alone.

One aspect to be remembered when defining these structures is the degree to which they are extensible. If there is, for example an *address* structure as follows:

structure *address*(**int** *house*; **string** *street, city*)

and later, a field for *postcode* is required, there are two ways to cater for this eventuality. If the program is under development and no real data has yet been developed, a *postcode* can be added and all programs using the structure must be edited and re-compiled. Even if there is some hard data, this may be the correct plan. In addition there will be the creation of a program to copy all of the data from the old structure to the new one. Note that this is a complex task in a reference based language, since all references to *address* objects will also have to be changed. A separate technique which exploits the **pntr** type is to plan for the eventuality by adding an *extension* field, as follows:

structure *address*(**int** *house*; **string** *street, city*; **pntr** *addressExtension*)

When the postcode is required, a new structure is created:

structure *address2*(**string** *postcode*; **pntr** *addressExtension2*)

which will contain the postcodes (and incidentally leaves room for further expansion).

The first steps in developing an application, then, are as follows:

1 Identify the objects of the program and their simple components.

2 Write specifications for them as PS-algol structures.

9.1.2 Graph-based Programming.

The next step consists of specifying the inter-relations between objects. In any application domain, various kinds of inter-relations between object classes can be represented using the same PS-algol mechanism, a **pntr** field in a structure. This mechanism, being polymorphic in the sense discussed in section 9.3, is capable of representing any links between two structures. Thus a **pntr** field can represent a specific link, like that between a reference and a reference type in the BRDB, or a more

general link, like an inheritance link between two types, as in IFO, PSRML, EFDM or MINOO.

In fact, the increase in generality is a consequence of the higher-level nature of the program - the programming is of the same order of complexity. The data modelling programs manipulate object types, while the BRDB manipulates references. The relationships to be modelled will include all of the following:

- the relationship between an object and a component where that component is itself a complex object;
- the relationship between an object and a complex attribute;
- the relationship between one object and similar objects, when these all belong to a structured set object (such as a list, a tree or a table);
- the relationship between an object and other objects which all belong to a particular sort of graph (such as a type-graph).

In fact, the whole of the programming world consists of a graph of objects, where the arcs represent these various kinds of relationships. This graph of objects is precisely the graph which the PS-algol browser navigates through (see section 4.2).

Having identified the inter-relations between objects, the structures for each of the object classes must be extended to include **pntr** fields to represent these relationships. For instance, the structure for a bibliographic reference is extended to include a **pntr** to a reference type. Here a deficiency of the PS-algol language is found in that there is no way of asserting that this field must point to a structure containing a reference type. This matter will be discussed in section 9.6.2. Annotation recording these referend types is recommended and procedures will be developed to manipulate them to comply with constraints.

Two more steps in the methodology are:

3 Identify all inter-object class relationships.

4 Extend structures for object classes with **pntr** fields to represent these relationships, either referring directly to the related object or referring to a data structure that groups related objects.

9.2 Starting the Program Design.

When a cluster of data structures has been defined, it is possible to write code to manipulate them. When there is a description of the data structures of the application, a start can be made in constructing the code. Two things need to be done. Firstly, a modular design of the program must be produced in the standard way and, secondly, a decision must be made on how best to place the individual modules with respect to the object graph. Having first-class procedures, but not being tied to an Object-Oriented style of programming, there is freedom to place a module in one of three ways.

- It may be associated with an object class as an operation factored out of the instances of that type and grouped together with other operations to form an Abstract Data Type.

- It may be an attribute or component of an object instance.

- It may be an object in its own right - part of a separate program graph, using a traditional programming style.

The next step will therefore be:

5. Divide the program into these three methods of implementation.

9.2.1 Providing Abstract Data Types and Object-Oriented Systems.

Given an object class for which the operations are well specified, it is possible to restrict access to the data structure to those operations. The code in Figure 9.2 shows how to restrict access to objects in class *S*,to two operations: *O1*, which has an additional string parameter; and *O2* , which returns an integer as a result. Notice firstly, that as all object references are passed via the abstract data type, there is now no need to wrap the state variables into a structure. These have been left as individual variables, thus speeding the implementation.

```
let makeAnS = proc( -> pntr )
    begin
        let V1 := ...                          ! Initial values for the
        let V2 = ...
        .....                                  !    hidden state variables.
        let anO1 = proc( string P )
            begin
                ... code uses P and the Vi's
            end
        let anO2 = proc( -> int )
            begin
                ... code uses VI's
            end
        structure SADT( proc( string ) O1; proc( -> int ) O2 )
        SADT( anO1, anO2 )                     ! Visible operations.
    end
```

Figure 9.2 An Abstract Data Type.

An object of class *S* can be created by the following:

let *anS* := *makeAnS*( )

and then all that can be done with it is to apply its two operations, as in:

anS(*O1*)("HELLO")
and **let** *anInt* := *anS*(*O2*)( )

There are many circumstances in which an ADT of this form could be created - the relational interface of GRAPE (section 6.3) gives one example of this. However, the mechanism should be used with care. If it is used to describe class *S*, and subsequently a database is populated with many instances of *S* and the class needs to be changed, there will be major problems. The data can only be accessed by the operations provided - there is no convenient back-door and therefore no way of changing the class of already existing objects.

Therefore, if a change of class description (such as adding a new operation) is likely, there must be operations which retrieve all of the data associated with an instance. These can then be used to get out all of the data and restructure the object. Adding a third operation to retrieve a description of the initial specification of *S* will provide this. Now, when an operation *O3* is to be added to the class, another operation to take in this description of the state is also added. This permits:

```
let anS := makeAnS( )
let Sdescription := anS( retrieveDescription )( )
let anS2 := makeAnS2( )
anS2( storeDescription )( Sdescription )
```

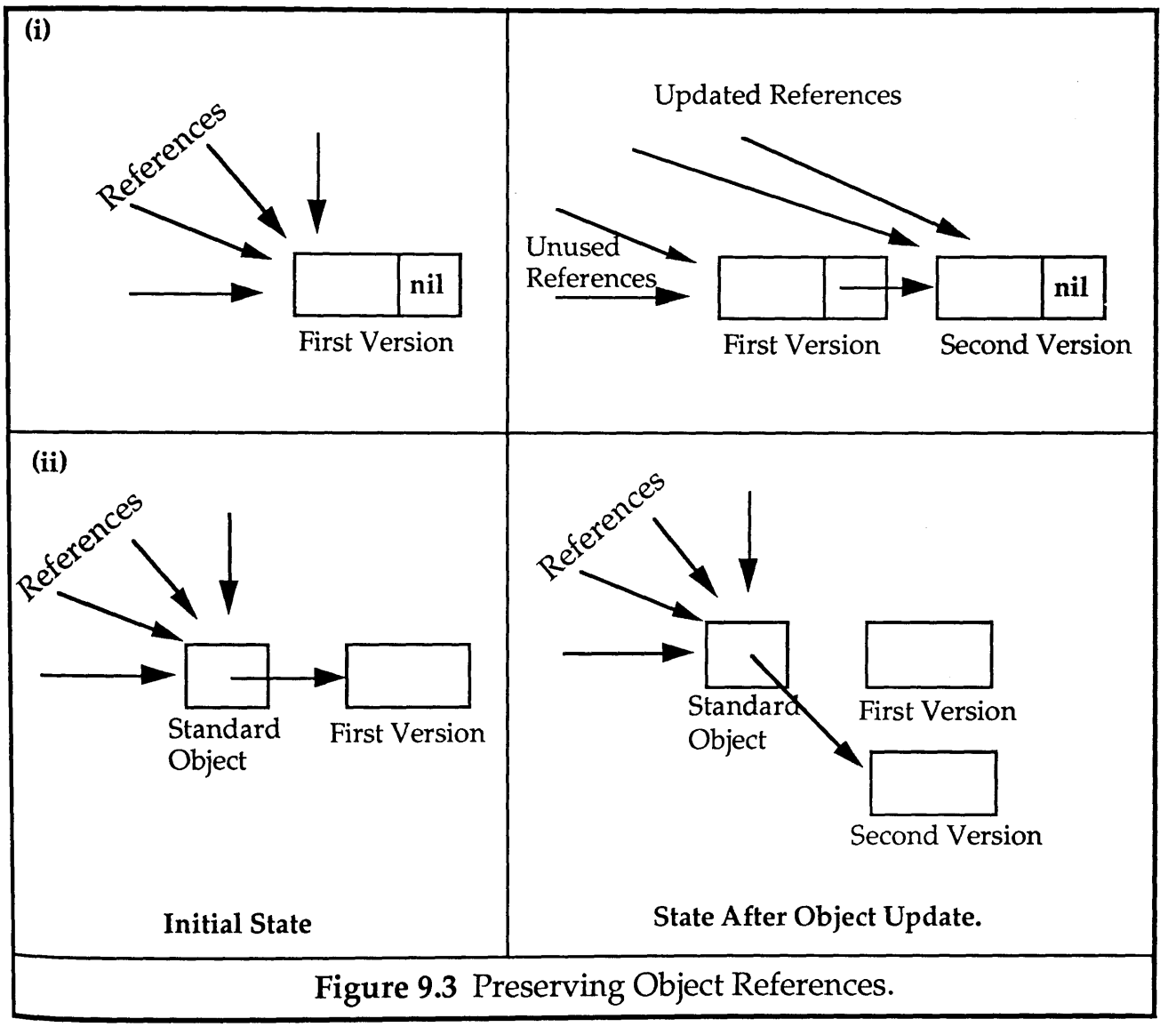


Figure 9.3 Preserving Object References.

This will then make *anS2* be an object of the newly constructed class with the same state as *anS1*.

None of this avoids another problem which arises from the previous code fragment - shifting references to objects when their structure changes. Two possibilities are outlined here and shown in Figure 9.3.

i) To every object add a forward reference field, initially **nil**, but when an object changes its representation, the field will point to the new version. All object references check this field, which will be copied back to the referend so that it only occurs once per reference. This is at the cost, for every object access, of: a test; sometimes an indirection; and phantom updates.

ii) For every object, keep a unique structure with one pointer field to its representation. Every reference will always be correct as it will point to this structure. The forward reference is changed automatically every time a representation change to the object is made. This is at the cost of an indirection for every object access.

Neither of the possibilities seems superficially attractive. They are intended to permit the reference semantics which accompany models attempting to make the real world representation as simple as possible, whilst permitting object type evolution. If neither is used, when redefinition becomes unavoidable, complex programs have to be written and run; but normal execution involves none of these costs.

```
let makeAnS = proc( -> pntr )
    begin
        structure SADT( string V1; int V2; proc( string ) O1; proc( -> int ) O2 )
        let HV1 := ...                          ! Initial values for the
        let HV2 = ...                           !    hidden state variables.
        let ADT := nil                          !Abstract Data Type Represntation
        let PV1 := ""                           ! Initial values for the
        let PV2 = 0                             !    public state variables.
        let HO1 = proc( string P )              ! Hidden operations.
            begin
                ... code uses P and the Vi's
            end
        let HO2 = proc( -> int )
            begin
                ... code uses VI's
            end
        let PO1 = proc( string P )              ! Public operations.
            begin
                ... code uses P and the Vi's and the HOi's
            end
        let PO2 = proc( -> int )
            begin
                ... code uses VI's and the HOi's
            end

        ADT := SADT( PV1, PV2, PO1, PO2 )       ! Visible components.
        ADT
    end
```

Figure 9.4 A More Generalised Abstract Data Type.

Returning to Figure 9.2, another point to notice is that the code of *makeAnS* consists of a list of definitions, firstly of hidden state variables, secondly of public operations. The choice of what is hidden and what is not was an arbitrary one and, using exactly the same framework, PS-algol provides the opportunity to choose exactly which variables and which operations are public and which are hidden. A more general scheme is presented as Figure 9.4, in which are seen hidden variables (the *HV*'s), public variables (the *PV*'s), hidden operations (the *HO*'s) and public operations (the *HP*'s). This ability to specify with complete orthogonality the public/hidden and attribute/operation dimensions of any class component is a natural consequence of the data-type completeness and subsequently appeared in the superior Object-Oriented systems such as Eiffel [Meyer, 1988].

Thus, the next step of the implementation is:

6. Identify all classes best represented by the generalised Abstract Data Type form shown in Figures 9.3 or 9.4 and implement them, making sure that there are back-door operations and/or indirections to rescue data likely to be trapped in out-of-date class structures and to allow references to be maintained after object replacement.

Notice that it is the responsibility of the application builder to choose the trade-off appropriate for each part of the application. In many systems, such decisions are pre-empted by the language.

9.2.2 Operations as Object Components.

Some parts of the code are best seen as modules which are the dependent parts of objects. The clearest example of this is a light button which is part of a menu or dialogue box. One of the components of a light button is the procedure which is activated when the button is clicked. Note that this is very different from an operation defined over an object-class as described in the previous section. Operations as described in the previous section were generally applicable to all instances of a class. These component operations will vary from object to object. In a given set of light buttons, one will have a component to produce some help information, another will quit the menu, etc. A subsidiary distinction is that a component operation need not necessarily operate on other components of the object at all, while an operation as described above will probably manipulate other aspects of the same instance.

To accommodate these dependent operations, the structure of the class is extended to include procedure fields. When objects of the class are instantiated, these procedure fields must be given values. Consider the structure:

structure *lightButton*(.....**string** *message*; **proc**() *activity*)

which shows the two fields of light button giving the message on the screen and the associated activity. To instantiate a light button, first define a procedure, for instance:

let *helpProc* = **proc**( )
.....

and then instantiate the button with:

let *helpButton* = *lightButton*(... "HELP", *helpProc*)

This shows the procedure being bound separately into each object in the same way that a data value would be bound. Contrast this with the ADT operation, which is provided as a general purpose piece of code for the whole class, bound when the instance generator is defined.

The next steps are therefore identified as:

7. Extend structure classes to include procedure fields where appropriate.

8. Write the particular instance procedures which will populate these fields.

9.2.3 Modular Programming Development and Software Libraries.

In an Object-Oriented system, all of the coding effort is forced into the structure described in section 9.2.1. A Persistent Programming Language such as PS-algol gives more freedom to retain a traditional programming style where appropriate. In PS-algol, any modules not dealt with by the methods described in the previous two sections can be organised in the traditional way, employing a methodology like the one described in Chapter 8. Whereas it has been demonstrated that any module can be constrained into the O-O style, there are many instances which do not fit well. A dummy object may have to be created so that a given module can be installed as its operation or unnatural object classes may be set up to provide a framework for high-level modules. The code which starts off an application is the most obvious example of such stand-alone modules. Starting off an application as described in [Meyer, 1988] requires considerable contortion of the code by the programmer. In general, a module should be viewed as stand-alone if there is no single object class to which it is clearly subordinate.

These stand-alone modules will be organised into a graph of procedures in which the arcs are references to procedures from other code. Each module will be implemented as a PS-algol procedure and stored in the persistent store. For simple programs, it may be sufficient to allow the graph to be implicitly created by the calls of one procedure on another. For a program of any complexity, however, it would be better to make the graph explicit as shown in Chapter 8. Use of a structured program library, version control system and configuration manager is recommended. All of these can be built in PS-algol as described in Chapter 8 and further discussion here would merely repeat that chapter's contents, so this section concludes with the step:

9. Produce a modular specification of the program and code each module as a PS-algol procedure. Manage these using any tools which are available.

9.3 Polymorphic Programming in PS-algol.

For any large scale program, there are usually modules which are required to manipulate a number of different classes of object. It would be costly to produce

manually versions of the module for each different class to be manipulated, rather than to provide procedures which operate on more than one class of data. The justification for wanting to do this is the same as that for introducing loops, arrays and other constructs into programming languages - the wish to achieve optimum code re-use. There is an obvious tension between wishing to do this and requiring the support of a strongly typed language to gain an early detection of type errors.

In PS-algol, the **pntr** type is used to resolve this tension. Essentially, the type system is split into those types, such as **int**, **proc(int -> int)**, etc., which cannot be used polymorphically and the PS-algol classes which can. The implications of this split will be discussed in section 9.6.1.

Although the scalars, vectors, procedures, etc. cannot partake in polymorphic code, the **pntr** type can be used to provide at least four kinds of polymorphism. These kinds will be introduced in the terminology of [Cardelli and Wegner, 1985] as outlined in Figure 9.5. This division, which expands on [Strachey, 1967], firstly divides all polymorphic operations between **universal polymorphism** (meaning that the same function will operate in the same way over an infinite set of types) and **ad hoc polymorphism** (meaning that the same function may operate in quite different ways over a finite set of types). This classification is problematic since it depends on the level of abstraction with which the operation is viewed. For example, consider a generic print routine. Abstractly, it produces an external representation for any object. More specifically, it executes different code depending on the type of each object. Using the callable compiler technology, a given procedure can be viewed either as a single parametrically polymorphic procedure, which works by compiling components at run-time, or as a set of procedures exhibiting *ad hoc* polymorphism.

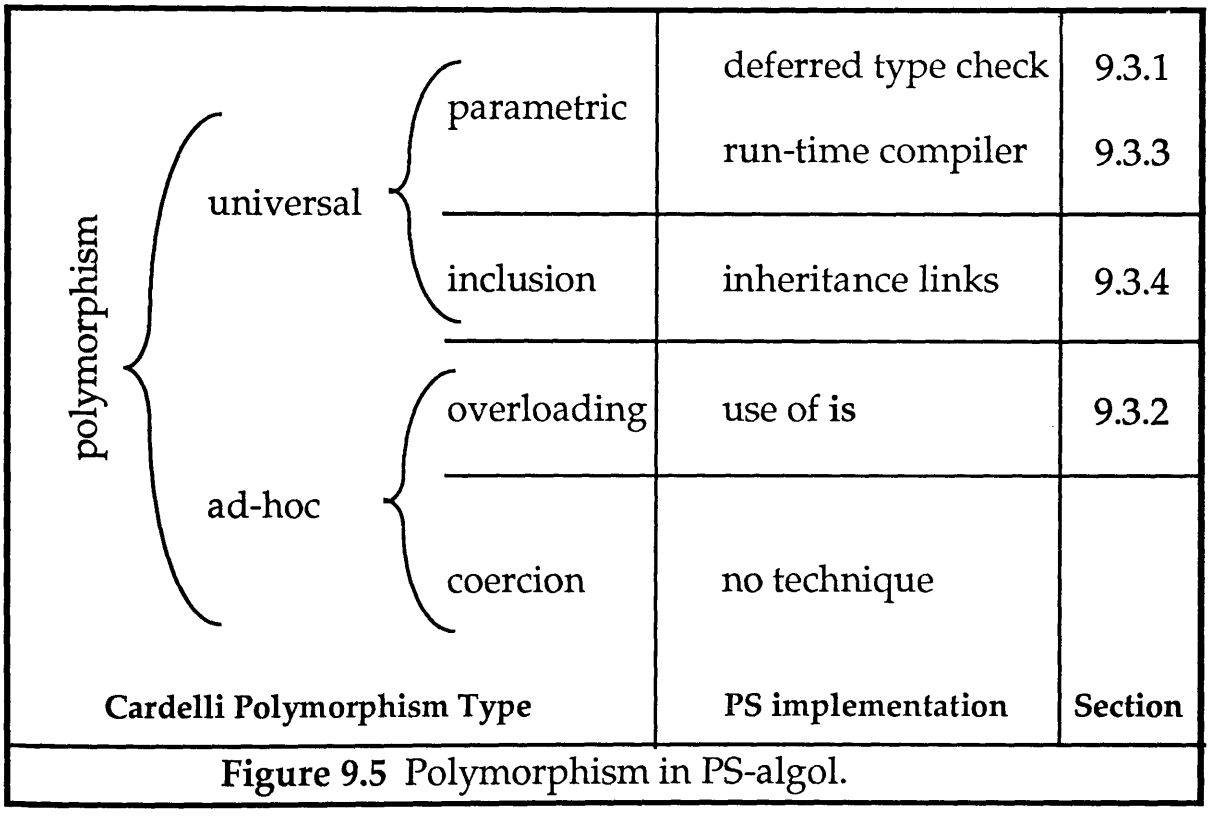


Figure 9.5 Polymorphism in PS-algol.

Universal polymorphism is further divided into **parametric polymorphism**, (meaning that the universal function receives, either explicitly or implicitly, a type parameter to instantiate it) and **inclusion polymorphism** (meaning that a function can

operate over a given type or any sub-type of that type). *Ad hoc* polymorphism can be divided into **overloading** (the same function name refers to different functions in the context of different types) and **coercion** (the objects are automatically translated to conform to the 'equivalent' value in the type appropriate to the function being applied). PS-algol provides mechanisms for providing all of these forms, except the last. In particular, four mechanisms will be discussed:

The first uses the **pntr** type to place a limit on the amount of type checking which needs to be performed at compile time, delaying the type check to run-time instead [Atkinson *et al.*, 1988]. This is appropriate when the procedure is independent of the internal structure of an object. One example of this is a procedure which inserts an object in a list as this does not need to know anything about the type of the object or of the other objects in the list.

The second uses the run-time compiler to generate automatically appropriate procedures for a range of object classes, using the class of the object as an implicit type parameter. This is appropriate when the code to be executed depends on the structure of the object. An example of this is a procedure which prints the objects in a list. In this case, the format of the print and access to the elements of the list depends on their structure.

The third mechanism uses inheritance links to provide inclusion polymorphism. When searching for the operations available on an object, not only the object's class will be checked, but so will all super-classes of this class (by following inheritance links).

Finally, PS-algol provides **is**, which is a type-case check and may be used to implement overloaded procedures.

Each of these mechanisms will now be discussed in turn.

9.3.1 Partitioning the Program Using Deferred Type Checking.

When the program has been specified as in steps 5 to 9, there will usually be some modules which perform some polymorphic operation on an object "blind". They navigate a graph, send objects through some communications channel, etc. Any module such as this can be rewritten polymorphically by using **pntr** references. The tables *package* of PS-algol provides both a relevant example and a counter-example.

A table, in general, is the union of two indexes - one from strings to complex objects, the other from integers to complex objects. Any object to which there is a **pntr** reference can be entered into a table by, for instance:

s.enter("aStringKey", *object*)

where the object can have any structure. Thus, a generalised associative access package has been provided for objects of any structure.

The same package is simultaneously a counter-example, because there are two sets of procedures for manipulating the table, one using string keys and one using

integers. PS-algol has no simple way of unifying these two sets of procedures (see 9.6.1 for more discussion on this point). A compromise could be achieved by making the key a **pntr** reference as well as in:

enter(aPntrKey, object)

However, now the procedure stops being as simply polymorphic as the two procedures *s.enter* and *i.enter*. This is because in order to perform the table insertion correctly, the procedure has to know the values of the key and therefore to look into its type. Moreover, the storage algorithm is likely to be different for the two forms and so will require the *ad hoc* polymorphism discussed in section 9.3.2.

However, for many purposes, there will be the opportunity to delay the type check of parts of an object and this should be exploited in the step:

10 For the operations on a given object, partition the program into parts which can and cannot operate without looking inside the object and code the former as polymorphic code.

9.3.2 Overloading Using "is".

A polymorphic form of the *enter* procedure was introduced in section 9.3.1, for which the internal state of the key was critical. Such a procedure could be written as follows:

```
structure intPack( int intVal )
structure stringPack( string stringVal )
let enter = proc( pntr key, value )
        case true of
                key is intPack:     i.enter( key( intVal ), value )
                key is stringPack:  s.enter( key( stringVal ), value )
                default:            ... error code
```

with further key types being explicitly added at will. This procedure exhibits *ad hoc* polymorphism via an overloading of the *enter* procedure. In PS-algol, this technique can be used to glue together two or more equivalent procedures containing different code for different classes.

The payoff here is whether several procedures, all simply described, are better than a single procedure which does the job polymorphically but is more complex. The latter has the distinct benefit that there is no need to invent new names for each object class used. It suffers because the procedure needs to be recoded for any addition of classes and because it cannot detect a type error until run-time. However, there may sometimes be procedures which are required to operate over a fixed set of classes, which will clearly never change. So, the next step is:

11 Identify such procedures and code them using **is**.

9.3.3 The Run-time Compiler and Parametric Polymorphism.

As well as the *ad hoc* mechanism described above, there is also a need for a more general parametric polymorphism in which a single procedure is written which can operate on a potentially infinite set of classes. In PS-algol, this is provided by using the run-time compiler to produce tailored procedures for each class encountered. This technique has been illustrated in the PS-algol browser (section 4.2), in GRAPE (section 6.3), in PSRML (section 7.2) and MINOO (section 7.4), so only a brief reiteration here is in order.

The technique involves discovering the type structure of the incoming object and inserting strings derived from this (such as the structure definition and field names and types) into an algorithm template representing the invariant parts of the required procedure. This procedure is then compiled and run against the input object. The resulting procedures may be memo-ised by the class description to prevent recompilation. The string manipulation involved may become complex (see section 9.6.3).

The technique as described so far has been used only to operate on any PS-algol class. For instance, the browser traverses any structure. There are no extra facilities required to produce procedures which operate on a bounded set of classes - for instance, all classes with a **string** field named *name*. Thus the kind of polymorphism provided by a language like Machievelli [Ohori *et al.*, 1989] is implementable in PS-algol.

Therefore, there are two more steps in the program development:

12 Identify any procedures that should be programmed using the run-time compiler and code them as shown in the examples.

13 Any remaining procedures should then be coded as specific procedures either stored directly in a procedure library or within other procedures.

9.3.4 The pntr Type and Inclusion Polymorphism.

The final form of polymorphism which can be provided in PS-algol is the inclusion polymorphism prevalent in Object-Oriented systems. This form of polymorphism is in evidence in the example of a class *PERSON* with an attribute *address* and a sub-type *STUDENT*. Associating an operation *printAddress* with the class *PERSON* makes it available to any object of class *STUDENT* as well. Examples have been shown (PSRML, section 7.2 and MINOO, section 7.4) to illustrate how this is implemented in PS-algol by means of inheritance links. For instance, one implementation of a type system would have nodes whose structure looks like:

structure type(**string** *name*; ***pntr** *operations*; .....; ***pntr** *supertypes*)

The code to find an operation for an object starts with a search of the *operations* field of that object's type. If the operation is not found there, the *supertypes* field is dereferenced and the supertype nodes are searched for the object.

Notice that this mechanism provides the possibility of **overriding** as well. If there is a *printDetails* operation defined on class *PERSON* and another *printDetails* is defined on *STUDENT*, which prints more details, then starting the search at the *STUDENT* class will cause the *STUDENT* version to operate. Indeed, the mechanism is flexible in that the code which searches for operations can have a variety of semantics imposed on it.

The following step summarises this section:

14 If an inclusion polymorphism is required among classes, structure the object type nodes as in the above example and write an operation dispatcher to traverse the type graph according to the required semantics.

9.4 Manipulating the Persistent Store.

This section is merely a personal recommendation on the way to set up PS-algol databases to maximum advantage. It is envisaged, first of all, that in a multi-application environment, there will be a need for central, read-only, databases. The language comes with programs for setting up a system database and a database of fonts. A database of utilities should also be set up along the lines described in Chapter 8.

For any specific application, an additional database of the kind illustrated for the BRDB should be set up. As for the BRDB, this will include a library of code specific to the application, meta-data and data. Indeed, Figure 5.9 is a reasonable template for any such database. In general, two top-level programs will be required, one which initiates the database and one to start the application running. All other code should be written in the form of programs which insert code into the database.

The internal organisation of the database will be determined by the data structures required for processing. Given the modelling power of the structures, these may in turn reflect the nature of the objects in the application domain in a way which mirrors their real-world structure. In fact, each data structure will be chosen either to support processing where significant computation is expected, or to model the real world where communication predominates. As one example of the internal structure of a database, a bulk object may be implemented in numerous ways:

as a **vector** - this means that all of the elements have the same type - vectors are best used if indexing is of paramount importance and insertion and deletion of objects is relatively rare;

as a **structure** - used for small sets of differently typed elements;

as a **list** - useful for small sets with many insertions and deletions;

as a **table** - useful for large sets with many insertions and deletions, also if the *scan* operation is required, but objects whose type is other than **pntr** need to be packaged.

In general, the choice between these objects is like to depend on the code of the program and will follow from a choice in the program design. This is one of the most desirable consequences of orthogonal persistence.

Thus the next step is:

15 Write programs which set up the application database, put the code modules in the database and run the application.

9.5 Organising the User Interface.

One of the key benefits of PS-algol is the provision of the graphical types, using which the user interface can be designed in the same way as the rest of the program. The separation of user interface design from the rest of the program, although apparently desirable, has been criticised, e.g. [Coutasz, 1987]. Conversely, the work here shows that a close integration of the interface and program brings benefits, which will now be re-iterated.

Firstly, the separation of user interface from the rest of the program is usually taken to the point where they are implemented in separate environments with a fixed and difficult to change interface between the two. Window managers come with tool sets which reduce the degree of choice available to the interface designer. The PS-algol approach is to provide tool sets, but leave the programmer with the ability to create different versions or completely different tools as required. Thus, PS-algol comes with the *menu* function, but Chapter 3 showed how different menu systems could be built to replace this.

Secondly, it is possible to keep data and their graphical representation together. The usual technique is to provide a library of symbols, possibly inextensible, and to have a given object connect to a symbol by a reference. In PS-algol, a structure for an object can have an extra field containing an icon for that object, thus simplifying the code required to manipulate the icons.

Thirdly, the pay-off between the cost and speed of storing images can be finely controlled. For any image on the screen, the program can either store the image itself (fast, but potentially requiring a lot of space) or store a program to create the image together with the relevant parameters. Which is chosen depends on the requirements of the application, but PS-algol permits the choice to be made in terms of these requirements and not in terms of the limitations of the development environment.

Therefore, the organisation of the interface can be determined entirely by the requirements of the application. The various phases of the interface, screen layouts and interaction styles can be specified and then code written to produce the required effects. In general, it is possible to produce a module for any particular user interface tool. These can then be structured into the user interface part of the program. The consistent style of menus and dialogue boxes used throughout this work is one such way of organising the interface, although other styles are possible. One alternative is the event driven architecture developed in [Cutts and Kirby, 1987], in which an Object-Oriented style of programming is provided as a Notifier, which registers actions, monitors events and initiates appropriate actions when the events happen.

The final step is:

16 Determine an interface style. Provide appropriate primitives and then build the interface.

It should be re-emphasised at this point that the design steps spelled out linearly may be interwoven and taken in different orders as appropriate to the application or part of the application under design.

9.6 Deficiencies of PS-algol.

This section discusses a number of potential drawbacks of PS-algol found in producing the software examples given in the body of this thesis. The type system imposes different methods of handling simple and complex data objects. There is no way to restrict **pntr** references to point to objects from a specific class. The run-time compilation system is cumbersome to use and is poorly interfaced with the calling program. The language lacks facilities for concurrent access. The commit function is very low-level compared with sophisticated transaction systems. Distribution is not dealt with. The reliance on memory garbage collection causes interruptions to program execution.

9.6.1 The Divided Type System.

The type system has two principal weaknesses. Firstly, the types of base and complex values are not well integrated. Polymorphic procedures can be provided, as has been shown, which range over any kind of structured value. It would be desirable to be able to range over all types, including **string**, **int**, ***int**, **proc(string -> int)**, etc. For example, it is not possible to write a procedure which permutes the elements of an arbitrary vector type. To operate polymorphically in a world which includes simple and complex values, it has been found necessary to package and unpackage the simple values.

Dealing with these exceptional objects constitutes a significant part of the code. The solution to this problem as proposed in languages such as Amber [Cardelli, 1984] and Napier88 [Morrison *et al.*, 1988b] is to provide a universal union type, called variously **dynamic** or **any**, which is truly universal. That is, every type in the type system of Napier88 is a sub-type of **any** and so polymorphic procedures can be written which take arguments which may truly be of any type. Checking these types is then done by projecting the type of the object into some sub-type or by using parametric polymorphism.

9.6.2 Unspecified pntr References.

In creating structures with **pntr** fields it is often possible to specify the type of these fields and expect the compiler or run-time system to enforce the type. For instance, in the following structure for nodes of lists of string:

stringListNode(**string** *value*; **pntr** *next*)

it would be useful to force the *next* field to point to another *stringListNode* (or **nil** of course), but this cannot be specified in PS-algol. These two problems point to the need for a more powerful and uniform type system, for instance that of Napier88.

9.6.3 Run-time Compiled Procedures Break the Uniformity.

Having access to the compiler at run-time is one of the back-bones of the implementation methodology. The merging of the algorithm and the type description into a string which is subsequently compiled and then applied to the object is a very powerful technique which effectively resolves the tension between the security of strong-typing and the expressive power of polymorphism. It also offers the potential of very high efficiency by optimisations, performed during the code production, which depend both on type and values. However, the way in which this is achieved in PS-algol diverges from the uniform programming style which is a primary quality of the language. There are three ways in which this divergence makes itself evident. Two of them are concerned with the way the string is constructed, while the third concerned with the way data are shared between procedures, an issue which will be dealt with in the next section.

The first point seems fairly trivial, but illustrates the divergence in the way string literals are put into a procedure. Suppose the line:

```
print "Hello mum'n"
```

is required in a compiled procedure. Because the line needs to be put into a string, it must be rewritten:

```
"print '"Hello mum "n"'"
```

because both the quote character and the escape character (') need themselves to be escaped when inside a string literal. Written thus, the form is ugly and difficult to parse by eye. This can be circumvented, for instance by writing:

```
"print #QHello mum#N#Q"
```

where #Q and #N are placeholders for quote and newline characters, which can be replaced automatically prior to compilation time. However, although this can be said to clarify the program somewhat, it still means that the line is different from the first form and implies that some parts of the program must be written using different syntax than other parts.

The second point concerns the way the string is built. Initially, a style was adopted in which a string variable was successively extended by string concatenation. This required lines of the form:

```
let source := source ++ "print '"Hello mum "n"' 'n"
```

which is clearly unsatisfactory. This style was subsequently superceded by the style introduced in Figure 4.13, in which the whole algorithm is written as a single string containing placeholders for any type dependent text. These placeholders are subsequently replaced by the actual text to appear there, using a standard set of replacement procedures. This technique produces much clearer code, although the implementation of repetitive pieces of text, such as initialising a vector or structure, can produce slightly unclear code. However, writing a procedure using this style is not the same as writing a procedure directly and so the much valued uniformity of PS-algol as a programming language has been violated.

9.6.4 Sharing Data With Run-time Compiled Procedures.

Worse still, a procedure written in this way does not exist in the same environment in which it is specified. To explain this point further, if a procedure *P* is written in the normal manner, it has available to it all of the variables which are in scope at the point of its declaration. A compiled procedure, *CP* say, has no such variables available to it, except, of course, the variables in the standard environment.

In order to get any interaction or data sharing between *CP* and the environment in which it is declared, one of two unsatisfactory methods must be used. Either shared data must be put into the Persistent Store and retrieved from there or the environment must be passed into the procedure as parameters. In MINOO for instance, every system- or user-created operation receives the type table and the symbol table as parameters, which means that it can traverse the environment with freedom as these are the two roots of all information. Neither of these techniques seems to be the natural way to model the programmer's intention.

What is required to overcome both of these problems is some mechanism for saying: compile algorithm *A* in the context of environment *E*, where *A* is parameterised. Quite what kind of parameterisation is useful here, whether by type or by data values, seems a fruitful area for language design research. Clearly the language Napier88 with its richer type system and environments goes a long way to solving all of the problems described here.

9.6.5 Constancy.

One aspect of the PS-algol type system which has not been exploited to the full in this work is that any value can be declared to be **constant**. Subsequently its value cannot be changed and so this mechanism can be used to provide some protection for data from corruption. This would be useful for implementing data models, for instance, in which some fields of an entity type are declared to be constant.

However, in this work little conscious use has been made of the facility because of the limited way in which it has been provided. In PS-algol, every value is either constant or it is modifiable. Therefore, if a conscious decision is made to declare a part of the data structure to be constant, then this decision may not be subsequently changed. Furthermore, in PS-algol, this is a property of the type and not of the value, thus prohibiting the passing of parameters of the wrong constancy to procedures. Therefore, the decision was usually to leave everything variable. There is an argument to be made that what is required here is a greater range of values for the constancy of an object, from "this can never be changed" to "this can always be changed", including some intermediate values such as "this can only be changed with difficulty". Research into this area is recommended.

9.6.6 Concurrency control.

As has been said, concurrency control in PS-algol is at the "database" level. Any database can be opened by one writer or multiple readers, but not both. In a multi-user system, this leads to problems. Consider, for instance, a system library of utility

procedures. Applications programs open this in read-mode to get at the utilities they require. Two problems occur, however.

Firstly, the utilities must not have any persistent state variables, as these would reside in the utilities database. Therefore, if they were changed, the utilities database would have to be opened in write-mode and so be inaccessible to other applications. This problem can be avoided by locating any persistent state in the application database, although the programming of this is often inelegant. More serious is the problem of library update. If a new or revised utility is put in, the utilities database must be opened in write-mode, and all of the dependent applications must be suspended. Better concurrency provision is required - for instance, locking of finer granularity.

There are two possible development paths to achieve this. Either the primitives of PS-algol could be changed or more complex control structures could be built on top of PS-algol. In fact both paths need to be followed. In retrospect, the "database" is an unnecessarily heavyweight object, which is used simultaneously to provide a persistent root and a unit of concurrency. Napier88 eliminates the database and reduces the store to a single persistent root. What is required instead are primitives to support concurrency. These may be lightweight processes [Wai, 1988], atomic locks and resources [Krablin, 1985] or mutexes, as proposed for Napier.

Further research is required to determine the nature of the low-level primitives, but they should be kept extremely simple and very few in number. Then, higher level facilities can be built on top of them using the techniques illustrated elsewhere. For instance, [Krablin, 1985] describes an extension to PS-algol called CPS-algol, which extends the language with primitives for atomic locks and resources. With these few additions, Krablin is able to build transactions which are atomic, serializable and recoverable, by representing them as Abstract Data Types and using the first-class procedures to pass "processes" around. This kind of approach, providing few low-level primitives and then building more sophisticated facilities on top of them, keeps the language simple and yet provides the power needed.

9.6.7 Commit and Database Update.

A similar granularity problem concerns the **commit** command. When commit is executed, **every** object which has been modified and is reachable from a persistent root is written to backing store and all the old values are lost. Two problems arise from this: you cannot commit some of the changes; and you cannot undo a commit. The former is of relevance in CAD when, for instance, a ship, in which each of the sections is in a different part of the same database. Some modifications are made to the hull, but not committed because they are tentative. Then some changes are made to the superstructure which prove useful and they are therefore committed. The hull changes are now made as well, even though this was not intended. Conversely, if the changes to the superstructure were to be aborted, the changes to the hull would be lost as well.

These problems are part of an overall lack of fine granularity, transaction control and rollback capabilities at the primitive level. In fact, however, as the development of the Bibliographic Database showed (Section 5.3.4), such capabilities can be built on top of the **commit** primitive. Similarly, any mechanism for object version

control and database history maintenance can be built as a re-usable component for any application. There remains, however, the suspicion that a better primitive would be a version of commit which took a pointer into the persistent store as a parameter and committed all the changes reachable from that pointer. But if **retract** is similarly parameterised, common sub-structures lead to ill-defined or complex semantics.

9.6.8 Distribution.

PS-algol does not attack the problem of distributed data and computation, which must become a critical issue over the next few years. An extension called Distributed PS-algol has been produced by [Wai, 1988], which introduces lightweight processes coupled by remote procedure calls. It is intended to extend this work to permit a program to refer directly to remote data. This looks a promising step, but there will be many problems encountered when trying to run programs against a widespread distributed store, which current store technology techniques do not begin to solve. Some of these problems touch on questions such as where remotely accessed data should be held (i.e. should it be replicated locally or moved?), what to do if nodes disappear from a network and how to find a given node. Other questions concern how to implement remote access efficiently. These questions are outside the range of this thesis.

9.6.9 Garbage Collection.

Extensive use has been made of the availability of procedures as first-class objects in PS-algol. This feature means that a running program has sometimes to be suspended so that local store can be statically garbage collected. To see the reason for this, examine the following code fragment:

```
begin
        let  x := 0
        let P := proc( )
          begin
                .....
                x := x  + 1
                .....
          end
        P
end
```

This is a block which exports a procedure, *P*. Inside the block, a local variable, *x*, is declared. In a language like Pascal, *x* could be thrown away at the end of the block. In PS-algol, it cannot be thrown away as it is used in the body of *P*, which itself may be referenced outside the block. It is hard, if not impossible, for the compiler to decide statically which objects it can throw away. Therefore space reclamation is performed at run-time. All objects which have been used are put onto a heap as they are declared. When there is no more space for the heap to grow, the program suspends itself and then all objects which are reachable from current objects are marked (the *x* would be marked because it is reachable from *P* if *P* itself is still referenced). Any unmarked objects are then discarded and the space they occupied is reclaimed.

In the current implementation, this garbage collection process happens during the run of the program and takes a noticeable time. This will be unacceptable for some programming tasks. Other techniques for managing garbage should be investigated, both improvements in compiler technology and in garbage collector technology. A more sophisticated compiler could discover which values will never be re-used and eliminate them before garbage collection. One new version of the store manager has been brought out which, at each garbage collect, only keeps written-to objects on the heap. This cuts down the number of such garbage collections, as for instance in the (common) case, in which the whole of a table has been scanned, perhaps to provide summary information. All of the objects in the table remain on the heap, filling it up, even though they are not likely to be used again. Other techniques which might be tried include layered garbage collectors, keeping reference counts (problems arise here with circular lists) or ageing (long-term unused objects are removed from the heap). Such improvements are required if PS-algol is to be used for production-quality software.

9.6.10 Summary of Deficiencies.

A number of deficiencies have been listed, which can roughly be divided into three groups: language design deficiencies which have now been superceded; issues which PS-algol was never designed to face; and implementation inefficiencies due to the novelty of the language. Napier88 can be seen as a significant improvement over PS-algol in providing polymorphic types, environments, variant types and abstract data types, as well as having a proposal for concurrency primitives, so many of the above criticisms are overcome by Napier88. Concurrency and distribution are hard problems for a database programming language to solve. The simplicity and power of PS-algol has provided a useful basis on which extensions can be built to test out ideas for providing such solutions, even though PS-algol itself does not do so. Finally, a fast industrial quality version of PS-algol with an optimising compiler, code generation and state-of-the-art garbage collector requires industrial quality resources.

9.7 Conclusions.

A methodology has been presented for creating a program in PS-algol. The data structures of the program have been described in a way reminiscent of semantic data modelling techniques. Then the dynamic components were added in a variety of ways depending upon features of the procedures. The freedom with which these procedures may be manipulated in PS-algol contrasts strongly with other programming systems in which procedures have arbitrary restrictions placed upon them. Finally the methods of designing a database and the user interface were described. Using this methodology, it is thought that software for large and complex tasks can be coded quickly.

A section on the deficiencies of the language was then presented. These deficiencies, in some cases serious, do not imply that the novel features of PS-algol must be abandoned, but rather that the language needs to be refined and re-engineered to circumvent these problems. The language Napier88, which is outside of the scope of this thesis, would seem to offer solutions to at least some of the problems. Others will take longer to correct.

Chapter 10. Conclusions.

In this chapter, the findings of the research are summarised, some general conclusions are drawn concerning the effectiveness of the persistent programming paradigm and recommendations for future work are made.

10.1 Summary.

In the introduction, the requirement for improving facilities for software production was introduced. A number of approaches were introduced including making software specification easier and reducing the amount of coding there is to do. The Persistent Programming paradigm was introduced as a potential foundation for improving the economy of software production.

The paradigm proposed languages which are simplified by the removal of discontinuities between the ways different parts of the programming task are carried out. In particular the differences in treatment of long-term and short-term data are removed, which leads to a programming environment in which there is only one data model encompassing both the computational and database aspects of the data. Another discontinuity which is eliminated in Persistent Programming is that between program and data. The procedures which represent modules of program code are treated in the same way as any other data value in the language. One further discontinuity which is removed is the differences in treatment of numerical, textual and graphical data.

A persistent environment makes uniform the ways in which all kinds of data values are manipulated. A persistent programming language is designed to be sufficient to handle more of the programming task than is normally the case, including the human computer interface and the storage of long-term data. The coherence and simplicity of languages which provide this are held to ease programming, since the programmer has fewer exceptional cases to remember, fewer data models to think about and fewer languages to master. The research reported here was designed to test whether or not these features resulted in the expected benefits.

Firstly, a survey of other approaches to the problem of improving software development was carried out, from the standpoints of language design, database systems and software engineering. In the area of language design, it was found that languages are providing increasingly useful and high-level programming constructs, which permit simpler descriptions of algorithms and the ability to prove the equivalence of a program and its specification. However, languages tend either to concentrate on one part of the problem (for instance, functional languages have yet to solve the problems associated with long-lived data) or to become extremely complex (for instance, Ada).

Database systems have grown in a number of ways. They have developed efficient storage and retrieval mechanisms, on the one hand, and higher-level data modelling tools, on the other. The current research activity into Object-Oriented Database Systems tries to match these two with a powerful computational model. However, such approaches seem to suffer because they propose the dominance of data over program in such a way that the representation of program "objects" no longer has

the generality and power traditionally associated with programs. OODBS often re-introduce multiple models and multiple languages.

Software environments are increasingly viewed as the solution to the crisis in software production. Automatic tools to assist programmers in maintaining the coherence of a large programming project are built into a support environment for the manipulation of program modules. This may be thought of as a database management task, in which the program modules are the data. Particular software development tools, version managers, configuration managers, etc. are applications within this DBMS. Given a sufficient application development environment, it will be possible to allow different projects to produce the environment which best suits their own needs.

The conclusions from this survey are that none of the approaches have yet reached the stage at which they can be confidently expected to produce the kinds of improvement in software production which are required. However, individually they provide parts of the solution: high-level data models; software development tools; strong type systems; the polymorphic description of code; the ability to prove the correctness of software; etc. Persistent Programming picks up many of these themes for incorporation into a uniform programming system.

Chapter 3 introduced the language PS-algol as the first significant exemplar of a Persistent Programming Language. This language is computationally complete, strongly typed and has first-class procedures, orthogonal persistence, graphical types, a mechanism for describing objects of arbitrary complexity and a compiler which is callable at run-time. It was concluded that the language had sufficient primitives with which to describe the data model (at a fairly high level), the computational model (with programming constructs typical of high-level procedural languages), the human computer interface and such systems functions as compilers and software managers.

Chapter 4 showed how certain tools are developed in PS-algol. These included such user interface components as menus, editors and dialogue boxes, together with system tools such as a general purpose database browser and compiler generation tools. A quick tutorial on the use of the language was given, particularly illustrating how the callable compiler and the **pntr** type are used to generate polymorphic procedures. It was concluded from this chapter that PS-algol was sufficient for such tasks.

Chapter 5 described how a stand-alone database application could be directly programmed in PS-algol. The application maintains a set of bibliographic references and builds bibliographies automatically. The development of this application was described, concentrating on: the way in which a database is managed in PS-algol; how the tools described in Chapter 4 emerged naturally from the development of this application and were stored in a re-usable form; the way in which a facility not present in PS-algol, a transaction manger, was provided on top of the primitives; and the way in which the user interface was designed and built. The development of the application was reasonably quick given that a methodology for such programming was being designed simultaneously.

Chapter 6 described two relational DBMS's. One, constructed by Pedro Hepp, provided a variety of interfaces to the same database, while the other used the callable compiler to tailor efficient storage mechanisms to particular data classes. Again these

systems were developed rapidly and use was made of the high-level facilities available in PS-algol to produce this speed.

Chapter 7 moved to higher-level database descriptive systems and showed how to produce executable data modelling tools quickly. The systems developed included implementations of the functional data model, a requirements modelling language (illustrating a design tool with rapid prototyping in mind), IFO and a minimal Object-Oriented language. These implementations were put into a common framework which could be used to implement any semantic data model. Again, facilities such as first-class procedures, the callable compiler and the **pntr** type were used to turn arduous programming tasks into rapid constructions.

Chapter 8 returned to the ideas of software engineering and showed how having procedures as first-class objects greatly facilitates the ability to create programs which aid software development. First, a utility library maintenance system, developed in the context of the bibliographic database, was described. Then, a more sophisticated system with version control facilities was described. These prototype systems were used as examples of the ways in which general purpose management systems could be developed not only for software modules, but also for CAD environments.

Chapter 9 took the work of the previous chapters and elicited a methodology for software in the form of a series of discrete steps, which could however be tackled in an order determined by the application and not by the system. These started with the description of the data structure of the application, presented a framework within which the code could be sensibly managed, showed several techniques for producing polymorphic code, described the organisation of the database and, finally, the organisation of the code. Such a methodology is only an initial step towards producing a framework for the organisation of an application.

10.2 The Major Findings.

Recall the statements of the theses in section 1.5.

> **Persistent programming, as exemplified by PS-algol, is a sufficient and effective foundation for the development of large, complex and long-lived systems.**
>
> **The paradigm beneficially influences the style of programming carried out.**
>
> **A methodology can be developed which facilitates this style of programming.**

In the research described here, a number of highly complex, large and long-lived systems have been produced. None of them took a long time to produce. Most of them require significantly less code than might be expected. For instance, the IFO and MINOO implementations take little over 1000 lines of code each. The range of applications discussed has included user interface tools, compiler construction tools, a

general-purpose, polymorphic browser, a database application, several data modelling tools and software construction tools. From these experiments, it is reasonable to conclude that PS-algol is a sufficient language for the description of most parts of many applications.

As discussed in section 9.6, the drawbacks to the use of PS-algol are that it is a prototype of the paradigm (inadequacies of the type system), a research prototype (speed) and that it does not successfully tackle certain issues (such as concurrency) yet. None of these seem drawbacks in the long term. Napier88 already shows a significant improvement in language design, while improvements in implementation techniques, for instance the garbage collector, are on the way for both languages. Providing the other facilities such as concurrency is a more long term project, principally because the semantics of such facilities have yet to be determined. What is clear is that successor languages in the paradigm will carry with them the powerful features of PS-algol that have been used here. It is safe to predict, therefore, that programming in such languages will make use of the techniques described here, as well as other techniques developed in response to the availability of additional features.

The style of programming shown here is subtly different from programming in other block-structured procedural languages. One principal feature is that a great variety of styles can be used. As procedures are first-class values in the language, it is possible to adopt a purely functional style of programming for those parts of the application which are computationally intensive. [Cutts and Kirby, 1987] and [Philbrow *et al.*, 1989] have shown how a purely Object-Oriented style can be used in PS-algol. PS-algol provides sufficient primitives for most programming jobs without restrictions on how they are to be used.

The other point to be made about programming style in PS-algol is that it does not directly provide many high-level functions, rather it enables users to program such facilities themselves. The transaction system (section 5.3.4) developed for the bibliographic database is a case in point. This was developed as the need arose since there were sufficient primitives to produce it. When such modules are produced they may be stored as re-usable modules for later applications to exploit. Conversely, where high-level functions are provided, they need not be taken as cast in stone. The PS-algol menu function, for instance, is a very useful facility, immediately available in the system. If a different form of menu is required, however, this can be produced as well. Users are not restricted to particular implementations of tools.

The major feature of Persistent Programming's style is that it is flexible and open-ended. The user is not tied to a particular way of writing programs, nor to particular toolsets. It is also unusual to write long sequences of code. Normally, a few procedures at a time are defined and stored in the persistent store for re-use.

Such freedom in a novel paradigm could be intimidating without a methodology for program construction and, in the Chapter 9, a number of steps in the production of PS-algol programs were outlined. This must be considered a first cut at producing a more sophisticated version. However, concentrating as it does on features of PS-algol which would be expected to be retained in any future persistent languages, it incorporates many of the expected features. A persistent language will give the

ability to describe the data structure of an application at a high-level. It will also include first-class values for program objects and so the program itself will be amenable to the kinds of analysis presented in section 9.2. Polymorphic programming may be performed differently, but the mixture of static and dynamic type checking and deferred binding can be expected to be present in some form since the need for these has been established. Similarly, the details of how the persistent store and the user interface will be set up may vary. However, the basic features of these will remain: a common data model between the program and the store; and facilities for user interface design which are incorporated into the language.

Therefore, the existence of a methodology for Persistent Programming, albeit embryonic, has been established. It remains to flesh out the details and improve it via refinement in the light of experience.

10.3 Future Work.

The discussion of future work is again divided into the areas of programming language design, database work and software engineering. The discussion will centre around the twin topics of the development and exploitation of the Persistent Programming Paradigm.

The experience with PS-algol, and in particular problems with the type system, has led to the development of the Napier series of languages [Atkinson and Morrison, 1987, Morrison *et al.*, 1988b]. These are introducing a number of significant improvements, including a much richer type system, building on the pioneering work of Luca Cardelli [Cardelli and Wegner, 1985; Cardelli, 1988; Cardelli, 1989], in which the behaviour of higher-level types is analysed. Some of the features of Napier, which are additional to those of PS-algol, include:

- a type **any** which is the union of all Napier types and which may be used for writing code for which the type check is to be deferred until run-time;
- a type constructor for abstract data types;
- the ability to specify types which have type parameters - i.e. generic types;
- a type **env** whose instances are environments - containers for sets of values and for which there are operations to add more values, remove values and to bring the values into scope for the following block of code;
- variant types;
- multi-entry processes.

These features would seem to make Napier a much more expressive language than PS-algol and should make possible yet more programming tasks. The type **any** has a unifying effect on the type system, overcoming the problems raised in section 9.6.1. The specification of parametric polymorphism (using generic types) and of abstract data types is made explicit. Environments remove the need to introduce values singly as in PS-algol, while also providing a useful structuring mechanism for

those values. Processes, as distinct from procedures, produce a more direct representation of active objects.

Experimentation with the language is now possible, as the first version, Napier88, is available. It will be interesting to see if these powerful features have the expected effects on the production of code. Such code should be shorter and more understandable if the new primitives have been correctly designed. It is possible, on the other hand, that intellectual difficulty with some of these constructs may prove a barrier to productivity.

Even richer type systems are being developed, for instance incorporating **kinds** - types whose instances are types [Cardelli, 1988] - as are languages such as Machievelli [Ohori *et al.*, 1989] which make a great deal of use of type inference. Type inference would seem to shorten programs, since some of the relationships between types are deduced by the compiler where they would otherwise need to be explicitly stated by the programmer. Some limit to any inference (for instance, being within a given environment) would seem to be necessary, since otherwise all kinds of unacceptable inferences would be made between types in widely differing domains. Work on type classes to bring more structure to *ad hoc* polymorphism is another potentially fruitful area [Wadler and Blott, 1989].

Developments in other paradigms can be expected to have some impact on Persistent Programming. For instance, it is conceivable that the functional programming paradigm will find some way of dealing with all of the problems of long-lived data without recourse to state - see for instance [Argo *et al.*, 1987] or [Trinder and Wadler, 1989]. Alternatively, the Object-Oriented paradigm may evolve to include mechanisms for describing active objects and thus be suitable for the whole application development task.

In short, languages for data intensive programming are still developing and will do so until they achieve the ability to produce a unified description of the whole of complex programs involving long-lived data, which are shared and distributed.

Database systems are also developing in a number of directions to optimise performance, improve high-level descriptive tools and data manipulation languages. The research that has gone into optimising the performance of classical database systems can now be re-used to improve the performance of Object-Oriented and Persistent systems. On the other, as this work shows, Persistent Languages provide a suitable tool in which to prototype data modelling tools. The only way of verifying the acceptability of a modelling tool is to implement and evaluate it. The implementation work reported here also points the way towards database systems in which the choice of modelling tool is left to the user. Indeed, work is in hand to produce a meta-modelling program, in which the user specifies which modelling constructs are required and what the user interface to those constructs is. A modelling tool is then constructed for use.

Data manipulation and query languages are being standardised, while database programming languages are providing a separate route, which may well replace such languages as SQL. If a full programming language is sufficiently simple and yet can act as a DML or query language, why construct these separately? One of the future aims of

the Persistent Programming is the replacement of these separate and limited languages.

Another aspect is that of concurrent access to databases and transactions. In conventional databases with short-term transactions, simple protocols for locking data, such as two-phase locking, were sufficient to prevent most problems. In design databases with long-term transactions, more sophisticated, fine-grained locking mechanisms with check-in, check-out protocols are required [Katz and Chang, 1987; Fernandez and Zdonik, 1989]. Persistent Programming can assist with this research as prototype locking systems can be built and tested, using Wai's distributed PS-algol [Wai, 1988] as a foundation. Such work is already in hand.

One other aspect of the performance of databases is the requirement that they store an increasingly rich set of basic types. PS-algol provides a step towards the provision of multimedia types - the graphical constructs. Databases for sounds, music, maps and other kinds of document will soon become commonplace. It is thus the job of any long-term database system designer to ensure that future systems will cater for these types.

More traditional database concerns must all be tackled by the designers of persistent systems. Concurrency and distribution have been mentioned previously, but there is also a need to provide reliable systems with a high degree of security.

Thus Persistent Programming is attempting to replace traditional database systems in the future, but it is already providing a testbed for the examination of problems with current database technologies.

There are many projects currently working to develop Software Development Environments. In a short time, no application development of any complexity will be cost effective without the use of such systems. One of the clearest lacks of a "bare" PS-algol system is that there is no framework within which software is conventionally stored. In this work, only the barest outline for such a system has been attempted, but it has shown that the production of SDE's is facilitated by a language such as PS-algol. Experiments are in hand to design and implement a proper system for PS-algol, which will act as a prototype for future languages. This will capitalise on the work produced in other contexts.

Furthermore, there is a need to design and implement large sharable libraries of software. This will require not only the specification of suitable components but also some considerable understanding of how programmers actually co-operate and exchange code.

All of these techniques will then need to be brought to bear on the production of large-scale applications of high quality. This will involve teams of programmers working in the iterative manner outlined in Chapter 9 on applications which are expected to store data and to evolve over a long period of time. Two examples of such systems are an OODBMS and a complete programming environment. The production of a good quality OODBMS using persistent technology would both act as a demonstrator and show considerable advantages compared with the *ad hoc* methods used elsewhere. The creation of a total programming environment (for instance, a

replacement for the MacIntosh environment) using a persistent kernel should show large savings in implementation cost. It would be of interest to see if such an implementation resulted in any changes of system use.

In summary, Persistent Programming Systems may be expected to develop in line with current developments in programming language, database and software engineering research. These findings will be incorporated into Persistent Systems, keeping them as simple as possible by extending the functionality of the systems in a uniform manner.

Bibliography.

Abdullah, 1989

H. Abdullah, Ph.D. Thesis, University of Glasgow, to appear.

Abiteboul and Hull, 1988

S. Abiteboul and R. Hull, "IFO: A Formal Semantic Data Model", *ACM TODS*, 12, 4, 525-565, December 1987.

Abrial, 1974

J.R. Abrial, "Data Semantics", *Data Base Management*, North-Holland, Amsterdam, 1-59, 1974.

ACARD, 1986

"Software - a Vital Key to UK Competitiveness", Advisory Committee for Applied Research and Development, HMSO 1986.

Albano *et al.*, 1985

A. Albano, L. Cardelli and R. Orsini, "Galileo: A Strongly Typed, Interactive, Conceptual Language", *ACM TODS*, 10, 2, 230-260, June 1985.

Apple 1984

Apple Computers Inc., "The MacIntosh Manual", 1984.

Argo *et al.*, 1987

G. Argo, R.J.M. Hughes, J. Launchbury, P. Trinder and J. Fairbairn, "Implementing Functional Databases", *Proceedings of the Workshop on Database Programming Languages, Roscoff,*, (F. Bancilhon and O.P. Buneman eds.), ALTAIR - CRAI, September 1987.

Atkinson 1978

M.P. Atkinson, "Programming Languages and Databases", *Proceedings of the 4th International Conference on Very Large Data Bases*, Berlin, (ed. S.P. Yao), IEEE, 408-419, September, 1978.

Atkinson, 1988

M.P.Atkinson, "Persistent Programming and Object-Oriented Databases", Joint International Seminar on Teaching Computer Science, Computer Laboratory, University of Newcastle-upon-Tyne, September 1988.

Atkinson and Buneman, 1987

M.P. Atkinson and O.P. Buneman, "Types and Persistence in Database Programming Languages", *ACM Computing Surveys*, 19, 2, 105-190, June, 1987.

Atkinson and Morrison, 1985a

M.P. Atkinson and R. Morrison, "Procedures as Persistent Data Objects", *ACM TOPLAS*, 7, 4, 539-559, October 1985.

Atkinson and Morrison, 1985b

M.P. Atkinson and R. Morrison, "Integrated Persistent Programming Systems", *Proceedings of the 19th Annual Hawaii International Conference on System Sciences*, (ed. B. D. Shriver), vol IIA, Software, 842-854, January 1986.

Atkinson and Morrison, 1987

M.P. Atkinson and R. Morrison, "Types, Bindings and Parameters in a Persistent Environment", *Data Types and Persistence* (M.P. Atkinson, R. Morrison and O.P. Buneman eds.), Springer-Verlag, 1987.

Atkinson *et al.*, 1981

M.P. Atkinson, K.J. Chisholm and W.P. Cockshott, "PS-algol: An Algol with a Persistent Heap", *ACM SIGPLAN Notices*, 17, 7, 24-31, July 1981.

Atkinson *et al.*, 1982

M.P. Atkinson, K.J. Chisholm and W.P. Cockshott, "Nepal - the New Edinburgh Persistent Algorithmic Language", in *Database, Pergammon Infotech State of the Art Report*, Series 9, No.8, 299-318, January 1982.

Atkinson *et al.*, 1983a

M.P. Atkinson, K.J. Chisholm and W.P. Cockshott, "Algorithms for a Persistent Heap", *Software Practice and Experience*, 13, 3, 259-272, March 1983.

Atkinson *et al.*, 1983b

M.P. Atkinson, K.J. Chisholm and W.P. Cockshott, "CMS - A chunk management system", *Software Practice and Experience*, 13, 3, 273-285, March 1983.

Atkinson *et al.*, 1983c

M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott and R. Morrison, "An approach to persistent programming", *The Computer Journal*, 26, 4, 360-365, 1983.

Atkinson *et al.*, 1983d

M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott and R. Morrison, "PS-algol a language for persistent programming", 10th Australian Computer Conference, Melbourne, 70-79, September 1983.

Atkinson *et al.*, 1988
M.P. Atkinson, R. Morrison and O.P. Buneman, "Binding and Type Checking in Database Programming Languages", *The Computer Journal*, 31, 2, 99-109, 1988.
Bachman, 1969
C.W. Bachman, "Data Structure Diagrams", *Data Base*, 1, 4-10, Summer 1969.
Bancilhon, 1988
F. Bancilhon, "Object-Oriented Database Systems", *Proceedings of the ACM SIGACT-SIGART Conference on the Principles of Dtabase Systems*, Austin, Texas, May 1988.
Bancilhon *et al.*, 1987
F. Bancilhon, T. Briggs, S. Khoshafian and P. Valduriez, "FAD: A Powerful and Simple Database Language", in *Proceedings of the 13th International Conference on Very Large Databases*, Brighton, England, 97-106, September 1987.
Bannerjee *et al.*, 1987
J. Banerjee, H-T. Chou, J.F. Garza, W. Kim, D. Woelk, N. Ballou and H-J. Kim, "Data Model Issues for Object-Oriented Applications", *ACM TOOIS*, 5, 1, 3-26, January 1987.
Bennet and Rowles, 1986
Bennett, S. and Rowles, J. - "Teeny: an Executable Language Based on RML", me-too Internal Report SETC/IN/215, STC Technology Ltd., Newcastle-under-Lyme, 1986.
Bird and Wadler, 1988
R. Bird and P. Wadler, "Introduction to Functional Programming", *Prentice-Hall International Series in Computer Science*, 1988.
Bjørner and Jones, 1982
Bjørner, D. and Jones, C.B., "Formal Specification and Software Development", Prentice/Hall International, 1982.
Blott and Campin, 1987
S. M. Blott and J. Campin, "Lgen, Pgen and Sgen: Language Development Tools", *PPRR49*, Universities of Glasgow and St Andrews, 1987.
Borgida *et al.*, 1989
A. Borgida, J. Mylopolous, J.W. Schmidt and I. Wetzel, "Support for Data-Intensive Applications: Conceptual Design and Software Development", Preliminary Report of the DAIDA Project (ESPRIT Contract #982), 1989.
Bott 1989
F. Bott (ed.), "ECLIPSE: An Integrated Project Support Environment", *IEE Computing Series 14*, 1989.
Brodie *et al.*, 1984
M.L. Brodie, J. Mylopolous and J.W. Schmidt (eds), "On Conceptual Modelling", Springer-Verlag, New York, 1984.
Brodie and Mylopolous, 1986
M.L. Brodie and J. Mylopolous(eds), "On Knowledge Based Systems", Springer-Verlag, New York, 1986.
Brown, 1989
A.L. Brown, "Persistent Object Stores", Ph. D. Thesis, University of St. Andrews, 1989.
Brown and Cockshott, 1985
A.L. Brown and W.P.Cockshott, "CPOMS - A Revised Version of The Persistent Object Management System in C", *Persistent Programming Research Report 13*, Universities of Glasgow and St. Andrews, 1985.
Buneman 1988
O.P. Buneman - private communication, 1988.
Buneman *et al.*, 1982
O.P. Buneman, R.E. Frankel and R. Nikhil, - "An Implementation Technique for Database Query Languages", *ACM TODS*, 7, 2, June 1982.
Buneman and Nikhil, 1984
O.P. Buneman and R. Nikhil, " The Functional Data Model and its Uses for Interaction with Databases", in Brodie *et al.*, 1984.
Campbell and Terwilliger, 1986
R.H. Campbell and R.B. Terwilliger, "The SAGA Approach to Automated Project Mangement", *Proceedings of the International Workshop in Advanced Programming Environments, Trondheim, Norway*, (R.Conradi, T. M. Didriksen and D. H. Wanvik eds.), Springer Verlag Lecture Notes in Computer Science 244, 142-155, June, 1986.
Cardelli, 1984
L. Cardelli, "Amber", *AT&T Bell Labs Technical Report*, Murray Hill New Jersey, 1984.

Cardelli, 1988
L. Cardelli, "Types for Data Oriented Languages" in *Advances in Database Technology EDBT 1988*, Venice, Italy, Lecture Notes in Computer Science 303, Springer-Verlag, J.W. Schmidt, S. Ceri and M. Missikoff (eds), 1-15, March 1988.
Cardelli, 1988b
L. Cardelli, "A preview of the programming language Quest", circulated January 1988.
Cardelli, 1989
L. Cardelli, "Typeful Programming", *Digital Systems Research Center Reports 45*, 1989.
Cardelli and Wegner, 1985
L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction and Polymorphism, *ACM Computing Surveys*, 17, 4, 471-523, December 1985.
Carrick *et al.*, 1987
R. Carrick, A.J. Cole and R. Morrison, "An Introduction to PS-algol Programming", *Persistent Programming Research Report 31*, Universities of Glasgow and St Andrews, 1987.
Chen, 1976
P.P. Chen, "The Entity-Relationship Model - Toward a Unified View of Data", *ACM TODS*, 1, 1, 9-36, 1976.
Cockshott *et al.*, 1984
W.P. Cockshott, M.P. Atkinson, K.J. Chisholm, P.J. Bailey and R. Morrison, "POMS : a persistent object management system", *Software Practice and Exerience*, *14*, 1, 49-71, January 1984.
Cockshott, 1983
W.P. Cockshott, "Orthogonal Persistence", Ph. D. Thesis, University of Edinburgh, February 1983.
CODASYL, 1971
CODASYL, "Data Base Task Group Report", *ACM*, New York City, New York, April 1971.
Codd 1970
E.F. Codd, "A Relational Model of Data for Large Shared Data Banks" *CACM*, 13, 6, 377-387, June 1970.
Codd, 1979
E.F. Codd, "Extending the Relational Model to Capture More Meaning", *ACM TODS*, 4, 4, 397-434, 1979.
Cole and Morrison, 1982
A.J. Cole and R. Morrison, "An introduction to programming with S-algol", *Cambridge University Press*, Cambridge, England, 1982.
Cooper, 1987
R.L. Cooper, "Applications Programming in PS-algol", *Persistent Programming Research Report 25*, Universities of Glasgow and St. Andrews, 1987.
Cooper, 1989a
R.L. Cooper, "Persistent Languages Facilitate the Implementation of Software Version Management", *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, (ed. B. D. Shriver), vol II, Software, 56-66, January 1989.
Cooper, 1989b
R.L. Cooper, "The Implementation of an Object-Oriented Language in PS-algol", *Proceedings of the Workshop on Persistent Object Systems, their Design, Implementation and Use*, (J.Rosenberg ed.), Newcastle, New South Wales, January 1989.
Cooper and Atkinson, 1987
R.L. Cooper and M.P. Atkinson, "The Advantages of a Unified Treatment of Data", Software Tools 87: Improving Tools, Advance Computing Series, 8, 89-96, Online Publications, June 1987.
Cooper and Atkinson, 1988
R.L. Cooper and M.P. Atkinson, "A Requirements Modelling Tool Built in PS-algol", *Persistent Programming Research Report 54*, Universities of Glasgow and St. Andrews, 1988.
Cooper and Qin, 1989
R.L. Cooper and Z. Qin, "An Implementation of the IFO Data Model in PS-algol", *Persistent Programming Research Report*, Universities of Glasgow and St. Andrews, 1989.
Cooper *et al.*, 1987a
R.L. Cooper, D.K. MacFarlane and S. Ahmed, "User Interface Tools in PS-algol", *Persistent Programming Research Report 56*, Universities of Glasgow and St. Andrews, 1987.
Cooper *et al.*, 1987b
R.L. Cooper, M.P. Atkinson, and S.M. Blott, "Using a Persistent Environment to Maintain a Bibliographic Database", *Persistent Programming Research Report 24*, Universities of Glasgow and St. Andrews, 1987.

Cooper *et al.*, 1987c

R.L. Cooper, M.P. Atkinson, D. Abderrahmane and A. Dearle, "Constructing Database Systems in a Persistent Environment", in *Proceedings of the 13th International Conference on Very Large Databases*, Brighton, England, 117-126, September 1987.

Cooper *et al.*, 1989

R.L. Cooper, J. Campin, D. Chan, D.J. Harper, D.A. Kerr, Z. Qin, F. Wai and R.C. Welland, "A Fast Prototype of the Most Relevant Components of the Object Data Management System", *Deliverable for the COMANDOS Project*, (ESPRIT Contract #834), 1989.

Coutasz, 1987

J. Coutasz, "The Construction of User Interfaces and The Object-oriented Paradigm", *Proceedings ECOOP*, Paris, 121-130, June 1987.

Cox 1986

B.J. Cox, "Object-Oriented Programming: An Evolutionary Approach", *Addison-Wesley*, 1986.

Cutts and Kirby, 1987

Q. Cutts and G. Kirby, "A PS-algol Toolset Utilising Event Monitoring", *Persistent Programming Research Report 47*, Universities of Glasgow and St Andrews, 1987.

Dahl and Nygard, 1966

O. Dahl and K. Nygaard, "Simula, an algol-based simulation language", *CACM*, 9, 9, 671-678, September, 1966.

Davison and Zdonik, 1986

J.W. Davison and S.B. Zdonik, "A Visual Interface for a Database with Version Management", *ACM TOOIS*, 4, 3, 226-256, July 1986.

Dearle, 1988

A. Dearle, "On the Construction of Persistent Programming Environments", Ph. D. Thesis, University of St. Andrews, 1988.

Dearle and Brown, 1988

A. Dearle and A.L. Brown, "Safe Browsing in a Strongly Typed Persistent Environment", *Computer Journal* 31, 3, 1988.

Dearle *et al.*, 1989

A. Dearle, R. Connor, A.L. Brown and R. Morrison, "Napier88 - A Database Programming Language?", *Proceedings of the 2nd International Workshop on Database Programming Languages*, Oregon, 213-230, June 1989.

Demers and Donahue, 1979

A. Demers and J.E. Donahue, "Revised Report on Russell", *Technical Report TR79-389*, Cornell University, 1979.

Dittrich *et al.*, 1986

K.R. Dittrich, W. Gotthard and P.C. Lockeman, "DAMOKLES - A Database System for Software Engineering Environments", *Proceedings of the International Workshop in Advanced Programming Environments, Trondheim, Norway*, (R.Conradi, T. M. Didriksen and D. H. Wanvik eds.), Springer Verlag Lecture Notes in Computer Science 244, 353-371, June, 1986.

Donahue, 1987

J. Donahue, "What's A Database?", *Proceedings of the 2nd International Workshop on Persistent Object Stores*, Appin, August, 1987.

Ecklund *et al.*, 1987

Ecklund D, Ecklund E, Eifrig R and Tonge F, "DVSS: A Distributed Version Storage Server for CAD", in *Proc 13th International Conference on Very Large Databases, Brighton,England*, 443-454, September 1987.

Fernandez and Zdonik, 1989

M.F. Fernandez and S.B. Zdonik, "Transaction Groups: A Model for Controlling Co-operative Transactions", *Proceedings of the Workshop on Persistent Object Systems, their Design, Implementation and Use*, (J.Rosenberg ed.), Newcastle, New South Wales, 128-138, January 1989.

Fishman *et al.*, 1987

D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derrett, C.G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M.A. Neimat, T.A. Ryan and M.C. Shan, "Iris, An Object-Oriented Database Management System", *ACM TOOIS*, 5, 1, 48-69, January 1987.

Gehani and McGettrick, 1986

N. Gehani and A.D. McGettrick (eds.), "Software Specification Techniques", *Adison Wesley International Computer Science Series*, 1986.

Glaser *et al.*, 1984

H. Glaser, C. Hankin and D. Till, "Principles of Functional Programming", *Prentice-Hall*, 1984.

Goguen and Winkler, 1988
J.A. Goguen and T. Winkler, "Introducing OBJ3", *SRI International Report*, SRI-CSL-88-9, 333 Ravenswood Ave., Menlo Park, California CA 94025, August 1988.
Goldberg and Robson, 1983
A. Goldberg and D. Robson, "Smalltalk-80: The Language and Its Implementation", *Addison Wesley*, Reading, Mass., 1983.
Greenspan, 1984
S.J. Greenspan, "Requirements modelling: a knowledge engineering approach to software requirements definition", *Technical Report CSRG-155*, University of Toronto, March 1984.
Greenspan *et al.*, 1986
S.J. Greenspan, A. Borgida and J. Mylopoulos, "A Requirements Modeling Language and its Logic", in Brodie and Mylopolous, 1986.
Habermann and Notkin 1986
A.N. Habermann and D. Notkin, "Gandalf Software Development Environments", *IEEE Trans. on Software Engineering*, 12, 2, 1986.
Hammer and McLeod, 1981
M. Hammer and D. McLeod, "Database Description with SDM: A Semantic Database Model", *ACM TODS*, 6, 3, 351-386, 1981.
Hartson and Smith, 1987
H.R. Hartson and E.C. Smith, "Rapid Prototyping for the Handbook of Human-Computer Interaction", *Virginia Tech. Technical Report TR 87-26*, May, 1987.
Hayes, 1987
I. Hayes (ed.), "Specification Case Studies", *Prentice-Hall International Series in Computer Science*, 1987.
Hepp 1983a
P.E. Hepp, "A DBS Architecture Supporting Coexisting Query Languages and Data Models", Ph. D. Thesis, University of Edinburgh, 1983.
Hepp 1983b
P.E. Hepp, "A DBS Architecture Supporting Coexisting User Interfaces: Description and Examples", *Persistent Programming Research Report 6*, Universities of Glasgow and St. Andrews, 1983.
Hewitt *et al.*, 1973
C. Hewitt, P. Bishop and R. Steiger, "A Universal ACTOR Formalism for Artificial Intelligence", *Proceedings of the International Joint Conference on Artificial Intelligence*, Palo Alto, California, August, 1973.
Hornick and Zdonik, 1987
M.F. Hornick and S.B. Zdonik, "A Shared, Segmented Memory System of an Object-Oriented Database", *ACM TOOIS*, 5, 1, 70-95, January 1987.
Hull and King, 1987
R. Hull and R. King, "Semantic Data Modeling: Survey, Applications and Research Issues", *ACM Computing Surveys*, 19, 3, 201-260, September 1987.
IBM, 1978
IBM Ltd., Internal Report on the Contents of Programs Surveyed, San Jose, California, 1978.
Jones, 1989
O. Jones, "An Introduction to X-Window System", *Prentice Hall*, Englewood Cliffs, New Jersey, 07632, 1989.
Katz and Chang, 1987
R.H. Katz and E. Chang, "Managing Change in a Computer-aided Design Database", *Proc 13th International Conference on Very Large Databases, Brighton*, 339-346, September 1987.
Kent, 1979
W. Kent, "Limitation of Record-Based Information Models", *ACM TODS*, 4, 1, 107-131, 1979.
Kerr and Cooper, 1989
D.A. Kerr and R.L. Cooper, "An Interactive Module Management System", *Persistent Programming Research Report*, Universities of Glasgow and St. Andrews, 1989.
King and Mcleod, 1984
R. King and D. McLeod, "A Unified Model and Methodology for Conceptual Database Design", in Brodie *et al.*, 1984.
Knuth 1984
D.E. Knuth, "The TEX book", *Addison-Wesley Publishing Company*, 1984.

Koch *et al.*, 1983
J. Koch, M. Mall, P. Putfarken, M. Reimer, J.W. Schmidt and C.A. Zehnder, "Modula/R Report, Lilith Version", *Tech. Report, Institute fur Informatik*, Eidgenossische Technische Hochschule Zurich, 1983.
Korth and Silberschatz, 1986
H.F. Korth and A. Silberschatz, "Database System Concepts", McGraw-Hill International Editions, 1986.
Krablin, 1985
G.L. Krablin, "Building flexible multilevel transactions in a distributed persistent environment, *Proceedings of Data Types and Persistence Workshop, Appin*, August 1985, 86-117.
Kulkarni, 1983
K.G. Kulkarni, "Evaluation of Functional Data Models for Database Design and Use", Ph. D. Thesis, University of Edinburgh, 1983.
Kulkarni and Atkinson, 1986
K.G. Kulkarni and M.P. Atkinson, "EFDM: Extended Functional Data Model", *Computer Journal*, 29, 1, 338-45, 1986.
Kulkarni and Atkinson, 1987
K.G. Kulkarni and M.P. Atkinson, "Implementing an Extended Functional Data Model using PS-algol", *Software Practise and Experience*, 17, 3, 171-185, March 1987.
Landin, 1966
P.J. Landin, "The Next 700 Programming Languages", *CACM*, 9, 3, 157-164, 1966.
Lécluse *et al.*, 1988
C. Lécluse, P. Richard and F. Velez, "O_2, an Object-Oriented Data Model", *Proceeedings of the ACM SIGMOD International Conference on Data Management Systems*, Chicago, 424-433, June 1988.
Lécluse and Richard, 1989
C. Lécluse and P. Richard, "The O_2 Programming Language", *Proceedings of the 15th VLDB Conference*, Amsterdam, August 1989.
Liskov *et al.*, 1977
B.H. Liskov, A. Snyder, R. Atkinson and C. Schaffert, "Abstraction Mechanisms in CLU", *CACM*, 20, 8, 564-576, August, 1977.
Liskov and Snyder, 1979
B.H. Liskov and A. Snyder, "Exception Handling in CLU", *IEEE Trans. on Software Engineering*, SE-5, 6, 546-558, November 1979.
Liskov and Berzins, 1979
B.H. Liskov and V. Berzins, "An Appraisal of Program Specifications", in *Research Dirctions in Software Technology* (P. Wegner, ed.), 276-301, MIT Press, 1979.
Matthews, 1985
D.C.J. Matthews, "Poly Manual", *SIGPLAN Notices*, 20, 9, September 1985.
Maier *et al*, 1986
D. Maier, J. Stein, A. Otis and A. Purdy, "Development of an Object-Oriented DBMS", *Proc ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 472-482, September-October 1986.
Meyer 1988
B. Meyer, "Object-oriented Software Construction", *Prentice-Hall International Series in Computer Science*, 1988.
Milner, 1984
R. Milner, "A Proposal for Standard ML", *Proceedings of the 1984 Symposium on Lisp and Functional Programming*, Austin, Texas, 1984.
Morrison 1981
R. Morrison, "Low cost computer graphics for micro computers", *Software Practice and Experience*, 12, 767-776, 1982.
Morrison, 1982
R. Morrison, "S-algol: a simple algol", Computer Bulletin II/31, March 1982.
Morrison *et al*, 1986a
R. Morrison, A.L. Brown, P.J. Bailey, A.J.T. Davie and A. Dearle, "A persistent graphics facility for the ICL PERQ", *Software Practice and Experience*, 14, 3, 1986.
Morrison *et al*, 1986b
R. Morrison, A. Dearle, A.L. Brown and M.P. Atkinson, "An integrated graphics programming environment", *Computer Graphics Forum*, 5, 2, 147-157, June 1986.

Morrison *et al.*, 1988a
R. Morrison, A.L. Brown, A. Dearle and M.P. Atkinson, "Flexible Incremental Binding in a Persistent Object Store", *ACM SIGPLAN Notices*, 23, 4, 27-34, April 1988.
Morrison *et al.*, 1988b
R. Morrison, A.L. Brown, R. Carrick, R.C. Connor and A. Dearle, "The Napier Reference Manual", Dept. of Computational Science, University of St. Andrews.
Morrison *et al.*, 1989
R. Morrison, A.L. Brown, R. Carrick, R.C. Connor, A. Dearle, M.J. Livesey, C.J. Barter and A.J. Hurst, "Language Design Issues in Supporting Process-Oriented Computation in Persistent Environments", *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, (ed. B. D. Shriver), vol II, Software, 736-744, January 1989.
Myers and Buxton, 1985
B.A. Myers and W. Buxton, "Creating Highly-Interactive and Graphical User Interfaces by Demonstration", *Proceedings of ACM SIGGRAPH*, 20, 4, 249-258, 1986.
Mylopoulous *et al.*, 1980
J. Mylopolous, P. A. Bernstein and H.K.T. Wong, "A Language Facility for Designing Database-Intensive Applications", *ACM TODS*, 5, 2, 185-207, 1980.
Nestor, 1986
J.R. Nestor,"Towards a Persistent Object Base", *Proceedings of the International Workshop in Advanced Programming Environments, Trondheim, Norway*, (R.Conradi, T. M. Didriksen and D. H. Wanvik eds.), Springer Verlag Lecture Notes in Computer Science 244, 372-394, June, 1986.
Norrie 1985
M.C. Norrie, "The Edinburgh Node of the Proteus Distributed Database System", *Department of Computer Science Report CSR-191-85*, University of Edinburgh, 1985.
O'Brien *et al*, 1987
P.D. O'Brien, D.C. Halbert and M.F. Kilian, "The Trellis Programming Environment", *Proceedings OOPSLA*, Orlando, Forida, October 1987.
Odesta, 1984
ODESTA Corporation, "Helix: A Data-based Information Management and Support System", 4084 Commercial Ave., Northbrook, Illinois 60062, 1984.
Ohori *et al.*, 1989
A. Ohori, O.P. Buneman and V. Breazu-Tannen, "Database Programming in Machievelli", *Proceeedings of the ACM SIGMOD International Conference on Data Management Systems*, Portland, 424-433, May-June 1989.
Ontologic 1986
Ontologic Inc, "Vbase Object Manager User Manual", 47 Manning Rd., Billerica, Mass 01821, November 1986.
Oracle, 1983
Oracle Corporation, "The Oracle Users' Guide", Thames Link House, 1 Church Rd., Richmond Surrey, TW9 2QE, 1983.
Owoso, 1984
G.O. Owoso, "Data Description and Manipulation in Persistent Programming Languages", Ph. D. Thesis, University of Edinburgh, 1984.
PCTE, 1986
PCTE, "A Basis ofor a Portable Common Tool Environment: Functional Specification 4th Edition", 1986.
Philbrow *et al.*, 1988a
P. Philbrow, "A Print Facility for PS-algol", 1988.
Philbrow *et al*, 1988b
P. Philbrow and M.P. Atkinson, "Exception Handling in a Persistent Programming Language", *The Computer Journal*, 1988.
Philbrow *et al.*, 1989
P. Philbrow, D.J. Harper and M.P. Atkinson, "Supporting an Object-Oriented Programmming Methodology using PS-algol", *Proceedings of the 2nd International Workshop on Database Programming Languages*, Oregon, 313-330, June 1989.
PS-algol, 1987
"The PS-algol Reference Manual - Fourth Edition", *Persistent Programming Research Report 12*, Universities of Glasgow and St. Andrews, 1987.
Richardson and Carey, 1987
J. Richardson and M.J. Carey, "Programming Constructs for DatabaseSystem Implementation in EXODUS", *Proceeedings of the ACM SIGMOD International Conference on Data Management Systems*, San Francisco, 208-219, May 1987.

Rochkind 1975
M.J. Rochkind, "The Source Code Control System", *IEEE Transactions on Software Engineering*, 1, 4, 364-370, December 1975.
Ross, 1977
D.T. Ross, "Structured Analysis (SA): A Language for Communicating Ideas", *IEEE Transactions on Software Engineering*, 3, 16-34, December 1977.
Rowe and Shoens, 1979
L. Rowe and K.Shoens, "Data Abstraction Views and Updates in Rigel", *Proceeedings of the ACM SIGMOD International Conference on Data Management Systems*, Washington D.C., 71-81, May 1979.
Rowe and Stonebraker
L. Rowe and M. Stonebraker, "The POSTGRES Data Model", *Proceedings of the 13th VLDB*, 83-97, 1987.
Schmidt, 1977
J.W. Schmidt, "Some High-level Language Constructs for Data of Type Relation", *ACM Transactions on Database Systems*, 2, 3, 247-261, 1977.
Schmidt and Mall, 1983 - DBPL should be
H. Eckhardt, J. Edelmann, J. Koch, M. Mall and J.W. Schmidt, "Draft Report on the Database Programming Language, DBPL", Johann Wolfgang Goethe - University, Frankfurt am Main, West Germany, 1985.
ServioLogic 1987
ServioLogic Corporation., "Programming in OPAL", 15025, S.W. Koll Parkway, 1A, Beaverton, Oregon 97006, 1987.
Shipman, 1981
D.W. Shipman, "The Functional Data Model and the Data Language DAPLEX", *ACM TODS*, 6, 1, 140-173, 1981.
Smith, 1987
R.B. Smith, "The Alternative Reality Kit: An Animated Environment for Creating Interactive Simulations", CHI+GI, *Proceedings of IEEE Conference on Human Computer Interaction and Graphical Interfaces*, 99-106, 1987.
Smith and Smith, 1977
J.M. Smith and D.C.P. Smith, "Database Abstractions: Aggregation and Generalization", *ACM TODS*, 2, 2, 105-134, 1977.
Smith *et al.*, 1983
J.M. Smith, S. Fox and T. Landers, "Adaplex: Rationale and Reference Manual - Second Edition", Computer Corporation of America, Cambridge, Mass., 1983.
Stefik and Bobrow, 1985
M. Stefik and D.G. Bobrow, "Object–Oriented Programming: Themes and Variations", *The AI Magazine*, 40–62, 1985.
Stemple, 1989
D. Stemple, "Exploiting the Potential of Persistent Object Store", *Proceedings of the Workshop on Persistent Object Systems, their Design, Implementation and Use*, (J.Rosenberg ed.), Newcastle, New South Wales, 328-342, January 1989.
Stocker, 1973
P. Stocker and P.A. Dearnley, "Self Organising Data Management Systems", *The Computer Journal*, 16, 2, 100-105, 1973.
Stonebraker *et al.*, 1976
M. Stonebraker, E. Wong, P. Kreps and G.D. Held, "The Design and Implementation of INGRES", *ACM TODS*, 1, 3, 189-222, 1976.
Strachey, 1967
C. Strachey, "Fundamental Concepts in Programming Languages", *Oxford University Pres*, 1967.
Stroustrup 1984
B. Stroustrup, "The C++ Programming Language", *Addison Wesley*, Reading, Mass., 1984.
Swartout and Balzer, 1982
W. Swartout and B. Balzer, "On the Inevitable Intertwining of Specification and Implementation", *CACM*, 25, 7, 438-440, July, 1982.
Tennent, 1981
R.D. Tennent, "Principles of Programming Languages", *Prentice-Hall International Series in Computer Science*, 1981.

Tichy 1985
W.F. Tichy, "RCS - A System for Version Control", *Software Practise and Experience*, 15, 7, 637-654, 1985.
Trinder and Wadler, 1989
P. Trinder and P. Wadler, "Improving List Comprehension Database Queries", *Proceedings of TENCON '89*, Bombay, India, November 1989.
Unilogic, 1985
UNILOGIC Ltd, "The SCRIBE Document Production User Manual", 1984
Wadler and Blott, 1989
P. Wadler and S.M. Blott, "How to Make *ad-hoc* Polymorphism Less *ad-hoc*", *Proceedings of the 16th ACM Symposium on the Principles of Programming Languages*, Austin, Texas, January 1989.
Wai, 1988
F. Wai, "Distributed Concurrent Persistent Programming Languages: An Experimental Design and Implementation", PhD Thesis, University of Glasgow, 1988.
Warren, 1988
The Warren Committee, "Parliamentary Trade and Industry Committee First Report on Infromation Technology", House of Commons Paper 25.1, HMSO 1988.
Wasserman *et al.*, 1981
A.I. Wasserman, D.D. Shertz, M.L. Kersten, R.P. Reit and M.D. van der Dippe, "Revised Report on the Programming Language, PLAIN", *ACM SIGPLAN Notices*, 1981.
Welland, 1989
R. Welland - private communication, 1989.
Zdonik, 1986
S.B. Zdonik , "Version Management in an Object-oriented Database", *Proceedings of the International Workshop in Advanced Programming Environments, Trondheim, Norway*, (R.Conradi, T. M. Didriksen and D. H. Wanvik eds.), Springer Verlag Lecture Notes in Computer Science 244, 405-422, June, 1986.
Zloof, 1977
M.M. Zloof - "Query-by-Example, A New Data Base Language" in *IBM Systems Journal*, 16, 4, 324-344, 1977