THE ENGINEERING OF DATA STRUCTURES

ADRIAN COLBROOK, B.Sc.

Thesis submitted for the degree of Doctor of Philosophy in the Department of Electronic and Electrical Engineering at the University of Surrey.

July 1990

© Adrian Colbrook 1990
Abstract

Abstraction in computer programming provides a means of reducing complexity by emphasising the significant information (program behaviour) whilst suppressing the immaterial (program implementation). This aids program construction, improves reliability and maintainability, and eases the application of formal correctness proofs. The importance of data abstraction in the specification, design and implementation of large systems raises the question as to whether such methods may be applied in the context of programming languages designed before the widespread use of abstraction techniques.

The program structuring facilities available in FORTRAN 77 support a form of encapsulation for simple data structures. In light of this mechanism provided by the language, state-based specification was found to be most appropriate. A specification technique incorporating object-oriented techniques is particularly suitable and allows a library of object classes to be specified and then implemented in sequential FORTRAN 77. Refinement extends the object classes so as to provide the commonly occurring generators for use in iterative constructs. Therefore, the advantages of data abstraction methods may be obtained in an early procedural language such as FORTRAN 77.

Data abstraction provides data independence: a change in the representation for a particular class of objects affects only the code that implements the associated operations. This allows parallel implementations to be considered, without changes to the original specification or to any user-code. The provision of such parallel data structures is required for the migration of sequential systems onto parallel distributed memory architectures.

As an illustration of this approach a general $2^{P-2}$--$2^P$ (for integer $P \geq 3$) search tree utilising a pipeline of processors in a distributed memory architecture is shown to provide a means of implementing the object classes. Variations in both the number of processors allocated to the pipeline and the value of $P$ allows the optimal search structure for a given architecture to be determined. These structures are highly efficient leading to improvements in both throughput and response time as processors are added to the array. An efficient parallel implementation of object classes is therefore achieved within the tight interface provided by abstraction.
In memory of James William Milton
Acknowledgements

As in most cases of human endeavour the completion of this project is due to the help of many people and organisations. The author should like to thank the Science and Engineering Research Council and Thorn EMI Central Research Laboratories for providing financial support. Thanks must go to the scientific staff of the Advanced Computing Section at Thorn EMI Central Research Laboratories, particularly Dr J.S. Severwright, who permitted the use of their computing facilities.

Acknowledgements are also due to the academic staff of the Department of Electronic and Electrical Engineering and the Department of Mathematics and Computer Science, University of Surrey. In particular Professor B. Cohen for the many valuable comments he has made; Dr D.H. Pitt for numerous discussions; and Dr C. Smythe and Professor S.A. Schuman for their guidance and inspiration.
## Contents

### Chapter 1  Introduction and Overview
1.1 Abstraction 1
1.2 Abstraction and FORTRAN 77 2
1.3 Multi-Processor Architectures 3
1.4 Programming Issues for Distributed Memory Architectures 5
1.5 Thesis Overview 7

### Chapter 2  Abstraction in Programming Languages
2.1 Abstraction in Programming 9
2.2 Data Abstraction 10
2.3 Abstraction and FORTRAN 77 12
2.4 Data Structure Libraries 15
2.5 Formal Specification for Data Abstraction 16
2.6 Concurrency and Data Abstraction 20

### Chapter 3  State-Based Specification
3.1 The Object-Oriented Set-Theoretic Specification Technique 22
3.2 Refinement of a Specification 29

### Chapter 4  Specification and Implementation of the Object Classes
4.1 Specification and Validation 32
4.2 Correctness of Usages 34
4.3 Correctness of an Implementation 35
4.4 An Alternative Representation 37
4.5 Multiple Instances and Refinement 40

### Chapter 5  Formal Data Refinement
5.1 Class Behaviour and History 46
5.2 Refinement Towards Implementation 47
5.3 Extraction Towards Implementation 51
5.4 An Alternative Representation 58
| Appendix B | Proof of Consistency for the Set Specification | 143 |
| Appendix C | Verification for the Set Data Structure | 148 |
| Appendix D | The Stack, Queue, Abstract Array and Binary Relation Object Classes | 156 |
| D.1 | The Stack Object Class | 156 |
| D.2 | The Queue Object Class | 157 |
| D.3 | The Abstract Array Object Class | 158 |
| D.4 | The Binary Relation Object Class | 160 |
CHAPTER 1  INTRODUCTION AND OVERVIEW

This thesis presents the introduction of data abstraction into FORTRAN 77 without extensions to the language’s syntax or semantics. FORTRAN 77 has received criticism from many quarters but continues to be used. Large FORTRAN 77 libraries have been built over a number of years and these represent significant resources. Many programmers have experience with the language and the majority of high-speed parallel computing is carried out using FORTRAN 77. With such an investment, the creation and maintenance of FORTRAN programs will continue to be a significant issue [Miller, 1988].

Abstraction reduces complexity and therefore improves the constructability, reliability, understandability and maintainability of software systems as well as easing the application of formal program correctness proofs. In addition, the independence of the representation of the abstractions from their use allows parallel implementations to be considered. It is proposed that this will aid in the migration of sequential FORTRAN 77 systems onto parallel multi-processor architectures.

1.1  Abstraction

"In the development of our understanding of complex phenomena, the most powerful tool available to the human intellect is abstraction" [Hoare, 1972b]. Abstraction is the process of ignoring certain details of objects, situations or processes and concentrating instead upon the similarities which are present. Those similarities which are relevant to the prediction and control of future events may be regarded as fundamental and constitute an abstract concept covering the set of objects, situations or processes in question. Abstraction therefore, provides a means of reducing complexity by emphasising the significant information whilst suppressing the immaterial.

When applied to computer programming, abstraction allows program behaviour to be emphasised and implementation details to be suppressed. Its most significant application is the development of high-level languages. Programmers achieve significant simplification through dealing with the constructs of a high-level language rather than the sequence of machine instructions into which such constructs translate.

The abstraction provided by a high-level language is only sufficient for reducing the complexity of small software systems to a manageable level. Means of achieving greater levels of abstraction are required for larger systems. One approach could be the development of very-high-level languages in which programs are built around general data
structures, which are manipulated by a set of powerful primitives. However, the set of all such data structures and primitives which may be required would make the language unwieldy. A more practical approach is the provision of language mechanisms which enable programmers to construct their own abstractions. Three mechanisms may be supported: computational abstraction, data abstraction and control abstraction.

Computational abstraction, embodying procedural and functional abstraction, allows the virtual machine defined by a programming language to be extended by the addition of a new operation. Such extensions are most useful when the problem decomposition results in a number of independent functional units. Data abstraction allows new kinds of data types or classes to be added to the virtual machine. The behaviour of these types or classes is expressed in terms of a set of operations. Control abstraction adds new control flow mechanisms to the virtual machine. An example is iterative abstraction, where the selection of individual objects from a collection is separated from the action to be performed on each object in a loop.

The design of programming languages has a major impact upon the effectiveness of such abstractions. The extent to which a particular method can be applied in a given language depends upon how well the language constructs match the structures that are being abstracted. Early languages such as FORTRAN and COBOL support only limited procedural abstraction explicitly. The languages CLU (Liskov, 1977), Alphard (Wulf, 1976) and Euclid (London, 1978) were amongst the earliest languages to include mechanisms for both data and control abstraction. Many of the more recent programming languages such as Modula-2 (Wirth, 1983) and Ada (Ada, 1983) include data abstraction facilities.

### 1.2 Abstraction and FORTRAN 77

FORTRAN 77 lacks many of the features considered desirable in programming languages, such as nested procedures, a user-definable typing mechanism and explicit support for data abstraction, although limited procedural abstraction is provided. The introduction of data and control abstraction into FORTRAN 77 would go some way towards overcoming these drawbacks by providing the following benefits:

- The development of new systems may proceed by problem decomposition based upon abstraction, resulting in programs which are better structured and therefore more reliable and understandable as well as easier to modify and maintain;
- The operations upon the abstraction can be defined in a rigorous mathematical fashion through formal specification. This means that the abstraction itself is a well defined mathematical system and it is possible to verify that the representation meets its specification. With increasing emphasis now being
placed upon the correctness of software, particularly in safety-critical and parallel systems, language constructs which readily support the use of formal notations are required;

- Data abstraction provides data independence: a change in the representation of the abstraction has to affect the code which implements the associated operations, but all changes are localised to just that section of code. This allows a number of alternative implementations to be considered, without changes to the original specification or to any user-code. One such alternative is a parallel implementation, which permits a number of queries to the data structure to execute simultaneously and may result in a significant improvement in query throughput.

The implementation of data and control abstraction in FORTRAN 77 has been considered in the literature and was achieved through the use of coding conventions [Ford, 1985], language preprocessors [Miller, 1988] or the encapsulation mechanisms provided by the language [Isner, 1982]. Coding conventions do not provide encapsulation and rely solely upon the good will of the programmer not to change the representation of the abstractions in an illegal fashion. Language preprocessors provide rigorous encapsulation and type checking but restrict the portability of any code to those systems with identical preprocessors. In addition, FORTRAN 77 programmers must become familiar with the semantics of new data types and structures. The use of the encapsulation mechanisms provided by the language places severe restrictions upon the kinds of abstraction which can be supported. However, sufficient encapsulation of abstract data structures can be achieved within standard FORTRAN 77. This is the approach that will be pursued here.

The specification and implementation of abstractions is a non-trivial exercise, and an unfamiliar one to the FORTRAN 77 programmer. For these reasons the support of a library of the commonly occurring data structure is a recommended solution [Guttag, 1980]. Such a library would allow parallel implementations of the data structure to be introduced as appropriate, aiding in the migration of sequential systems onto multi-processor architectures.

1.3 Multi-Processor Architectures

The mainstream use of computers is increasing in sophistication to a point where single processor systems are unable to cope with the demand [Hwang, 1984]. Multi-processor architectures offer a number of advantages over their single processor counterparts, which include increase in program throughput, improved levels of system availability and ease of incremental growth. Architectures may be classified in terms of the potential multiplicity of
hardware used in manipulating the instruction and data streams of the computer [Flynn, 1966]. Although this is rather vague, the classifications made may be subdivided still further so as to partition parallel systems.

Asynchronous multi-processor architectures consist of a number of separately programmable units and are the most generally applicable parallel systems. They may be divided into shared memory multi-processor and distributed memory multi-processor architectures [Quinn, 1987].

Shared memory multi-processor architectures are characterised by a shared address space between the processing elements. This is usually provided by a shared global memory, which is accessed by a central switching mechanism [Fuller, 1976] [Gottlieb, 1983]. Alternatively, the shared address space may be formed by combining the local memories of the processing units [Stone, 1978].

Distributed memory architectures have no shared global memory. Each processor has its own local memory and processors communicate through message passing or memory shared by processor pairs. Commercially available systems include Intel's iPSC [Pase, 1988] and Thorn EMI's Supernode [Harp, 1987].

Kallstrom and Thakkar [Kallstrom, 1988] contrasted programming issues for the iPSC, a network of transputer processors (a distributed memory architecture) and the Sequent Balance [Thakkar, 1988] (a shared memory architecture). A program containing inherent parallelism may be partitioned into a number of concurrent modules for execution on a parallel system. The execution time of a module is termed the granularity of that module. Kallstrom and Thakkar demonstrated that shared memory architectures are suited to small granularity parallel applications whereas the distributed memory architectures perform well with larger grains and reduced communication overheads. In shared memory architectures, system expansion is limited by the requirement for a shared communication bus between the processing elements and the shared memory. The number of processing elements which can be usefully put together in such systems is restricted to not more than about thirty [Otto, 1989]. For distributed memory architectures, decomposition of the problem between processing elements and load balancing of dynamically fluctuating workloads present major difficulties [Otto, 1989]. These are largely caused by the need to distribute the program data across processing elements in order to reduce the inter-processor communication overheads. Despite these drawbacks, distributed memory multi-processor systems show an increasing promise of improved performance over a wide range of applications, including physics and engineering applications [Hey, 1987], image processing, graphical imaging and simulation [Hey, 1988].
1.4 Programming Issues for Distributed Memory Architectures

In 1983 Halsall, Grimsdale, Shoja and Lambert [Halsall, 1983] noted the lack of suitable software tools to aid the development of application software for parallel systems. Since then, the situation has not significantly improved — "The general reluctance to redesign and rewrite existing code, and a lack of familiarity with transputer technology, occam and suitable multi-process program design methodologies currently presents a significant barrier to the widespread general use of transputer systems in engineering environments in industry and academia" [Robinson, 1988].

McGraw and Axelrod [McGraw, 1988] identified four ways in which applications software could be written for multi-processor architectures during the next five to ten years:

- extending existing languages with new operations that allow concurrency and synchronisation to be expressed;
- extend existing compilers to identify concurrent operations where possible and add the necessary synchronisation to maintain program correctness;
- add a new language layer on top of an existing language to describe the desired concurrency and necessary synchronisation while allowing the basic applications program to remain relatively unaltered;
- define a totally new language and compiler system that integrate the concepts of concurrency and synchronisation with all of our existing views of describing computational algorithms.

Extending existing languages (usually FORTRAN 77 or C [3L, 1988a] [3L, 1988b]) typically requires support for the creation and termination of parallel processes, synchronisation of parallel processes and identification of shared data. In many cases these extensions can be added solely through new library routines, allowing existing applications to be converted to run on multi-processor architectures with minimal change. However, difficulties arise from the compiler’s lack of knowledge about how parallelism is being used. This makes it impossible to analyse how data is shared between processes, which restricts optimization and debugging.

One of the major areas of effort has been the development of “parallel-extracting” compilers to aid in the migration of existing sequential programs [Robinson, 1988]. Such compilers take sequential language source code and generate parallel code to take advantage of a particular machine architecture. Most of the work in this area has centred around shared memory
multi-processor architectures with the major language supported being FORTRAN 77 [Kuck, 1981] [Kamiya, 1983] [Scarborough, 1986]. Software tools for languages are also available [Davies, 1986], [Brode, 1981] which utilise data-dependency and control-flow analysis to restructure source code to a parallel form. The problems encountered when taking sequential code and running it successfully on a parallel architecture were highlighted by Karp and Babb [Karp, 1988] in their comparison of twelve parallel FORTRAN dialects. The conversion of a simple pi approximation program to run on nine commercially available multi-processor architectures is shown to be a surprisingly complex task. Most compilers concentrate their efforts upon the fine grain parallelism associated with looping constructs [Padua, 1987] and Karp and Babb concluded that it will be several years before compilers can capture all the parallelism in a sequential program for a shared memory multi-processor architecture.

For distributed memory systems the problem is more severe as there is no compiler technology available that will map the program data to the distributed memory of these machines. Compilers must be capable of recognising other forms of parallelism than merely the vector forms so far detected [Kamiya, 1983]. Therefore, the conversion of existing software for distributed memory architectures is unlikely to be totally automated for quite some time.

Language layering for concurrency [McGraw, 1988] distinguishes between the parallel structure or skeleton of the code and the details of the computation. A new language specifies the skeleton while an existing language specifies the computation. For the skeleton language to be useful, it must be possible to specify not only synchronisation interactions between processes but also data partitions, but it is not clear how best to do this. Also, as is always the case with preprocessor implementations, debugging becomes more difficult.

Defining new languages and compiler systems is the most expensive and labour intensive option. Such languages include the applicative languages SISAL [McGraw, 1983], FP [Backus, 1978] and KRC [Tumer, 1981], object-oriented programming method [Yonezawa, 1987] [Bershad, 1988] and others such as CHILL [CCITT, 1984] and Argus [Liskov, 1987]. In these languages and systems concurrency is controlled by the language definition to ensure determinate behaviour. Hey [Hey, 1989] commented on this alternative - “it is undoubtedly true that new languages can and will make the effective exploitation of parallelism much easier and more controllable. However, there is an economic fact of life that cannot be ignored, namely, that large companies with many hundreds of man-years invested in sequential software will not invest substantially in parallel hardware until there is a convincing ‘migration route’ to such machines.”
Although the level of work in the development of new languages and systems will increase with the growing demand for parallel architectures, extending existing languages remains the most viable option for the migration of sequential applications. In these systems, data structures should be shared between processors only after careful consideration. Where data is to be shared between independent processors, methods of synchronising access to preserve consistency and supporting the parallel execution of queries must be found. These tasks could be left to the individual processors accessing the structure, but this would lead to increasing overheads and necessitate the programmer being concerned with such detail. A far more favourable alternative is the use of abstraction with the details of synchronisation, consistency, and parallel access being hidden from the user. In addition to the parallel library routines required for synchronisation, inter-process communication, and numerical computation, a library of parallel data abstractions should be supported to aid in the sharing of data between asynchronous processors.

1.5 Thesis Overview

This thesis describes the introduction of a restricted library of abstract object classes into FORTRAN 77, a language with explicit support for only limited procedural abstraction. A number of object classes are identified, specified using an appropriate formal notation and implemented in standard FORTRAN 77. Since the representation of an object class is independent of any user-code, several alternative implementations are possible and these include parallel implementations. By way of example, a search tree algorithm is developed for a pipeline of processing elements in a distributed memory multi-processor architecture. This search tree is shown to provide means of implementing object classes, resulting in parallel implementations of data structures, which are encapsulated within the tight interface provided by abstraction.

Chapter two is a survey of abstraction techniques and includes a review of the relevant literature. The methods used to date for introducing data abstraction into FORTRAN 77 are described together with the formal notations used to specify abstractions.

In light of the abstraction mechanism provided by the language, it is argued that a state-based formal specification notation is most appropriate, namely the object-oriented set-theoretic technique of Schuman and Pitt [Schuman, 1987]. This notation is introduced by way of example in Chapter three.

Chapter four presents the specification and implementation of a set data structure by way of example. The proof obligations which ensure the consistency of the specification and verify that the implementation meets its specification are described. The specification is then refined to introduce multiple objects of the class.
An alternative to the classical approach of specification followed by implementation and verification is given in Chapter five. The object-oriented set-theoretic specification technique permits the implementation of a class to be derived through refinement. This technique is demonstrated for the set data structure.

Chapter six reviews iterative abstraction. Two iterative constructs over data structures are identified and the corresponding generator over the set object class is specified by refinement. The implementation in standard FORTRAN 77 is given and binary operations are introduced.

Chapter seven introduces parallel implementations of data structures on distributed memory multi-processor architectures by a survey of the different approaches available. A processor array implementation of a tree data structure is identified as being optimal and a $2^P-2^P$ (integer $P \geq 3$) search trees is described, which allow insertion and deletion operations to proceed in a top-down fashion.

Chapter eight presents a formal specification of the $2^P-2^P$ search tree. This allows the pipeline operation and concurrent composition of operations to be demonstrated.

The search tree provides a means of implementing the set object class and allows the introduction of generators and multiple instances. Chapter nine presents the implementation of sets using the $2^P-2^P$ search tree on a pipeline of transputer processors. Variations in the value of $P$ and the number of processors allocated to the pipeline allow an optimal structure to be identified. The throughputs and response times are compared and the structure is shown to be highly efficient.

Chapter ten is a general conclusion together with a review of further work.

A number of Appendices are included. In Appendix A a comprehensive glossary of the mathematical notation used throughout the thesis is given. In Appendix B the proof of consistency for the specification of the set object class is presented. Appendix C contains the verification that the implementation of the set object class satisfies the specification. Appendix D presents the specifications and implementations of the object classes (other than the set) which are contained in the candidate library of data structures. Appendix E presents the proofs that transformations applied to the $2^P-2^P$ search tree preserve the tree structure.
CHAPTER 2 ABSTRACTION IN PROGRAMMING LANGUAGES

This chapter presents a survey of abstraction in programming languages together with a review of the formal specification techniques available, which includes an assessment of their relative merits and areas of application. A brief outline of the use of abstraction in parallel systems is also included.

2.1 Abstraction in Programming

"The amount of complexity that the human mind is able to cope with at any instant in time is considerably less than that embodied in much of the software one might build" [Dijkstra, 1972]. Therefore it is necessary to reduce the amount of complexity that must be considered and this may be achieved through the use of abstraction. The purpose of abstraction in programming is to separate behaviour from implementation. Abstraction is the process of ignoring certain details in order to simplify the problem and was described by Shaw [Shaw, 1984] as "...a simplified description of a system that emphasises some of the system details or properties while suppressing others. A good abstraction is one in which information that is significant to the reader is emphasised, and details that are immaterial or diversionary, at least for the moment are suppressed."

Abstraction enables the specification, design and implementation of a system to be carried out in a step-wise fashion and may be applied to computations (functional or procedural), data (value or object) or control flow.

Procedures and functions were the first programming abstraction mechanisms and facilitated the description of abstract events. Procedures are inherently state-based, performing a computation to query and/or update the state of the system. In contrast, functions are value-based, performing a computation to yield a value which is derived from the set of input values. The abstraction mechanism introduced by procedures and functions [Zilles, 1973] led to the modular programming methodology [Panas, 1972b], where a program is organised around the connections between its modules. From this arose data abstraction [Hoare 1972a] [Liskov, 1974].

Data abstraction allows the description of abstract types or classes as opposed to events, their behaviour being defined in terms of operations. Guttag's [Guttag, 1977a] [Guttag, 1980] discussion of the distinct attributes which are associated with data structures provides a clear view of the use of abstraction. Attributes may describe the representation of structures
and the implementation of the operators associated with the structures, or may specify names and define the abstract meaning of the operators associated with a structure. Given a certain level of refinement, it is possible to require knowledge only of the names and abstract meaning of the operators. This leads to the representation being encapsulated by the set of operations which manipulate it and using programs are restricted so that they cannot access the representation directly. Encapsulation of this nature is a feature of programming languages which support data abstraction [Hanson, 1979].

Two distinct “schools of thought” have emerged as to the proper basis for applying data abstraction and this division represents a fundamental difference between values (value-based) and objects (state-based) [Schuman, 1987]. In the case of values system decomposition is based upon “abstract data types”, which characterise the domain of interest in terms of constructor and selector functions. The abstraction then involves specifying an equality relation over the values. In contrast, “class” of abstract objects, in contrast to a “type” of values, serves to encapsulate the definition of some internal state in conjunction with an associated set of access operations for querying and/or updating any individual instance of that class.

Control abstraction defines a method for sequencing arbitrary actions. All imperative languages provide built-in control abstractions such as the familiar while statement. However, many looping constructs perform an action on all the objects in a given collection. Iterative abstraction [Liskov, 1986] [Bishop, 1986] [Bishop, 1990] permits the selection of the objects to be separated from the action to be performed. A differentiation may be made between control flow and control computation in a loop [Pratt, 1978]. Control flow gives details of where a loop body begins and ends, and identifies any additional entry and exit points. Control computation identifies the variables used within the loop, in particular variable initialisation, assignment and where variables are checked and terminated. Generators supply the control computation required for data abstraction by providing the interface through which control-flow constructs can access the data in a non-destructive and uniform manner. The encapsulation property, which is so fundamental to the data abstraction process, is preserved.

2.2 Data Abstraction

Data abstraction originated from earlier considerations of data types in general. In 1973, Morris [Morris, 1973] disputed the then prevailing description of a data type as merely a set of values. He reasoned that a type is a language mechanism to enforce authentication and security and includes operations as well as abstract entities. For example, the type integer includes a bounded set of integer values as the abstract entities together with the set of integer operations such as addition, subtraction, multiplication and integer division.
Chapter 2  
Abstraction in Programming Languages

An abstraction is defined by a specification and implemented by a program module. The specification describes what the abstraction does but omits any information concerning the implementation. The major advantages of data abstractions (both value-based and state-based) are [Berztiss, 1983] [Bishop, 1986] [Liskov, 1986]:

- The operations can be defined in a rigorous mathematical fashion by the specification, which means that the data abstraction itself is a well defined mathematical system. A systematic development of a body of knowledge is thus made possible;

- the representation is encapsulated by the set of operations which manipulate it and user programs are restricted so that they cannot manipulate the representation directly. This separation of representation from usage issues results in data independence: a change in the representation of the data abstraction has to affect the code that defines the operations, but all changes are localised to just that section of code. Provided that development is achieved by means of problem decomposition [Liskov, 1977] based upon the recognition of abstraction, the result is programs which are effectively constructed, reliable [Gannon, 1975] [Hoare, 1975] [Gannon, 1977], understandable, and easy to modify and maintain [Linden, 1976];

- The maintenance of existing software involves modification to correct faults, improve performance or to adapt the product to a changing environment. This is the single most expensive phase of the software life cycle [Arthur, 1988]. The introduction of abstraction in the development of new systems, or retrospectively into existing systems, eases the maintenance task because:
  - changes to an abstract representation are restricted to only the code which implements the associated operations
  - abstraction eases the process of relating program action to program code
  - the prediction of possible spurious behaviour following a change is simplified;

- A mathematical definition of a data abstraction ensures that the implementor knows precisely what is to be implemented. Indeed it is possible to verify formally that the code meets its specification [Dennis, 1975] [Liskov, 1975a] [Jones, 1979] [London, 1979] [Ford, 1985];

- The locality which results from abstraction being supported by specification and encapsulation allows a program to be implemented, understood or modified one module at a time. This supports fast prototyping and program evolution [Liskov, 1988].
Much of the work in data abstraction has centred around the development of new languages, formal specification techniques and program verification. Simula [Dahl, 1968] [Dahl, 1972] [Birtwistle, 1973] was the first programming language to support the association of abstract operations with entities via its class construct. However, entities could be accessed directly without the use of the associated operations [Geschke, 1975] and the language Mesa [Geschke, 1977] was developed as an attempt to overcome this problem. Parnas [Parnas, 1972a] recognised the need for encapsulation to prevent accessing the representation directly and the languages CLU [Liskov, 1977] [Liskov, 1981], Alphard [Wulf, 1976] [Shaw, 1981] and Euclid [London, 1978] were among the earliest languages to enforce this restriction upon abstract data types. Many of the more recent programming languages such as Modula-2 [Wirth, 1983] and Ada [Ada, 1983] include data abstraction facilities. The influence of data abstraction upon the design of programming languages is continuing with the on-going research into object-oriented languages for example Smalltalk80 [Goldberg, 1983] and C++ [Stroustrup, 1985]. Some functional languages, such as Hope [Burstell, 1980b], also provide data abstraction and MacQueen [MacQueen, 1988] demonstrated the use of data abstraction and modules in structuring ML [Harper, 1986] programs. However, the implementation of data abstraction in early languages such as FORTRAN has not been so widely considered.

2.3 Abstraction and FORTRAN 77

FORTRAN has been extended so as to support non-standard arithmetic packages and data types including multiple or extended precision arithmetic [Brent, 1978] [Wyatt, 1976] and interval arithmetic [Yohe, 1979]. These extensions were provided through the use of precompilers. Precompilers to extend the data types in FORTRAN were first designed to add a specific type or group of related types to the language, for example significant arithmetic [Bright, 1968] and interval arithmetic [Knowlton, 1968] [Knowlton, 1970]. The general purpose precompiler Augment [Crary, 1979] furnished programmers with a means of using non-standard arithmetic by supporting the definition of additional data types. The interaction between a non-standard data type and other types (both standard and non-standard) was defined by the user. Burton [Burton, 1979] introduced Classy FORTRAN, an extended FORTRAN supporting data abstraction definitions through the use of a precompiler and an extension to the language, which provided the dynamic allocation of storage. Isner [Isner, 1982] proposed a technique based upon the encapsulation mechanisms supported by the language for simple data abstractions. However, for more complex structures a heap was required, from which storage was dynamically allocated. The requirement for dynamic storage allocation together with the associated pointer variables was problematic for three reasons:
Chapter 2  Abstraction in Programming Languages

- Existing FORTRAN libraries have to be rewritten before they can be used to implement abstract data types. If such a rewrite takes place, it may be more cost effective to rewrite in a language that already supports pointer variables and data abstraction features;

- Most FORTRAN environments do not include pointer variables. Isner’s (Isner, 1982) assumption that a FORTRAN compiler includes heap management is therefore not normally the case;

- Abstract data type users and implementors must learn the semantics of pointer variables. This will require retraining and the more extensive this retraining is, the less advantage is realised by retaining FORTRAN as the language of choice.

A further system, FAD (FORTRAN Abstract Data) has been designed (Miller, 1988) to support and enforce data abstraction through the use of a preprocessor. In contrast to the systems proposed in (Burton, 1979) and (Isner, 1982), no pointer variables were introduced. Another system, the Object Programming Applications Language (OPAL) (Zelkowitz, 1987) was originally used with FORTRAN code and was based upon run-time objects. This system required heap management and is now an Ada-based system.

In their discussion of FAD, Miller, Morell and Stevens compared the three different implementation methodologies for data abstraction: coding conventions (value-based), a preprocessor (value-based) or the encapsulation mechanisms supported by the language (state-based). Coding conventions (Ford, 1985) rely upon abstraction being maintained through the imposition of programming standards. Representation information is required before an abstract data type can be used and no restrictions are placed upon the user’s direct manipulation of the representation. Preprocessors automatically enforce information hiding but suffer from four major disadvantages (Miller, 1988):

- A preprocessor requires machine resources for translation;

- It may require the programmer to use additional notation;

- Compiler listings refer to generated source code instead of the original code;

- Code portability relies upon the preprocessor being available at the target site.

The use of the encapsulation mechanisms (Isner, 1982) provided by standard FORTRAN 77 (ANSI, 1978) offers several advantages:

- code portability is not compromised as no use of preprocessors or non-standard language extensions is made;
• the standard language compiler enforces much of the required encapsulation;

• the approach appeals to the existing knowledge of FORTRAN 77 programmers as no new data typing mechanism need be introduced;

• an interface between FORTRAN 77 code and data abstractions is established and this may be exploited by a variety of representations including parallel implementations.

In this approach object classes as opposed to data type extensions are introduced as there are no user-definable typing mechanism in FORTRAN 77. Binary operations upon two objects (set union for example) can be supported via procedures and generator constructs.

Both the language of implementation and the method of implementation within the language dictate the semantics of the abstract operations which may be applied to a data structure. Isner's [Isner, 1982] technique suffers from two drawbacks, a decrease in program efficiency and the necessity of using a relatively dangerous construct to achieve encapsulation (the FORTRAN 77 multiple entry point procedure). Despite these limitations, he [Isner, 1982] reported that a low degree of module coupling was achieved as a result of its application leading to increased system reliability and maintainability.

By using the FORTRAN 77 multiple entry procedure facility together with the save statement [ANSI, 1978] to provide static variables within procedures, it was possible to encapsulate the implementation of simple data abstractions within the procedure. The scope rules of FORTRAN 77 ensured that elements declared internally within a procedure were inaccessible from a point external to that procedure. Each entry point corresponded to a single operation of the abstraction and exceptions were raised when a precondition was not satisfied on entry. Information about the object state was obtained via the procedure entry points, which implemented the operations of the specification and returned values of FORTRAN 77 built-in data types. By way of example, the implementation of a simple up/down counter is given in Figure 2.1. The counter is initialised (to zero) by calling the count routine. The parameter $M$ places an upper bound upon the counter value so that the counter may take any value between 0 and $M$. The entry point inc supports a increment operation on the counter with dec providing the inverse operation. Note that in both these cases a routine except is called if an attempt is made to violate the counter bounds (incrementing a counter which has a value $M$ or decrementing a counter which has a value 0). The operation val merely returns the current value of the counter and causes no change of state.
Figure 2.1 An implementation of a simple counter abstraction in standard FORTRAN 77

2.4 Data Structure Libraries

Writing specifications for data abstraction is a non-trivial exercise. A degree of specialist mathematical knowledge is required which is not normally included in the training of computer staff. This problem has led to the proposal that such a task be left to experts [Guttag, 1980] and that libraries of data structures should instead be supported [Miller, 1988]. Low [Low, 1978] presented a system for automatically choosing efficient representations for abstract data types from a library of implementations and a model was presented in [Rosenchein, 1977] to aid programmers in choosing efficient representations. Hoare [Hoare, 1972b] [Hoare, 1975] noted the high degree of commonality in the use of the concept of type by mathematicians, logicians and programmers. The types of interest to the programmer are those already familiar to the mathematician; namely Cartesian products, discriminated unions, sets, functions, sequences and recursive structures. Berztiss and Thatte [Berztiss, 1983] proposed a restraint upon the number of data abstractions in a similar manner to that placed upon control-flow constructs in structured programming [Dijkstra, 1965]. They outlined the selection principles for such a set of abstractions. From mathematics the set, sequence and binary relation were identified as being of fundamental importance. Four access mechanisms were also discussed, arbitrary ordering, access via a key field, ordered access specified by an iterator, and restricted access via a structure such as a stack or queue. The final selection criteria, data organisation, indicated the need for a heterogeneous data structure. Berztiss and Thatte applied empirical techniques to verify that the model was adequate but stipulated that it was not exhaustive. Their resulting classes of data structure were a set, a record, an array, a linear list and a tree.

A similar restriction upon the classes of data structure available was proposed by Mills and Linger [Mills, 1986], who introduced the concept of data structured programming. They described a disciplined access method for program data and proved that any result achieved using arrays and pointers could be repeated using instead sets, stacks and queues. Citing
work carried out at IBM Corporation in Bethesda, Mills and Linger suggested that programs designed using data structured programming will contain fewer variables and data references, are more obviously correct and have shorter verification proofs than those designed with arrays and pointers. Earlier a programming methodology based upon the removal of pointers was proposed by Kieburtz [Kieburtz, 1976].

Gries [Gries, 1981] proposed that an abstraction should be made so as to consider an array as a partial function from subscript values to domain values. The index of the array being the domain of the function, which maps onto values assigned to the elements of the array. Downey and Sethi [Downey, 1976] highlighted the problems associated with arrays by showing that testing the equivalence of straight line programs with arrays was NP-complete while straight line programs without arrays can be tested for equivalence in linear time.

Berry, Erlich and Lucena [Berry, 1976] discussed the problems associated with data representation via pointers and analysed the cases for and against the inclusion of the pointer type as a construct in high-level languages. They outlined a compromise based upon data abstraction. Dembinski and Schwartz [Dembinski, 1977] treated the pointer type in a way consistent with other types and applied formal program verification methods to programs using pointers. Pointers and arrays represent arbitrary access to data, which is clear from the length and complexity of any formal proof applied to programs containing these structures [Dijkstra, 1976] [Reynolds, 1979].

The complexity of data abstraction specification and implementation together with the unfamiliarity of its use in FORTRAN 77 systems suggests the introduction of a library of data structures. The set, stack, queue, binary relation and abstract array (partial function) compose the library herein although it is not exhaustive and additions may be made.

If the specification of a data abstraction is written in a language with precise semantics then an implementation may be proved correct with respect to the specification. In the following section a review of the formal specification techniques available is presented.

2.5 Formal Specification for Data Abstraction

The most fundamental desirable property of software is correctness – does it do what it is supposed to do. Techniques for establishing the correctness of a program may be classified as informal or formal. Informal techniques (such as debugging, testing and program reading) are inadequate as too great a reliance is placed upon human ingenuity and intuition. Formal techniques establish the correctness of a program by appealing to rules of inference, axioms and theorems, and do not suffer from the inadequacies of their informal counterparts.
A program begins as a concept, which has an infinite number of possible implementations [Liskov, 1975b]. A formal program development technique interposes a specification between the concept and the program. The specification provides a mathematical description of the concept and the correctness of an implementation may be established by proving that it satisfies the specification.

Formal specifications use mathematical notations to describe in a precise manner the properties of an information system, without constraining the way in which these properties are achieved [Spivey, 1989]. In addition, certain forms of inconsistency or incompleteness [Guttag, 1978a] [Berztiss, 1983] can be detected automatically and this unambiguous description of the system increases the rigour in program development [Liskov, 1979]. Support for the use of formal notation has been well documented [Liskov, 1975b] [Liskov, 1979] [Berztiss, 1983] [Meyer, 1985] [Liskov, 1986] [Cohen, 1982] [Cohen, 1989] and a survey of the computer support for formal notations was presented in [Lindsay, 1988].

Many formal specification techniques for data abstraction have been proposed and useful criteria for comparing them were outlined in [Liskov, 1975b]:

- **Formality** – specifications should be written in a notation which is mathematically sound. This allows proofs of correctness to be established, gives a machine understandable notation and allows specifications to be studied mathematically;

- **Constructability** – it must be possible to construct a specification without undue difficulty. Two facets of the process are of interest:
  - difficulty of constructing the specification
  - difficulty of knowing the specification captures the concept;

- **Comprehensibility** – a person trained in the notation being used should be able to read a specification and then, with a minimum of difficulty, reconstruct the concept which the specification is intended to describe;

- **Minimality** – the properties which are of interest must be described precisely and unambiguously but in a way which adds as little extraneous information as possible;

- **Wide range of applicability** – associated with each specification technique is a class of concepts which the technique can describe in a natural and straightforward fashion. The larger this class of concepts, the more useful the technique;
• Extensibility – it is desirable that a minimal change in a concept results in a similar small change in its specification.

In [Liskov, 1975b] a number of specification techniques were surveyed and evaluated with respect to the criteria. These techniques were the use of a fixed discipline such as graphs [Early, 1971], use of an arbitrary discipline such as that used by Hoare to specify sets and certain subsets of integers [Hoare, 1972a], a state machine model [Panas, 1972a], an axiomatic description [Hoare, 1969] and an algebraic definition [Guttag, 1978a].

The use of a fixed discipline relies upon an established mathematical discipline being used to provide an abstract model of the desired data abstraction. This approach satisfies many of the adequacy criteria. However, not all data abstractions can be specified with equal facility. For example, the graphical representation of a set data structure interposes an ordering upon the set elements resulting in over-specification. This violates the criterion of minimality and places practical limits upon the range of applicability. Allowing the specification to be expressed in terms of any arbitrary discipline reduces the unwanted representation details present in the use of a fixed discipline. However, in practice the number of disciplines which may be used must be sufficiently small otherwise a completely free choice could result in incomprehensible specifications. Again not all data abstraction can be specified in any of the chosen disciplines with equal facility and the criteria of minimality and range of applicability are violated.

A data abstraction can be modelled in terms of a state machine [Panas, 1972a] by partitioning the operations associated with the data structure into two groups: those which cause state changes (O-operations) and those which allow some aspect of the state to be observed without causing a change of state (V-operations). Certain O-operations have a delayed effect on V-operations. For example, for a stack data structure pushing an element onto a stack has a delayed effect upon the V-operation which returns the top element of the stack. The former top element being no longer directly observable until a pop operation has been applied. These delayed effects can be described by functions representing aspects of the state which are not immediately observable. However, functions of this nature add representational detail and detract from the minimality of the specification. In addition, there are problems with formal construction and proofs of correctness.

Axiomatic specification was first used by Hoare [Hoare, 1969] to define the built in data types of programming languages. It has since been the basis for the state-based descriptions such as those given in VDM [Jones, 1980], Z [Spivey, 1988a] [Spivey, 1988b] and the object-oriented set-theoretic technique of Schuman and Pitt [Schuman, 1987]. The desired properties of the data abstraction are modelled in terms of a well defined mathematical discipline to give axioms and rules of inference. In essence they are operational, depending upon an
underlying representation domain such as ZF set theory [Berziss, 1983]. For example, a VDM specification contains three components: a model of the state, invariants on the state and operations over the data abstractions comprising the state. The state definition describes the structure of the class of objects representing the state in terms of familiar basic types, complex types built from basic types and constructors such as sets, sequences, tuples and mappings. The invariants limit the objects within the class of objects representing the state to those that represent valid states. Invariants are thus constraints which must be preserved by operations. The operations are defined implicitly by preconditions and postconditions. The former is a predicate over the initial state and inputs and defines the conditions under which the operation produces valid results. The latter is a predicate over the initial state, inputs and final state which defines the effect of the operation. A state-based approach to specification has been advocated for specifying the data abstractions in Alphard [Shaw, 1981], CLU [Liskov, 1981] and Euclid [London, 1979] and expository introduction to state-based specification in Euclid was given in [Guttag, 1980].

Algebraic specification (value-based) [Guttag, 1977a] [Guttag, 1977b] [Goguen, 1978a] [Guttag, 1978a] [Guttag, 1978b] [Kamin, 1979] [Guttag, 1980] [Liskov, 1986] views a data abstraction as a collection of values and operations which may be described by an algebraic system. The language of heterogeneous algebras [Birkhoff, 1970] is particularly suited to specifying such systems as it allows the specification to be made using multiple domains (eg sets of integers act within two distinct and disjoint domains: sets of integers and integers). An algebraic specification is a pair consisting of a signature and a set of relations or axioms constraining semantics. The signature provides the abstract data type name and the names of the operations together with their domain and range sorts. The semantics are constrained by giving relationships between the operations. Operations may be partitioned into constructors and observations [Liskov, 1986]. Constructors have as their range sort the abstract data type being specified and the observations return other sorts. Systems designed to test such specifications include OBJ [Goguen, 1982], AFFIRM [Musser, 1980], CLEAR [Burstall, 1980a] and DAISTS [Gannon, 1981].

The algebraic approach is most appropriate for specifying values, where decomposition is based upon abstract data types, characterising the domain in terms of constructor and observation functions [Guttag, 1978a]. State-based methods involving preconditions and postconditions, expressed in terms of a suitable model, would appear to be most natural for specifying objects. In general, there are a greater number of proof obligations in a state-based specification but composition of large specifications from a number of smaller ones is straightforward. For an algebraic specification the number of proof obligations is smaller, but specification composition is difficult. Therefore, the choice of specification technique should be governed by whether data types or classes are required.
Chapter 2
Abstraction in Programming Languages

A major issue in formal specification is consistency [Guttag, 1978a]. Classically, a theory is consistent if and only if it is impossible to derive a contradiction as one of its consequences [Berztiss, 1983]. The most tractable method of proving the consistency of a specification is to demonstrate that there is a model for it. In addition, for algebraic specifications the issue of sufficient completeness arises [Guttag, 1978a]. An axiomatization is sufficiently complete if all properties of the type of interest are derivable from the axioms. In general there exists no finite procedure to determine if an axiomatization is sufficiently complete so it is necessary to determine instead sufficient conditions for sufficient completeness. That is, what conditions should be imposed on the axiomatization so as to guarantee sufficient completeness. Guttag and Horning [Guttag, 1978a] formulated a number of conditions which when imposed ensured sufficient completeness.

The formal specification of data abstractions which are to be implemented in FORTRAN 77 has received little consideration. Isner [Isner, 1982] proposed the use of the state machine model [Parnas, 1972a] for the specification of data abstraction in FORTRAN 77, supporting this with the statement that more formal techniques are less comprehensible. However, with the state machine technique it is not a straightforward task to build a simple description of the effect of the constructor operations, as outlined in [Liskov, 1975b], and proofs are difficult to construct. A more satisfactory alternative is a state-based approach wherein the data abstraction takes a form similar to that implemented by Isner [Isner, 1982]. The representation was hidden from the user-code within the procedure and only a reference was returned to distinguish between different instantiations of the same data class. Information about the object state was obtained via the procedure entry points, which implement the operations of the specification and returned values of standard FORTRAN 77 data types. Ford and Miller [Ford, 1985] adopted an algebraic approach, which was appropriate for the description of the new data types they introduced.

The data independence supported by formally specified abstractions allows parallel representations of data structures to be introduced, wherein the implementation of the parallel algorithm is hidden from the user-code. This leads to improvements in data query throughput without the need for the user's consideration of consistency within the data structure.

2.6 Concurrency and Data Abstraction

It has long been recognised that the potential speed-up of program execution is reaching its limit on single processor systems [Bernstein, 1966] [Amdahl, 1967]. Multiple processor systems offer a number of advantages over their single processor counterparts [Halsall, 1983] [Quinn, 1987], which include increases in program throughput, improved levels of system availability and ease of incremental growth.
Chapter 2  
Abstraction in Programming Languages

Data abstraction provides data independence: a change in the representation of the data abstraction has to affect the code that implements the associated operations, but all changes are localised to just that section of code. This allows a number of alternative implementations to be considered, without changes to the original specification or to any user-code. One such alternative is a parallel implementation, which allows a number of queries to the data structure to execute simultaneously whilst maintaining consistency.

Abstraction has already been applied to parallel systems [Bishop, 1986] resulting in structures such as monitors being supported by a number of parallel languages (Mesa [Mitchell, 1979], Pascal-Plus [Welsh, 1979] and Concurrent Euclid [Holt, 1982]). Unlike conditional critical regions [Brinch Hansen, 1981], where operations on a shared data structure are dispersed throughout the processes, monitors encapsulate the shared data structure and operations upon it and control all access to the structure.

Message passing systems move away from the shared data structure approach with processes cooperating by sending and receiving messages. Languages utilising this technique include Gypsy [Ambler, 1977], CHILL [CCITT, 1984], Argus [Liskov, 1987], the CSP notation [Hoare, 1985] and occam [May, 1983] (an implementation of CSP).

Several programming paradigms for concurrent computation have been proposed, which include process-oriented [Hoare, 1978] [Milner, 1980], functional and logic [Agerwala, 1982] [Shapiro, 1983] and object-oriented [Yonezawa, 1987] [Bershad, 1988]. Object-oriented computation provides a powerful means of controlling access to shared data [Kahn, 1981]. Abstraction principles may be applied to objects allowing the parallelism contained within an object to be hidden from the outer environment.

Several sources have noted the need for a migration route for sequential software to parallel hardware [Robinson, 1988] [Walden, 1988] [Hey, 1989], with one of the major areas of effort being the development of “parallel-extracting” compilers [Karp, 1988] [Robinson, 1988]. The use of abstraction coupled with parallel implementations of data structures has so far not been considered as a method of aiding this migration. In addition, abstraction provides numerous benefits in the development of new code for parallel systems.
CHAPTER 3 STATE-BASED SPECIFICATION

The object-oriented set-theoretic specification technique proposed by Schuman and Pitt [Schuman, 1987] is described in this chapter. Chapter 4 shows that it is an excellent choice for the specification of object classes in FORTRAN 77.

3.1 The Object-Oriented Set-Theoretic Specification Technique

A state-based specification technique describes the desired properties of the data class in terms of a well defined mathematical discipline to give axioms and rules of inference [Hoare, 1972a]. The object-oriented set-theoretic technique proposed by Schuman and Pitt [Schuman, 1987] [Schuman, 1989] is based upon ZF set theory. This approach originated from the work by Abrial [Abrial, 1982] which has subsequently been manifested as the Z Specification Language [Hayes, 1987] [Spivey, 1988a] [Spivey, 1988b]. The object-oriented paradigm has proven to be an effective technique for decomposing and reasoning about complex systems. This specification technique provides a counterpart to object-oriented design and implementation in terms of classes and objects. An abstract class (of objects), as opposed to an abstract type (of values), encapsulates the definition of an internal state and an associated set of access operations, which serve to query and/or update the state of such objects. The assumption is that the classes of abstract objects will be instantiated as independent subsystems in different contexts. The mechanisms for composition include multiple inheritance and refinement (by extension and/or restriction).

The specification of an object class is made up of a single state schema and an associated set of event schemas. The state schema identifies the class and names any formal parameters for the class, which behave as constants in the specification. Component declarations are given for the specification together with any invariant or initialisation predicates. The set of event schemas describe object state changes. Within the schema framework the specification is expressed in terms of predicate calculus and ZF set theory. A significant departure from other notations (namely VDM and Z) is made through a rule of historical inference which allows only the minimal effect (change-of-state) to be specified for each possible event (only changes to components are given in the postconditions of the events).

By way of example, the state schema for the class $fset$ (finite set) is given in Figure 3.1.
The state of objects belonging to the class \texttt{FSET} is described by the state components \( s \) and \( n \), where \( s \) is a finite subset of some given (carrier) set \( \mathcal{I} \) (items) and \( n \) is always equal to the cardinality of \( s \) (from the state invariant). The set \( s \) is initially empty.

Query operations may be introduced to interrogate the internal state of the objects, examples being an event to return the cardinality of the set, an empty set test and a membership test, shown in Figure 3.2.

Events to include a new item \( i \) in the set and exclude some \( i \) from the set are shown in Figure 3.3.

The events are characterised in terms of preconditions and postconditions, wherein the usual convention that dashed variables denote the corresponding component values after each occurrence of that event is adopted. The parameter names represent constant values so never appear in dashed form. The preconditions specify what must hold (with respect to the parameter and current state component values) in order for the operation to be applicable to an object of that class. The postcondition specifies the explicit effect of such an application in terms of the minimal properties that must afterwards hold. These same conventions are used as part of the state schema to specify initialisation, which may be considered to be a
pseudo-event that is implicitly invoked whenever a new object of the class is first instantiated.

The events \texttt{FSET.Include} and \texttt{FSET.Exclude} could have been specified more concisely as demonstrated in Figure 3.4. Note that textual sections of the schema framework in which nothing needs to be stated are simply omitted.

| \texttt{FSET.Include} \begin{array}{l} \text{\texttt{i}: \texttt{X}\setminus\texttt{S}} \\ \text{\texttt{i} \in \texttt{S'}} \end{array} | \texttt{FSET.Exclude} \begin{array}{l} \text{\texttt{i}: \texttt{S}} \\ \text{\texttt{i} \notin \texttt{S'}} \end{array} |

\textbf{Figure 3.4} The concise specification for \texttt{FSET.Include} and \texttt{FSET.Exclude}

If it is intended for the \texttt{FSET.Include} and \texttt{FSET.Exclude} operations to be always applicable, whether the argument \texttt{i} was or was not already a member of \texttt{s}, then additional definitions may be introduced, shown in Figure 3.5. This is termed overloading the event names. The absence of any postcondition in such schemas is taken to mean that although the event may indeed occur, provided its precondition is satisfied, the state of the object to which it is applied remains unchanged. Note that the preconditions of the two separate event schemas introduced for each operation are disjoint giving fully deterministic behaviour in this specification.

| \texttt{FSET.Include} \begin{array}{l} \text{\texttt{i}: \texttt{S}} \end{array} | \texttt{FSET.Exclude} \begin{array}{l} \text{\texttt{i}: \texttt{X}\setminus\texttt{S}} \end{array} |

\textbf{Figure 3.5} Additional definitions for \texttt{FSET.Include} and \texttt{FSET.Exclude}

There is no real need to resort to such specification by case-analysis in this example. Instead, the event schemas for each operation could simply be replaced by a single one having a weaker precondition, shown in Figure 3.6.

| \texttt{FSET.Include} \begin{array}{l} \text{\texttt{i}: \texttt{X}} \\ \text{\texttt{i} \in \texttt{S'}} \end{array} | \texttt{FSET.Exclude} \begin{array}{l} \text{\texttt{i}: \texttt{X}} \\ \text{\texttt{i} \notin \texttt{S'}} \end{array} |

\textbf{Figure 3.6} Weaker definitions for \texttt{FSET.Include} and \texttt{FSET.Exclude}

The object-oriented approach makes it plausible to speak about the “behaviour” of individual instances of a given class (how the state of such objects may evolve over time) [Schuman, 1989]. This is most easily visualised as a “decision tree”. For the class \texttt{FSET} (with \texttt{X=\texttt{NAT}}) a small part of the initial pattern of behaviour might be depicted as in Figure 3.7.
Branching within such a tree portrays choice, that is alternative events which may occur at distinct points in some possible "history" (a sequence of nodes along any path starting from the root), where time advances downwards in the tree. Branches are labelled by a "denotation" for one particular event, which in general reflects further choice amongst specific argument values admissible at that point. The "fan-out" from each node is to suggest other choices (of argument values or of different events that may also be applicable).

The nodes have been annotated with relations corresponding to the postconditions of the immediately preceding event; the root annotation derives from the initialisation relation given in the state schema. Within these annotations, component names are subscripted so as to distinguish different points in time. If the subscripts are ignored, then these and other relations which hold at successive nodes along a given history are to be interpreted as incremental (or cumulative) assertions about the state of an individual object up to that point; note that later assertions may then "over-ride" earlier ones.

Figure 3.7 The decision tree for the behaviour of $\text{FSET}$
Another way to characterise the behavioural properties that may be inferred from a specification is in terms of traces [Hoare, 1985]. A trace is a finite sequence of event denotations, which record a possible order of distinct occurrences over time. For example a set of initial traces for an object of the class \( \text{fset} \) (including initialisation, denoted by \( \text{fset}(()) \)), might be the following:

\[
\emptyset \\
\langle \text{fset}(), \text{include}(1) \rangle \\
.............
\langle \text{fset}(), \text{include}(1), \text{include}(2) \rangle \\
\langle \text{fset}(), \text{include}(1), \text{exclude}(1) \rangle
\]

For the present approach [Schuman, 1989], a history is a pair \( \langle \tau, P \rangle \) where \( \tau \) is a trace comprising a possible sequence of event occurrences for an object and \( P \) is a set of (undashed) predicates closed under logical inference: \( P \) embodies all properties which can be (syntactically) inferred about the state of an object after it has engaged in the particular sequence denoted by \( \tau \).

An alternative probably more familiar specification for the class \( \text{fset} \) is given in Figure 3.8.

\[
\begin{align*}
\text{FSET1} \\
\text{S: set}[x] \\
\text{n: NAT} \\
\text{n = #S} \\
\text{S' = S} \\
\end{align*}
\]

(Identical) state schema with the same two events:

\[
\begin{align*}
\text{FSET1. Include}(i) & \quad \text{FSET1. Exclude}(i) \\
i: X \setminus S & \quad i: S \\
S' = S \cup \{i\} & \quad S' = S \setminus \{i\}
\end{align*}
\]

Figure 3.8 An alternative specification for the class \( \text{fset} \)

The only apparent difference between \( \text{fset} \) and \( \text{fset1} \) is the way the postconditions are expressed (although \( \text{fset} \) and \( \text{fset1} \) are indeed different specifications).
The same part of the initial behaviour pattern for the class $\text{FSET1}$ is depicted in Figure 3.9.

The intention is that, in the absence of other (contextual) constraints, the classes $\text{FSET}$ and $\text{FSET1}$ should imply precisely the same behaviour. The primary purpose of the stronger postconditions, as specified for the class $\text{FSET1}$, is to carry forward historical values from the set $s$. This can also be achieved by implicitly augmenting the weaker postconditions, as specified for the class $\text{FSET}$, with suitable central relations, which serve to state that "the rest stays unchanged". In $\text{FSET}$, the appropriate central relation for both $\text{Include}(i)$ and $\text{Exclude}(i)$ is:

$$S' \setminus \{i\} = S \setminus \{i\}$$

Conjoining this to the explicit postconditions gives:

- for $\text{Include}(i)$: $i \in S' \land S' \setminus \{i\} = S \setminus \{i\}$
  $$\iff S' = S \cup \{i\}$$

- for $\text{Exclude}(i)$: $i \notin S' \land S' \setminus \{i\} = S \setminus \{i\}$
  $$\iff S' = S \setminus \{i\}$$

Figure 3.9 The decision tree for the behaviour of $\text{FSET1}$
Chapter 3  
State-Based Specification

This ensures that the same things are inferred after these events as can be inferred for $FSET_1$.

The preservation of the state invariant ($N = \# S$, in this case) is always implicit in the specification for $FSET$. Any postcondition whose sole purpose is to maintain (or restore) such invariants need not be stated explicitly. These particular interpretations are embodied in the validation conditions for a class specification as illustrated in Chapter 4, and in the special rules of “historical inference” over its implied behaviours [Schuman, 1989].

This approach has certain advantages in that:

- (real) specifications are easier to read and write;
- certain pitfalls of “over-specification” are avoided;
- more specifications can be concurrently composed (since the central relations are context-dependent).

To illustrate this latter point, consider the two different formulations for the Include event given in Figure 3.10.

A new event may be defined for “double inclusion” by concurrent composition, as shown in Figure 3.11.

Such composition is just the logical conjunction of the preconditions and postconditions of the events involved, as shown by the expansion given in Figure 3.12.
This example raises the issue of interference - since the two **Include** events of the class operate upon the same state component (namely \( s \)). Here a principle of avoiding over-specification, which is embodied within this technique, comes into its own. **FSET1.Include2** manifests inconsistencies in the conjoined postcondition (unless \( i = j \), which is excluded by the precondition or \( \{ i, j \} \subseteq s \), implying that the event has no effect in any case). No such inconsistencies appear when composing the weaker postconditions of **FSET**.

### 3.2 Refinement of a Specification

Much of the appeal of the object-oriented approach is due to its support for subsequent specialisation of any previously defined class of abstract objects, giving rise to one or more subclasses. The properties associated with the class are inherited by the subclass. The specification of a class may be refined by introducing a new subclass for which properties are specified relative to the base class. A subclass definition takes the form of a state schema identifying the base definition and containing an embedded subschema which gives the actual extensions. Operations associated with the new subclass may be defined in a similar fashion by promoting existing operations from the base class. This allows a specification to be refined and built-up in a hierarchical fashion. Complex specifications may then be simplified by using a step-wise approach.

Consider the restriction on **FSET** to specify the subclass **BSET** (bounded set), shown in Figure 3.13.
The bound is introduced by the parameter \( m \) through the additional invariant. Promoting the Include and Exclude operations requires a further precondition for the Include operation to ensure that the invariant is maintained, shown in Figure 3.14.

\[
\begin{align*}
\text{BSET.} &. \text{Include}(i)  & \text{BSET.} &. \text{Exclude}(i) \\
& \quad \text{FSET.} &. \text{Include}(i) & \quad \text{FSET.} &. \text{Exclude}(i) \\
& \quad (i \not\in S) \Rightarrow (n < m)
\end{align*}
\]

**Figure 3.14** The promoted Include and Exclude operations

The other queries may be promoted in a similar fashion, shown in Figure 3.15.

\[
\begin{align*}
\text{BSET.} &. \emptyset (\rightarrow b)  & \text{BSET.} &. \text{card} (\rightarrow k)  & \text{BSET.} &. \text{member}(i \rightarrow b) \\
& \quad \text{FSET.} &. \emptyset (\rightarrow b) & \quad \text{FSET.} &. \text{card} (\rightarrow k) & \quad \text{FSET.} &. \text{member}(i \rightarrow b)
\end{align*}
\]

**Figure 3.15** The promoted queries

Additional queries or events may be introduced as required in the context of such refinement. For example an event \( \text{AnyMemb} \), which involves arbitrary selection, is shown in Figure 3.16.

\[
\begin{align*}
\text{BSET.} &. \text{AnyMemb}(\rightarrow i)  \\
& \quad \text{FSET.} &. \text{Exclude}(i) \\
& \quad i \in S
\end{align*}
\]

**Figure 3.16** The AnyMemb event
The expanded specification for the class `bset` is given in Figure 3.17.

<table>
<thead>
<tr>
<th>BSET (m)</th>
<th>BSET. Include (i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>m, n: NAT</td>
<td>i: S</td>
</tr>
<tr>
<td>S: set[X]</td>
<td>n: #S</td>
</tr>
<tr>
<td></td>
<td>n &lt; m</td>
</tr>
<tr>
<td></td>
<td>i ∈ S'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BSET. empty (→ b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b: BOOL</td>
</tr>
<tr>
<td>b = (S = ∅)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BSET. card (→ k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>k: (n)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BSET. member (i → b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i: X</td>
</tr>
<tr>
<td>b: BOOL</td>
</tr>
<tr>
<td>b = (i ∈ S)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BSET. Include (i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i: S</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BSET. Exclude (i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i: X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BSET. AnyMemb (→ i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i: S</td>
</tr>
</tbody>
</table>

**Figure 3.17** The expanded specification of the `bset` class
CHAPTER 4 SPECIFICATION AND IMPLEMENTATION OF THE OBJECT CLASSES

The validation of the bounded set formal specification is presented in this chapter and the proofs associated with the correctness of usage and implementation of the data structure are described. Two different implementations are considered and a refinement is made to introduce multiple instances.

4.1 Specification and Validation

The formal specification of a bounded set of items drawn from some given set \( \mathbb{I} \) (items) as developed in the previous chapter, is shown in Figure 4.1.

The formal parameter \( m \) places an upper bound upon the cardinality of the set through the invariant predicate \( n = \#S \land n \leq m \). The component \( S \) is the internal representation of the set, and \( \mathbb{I} \) may be any built-in FORTRAN 77 data type. The event \texttt{Include} adds an element \( i \) to the set and is overloaded to take account of the two cases, when \( i \) is or is not already present in the set before the event. If \( i \) is not already a member of the set, the precondition

\[
BS_{\text{E}T}(m) \quad BSET.\text{Include}(i) \\
\begin{align*}
m, n: \text{NAT} \\
S: \text{set}[\mathbb{I}] \\
n = \#S \\
n \leq m \\
S' = \emptyset \\
i: \mathbb{I} \setminus S \\
n < m \\
i \in S' \\
BS_{\text{E}T}.\text{empty}(\rightarrow b) \quad BSET.\text{Exclude}(i) \\
b: \text{BOOL} \\
b = (S = \emptyset) \\
BS_{\text{E}T}.\text{card}(\rightarrow k) \quad BSET.\text{AnyMemb}(\rightarrow i) \\
k: \{n\} \\
i: \mathbb{I} \\
i \notin S' \\
i \in S \\
BS_{\text{E}T}.\text{member}(i \rightarrow b) \\
b: \text{BOOL} \\
b = (i \in S) \\
\end{align*}
\]

Figure 4.1 The specification of the bounded set

The formal parameter \( m \) places an upper bound upon the cardinality of the set through the invariant predicate \( n = \#S \land n \leq m \). The component \( S \) is the internal representation of the set, and \( \mathbb{I} \) may be any built-in FORTRAN 77 data type. The event \texttt{Include} adds an element \( i \) to the set and is overloaded to take account of the two cases, when \( i \) is or is not already present in the set before the event. If \( i \) is not already a member of the set, the precondition
n < m ensures that n never exceeds m, thus violating the invariant (after an application of the event). The event **exclude** removes an element i from the set; if i is not a member of the set then no change is made. The event **anyMember** returns a member of the set and removes the returned item from the set. The query **empty** returns a **bool** indicating whether the set is empty. **bset.card** returns the cardinality of the set and **bset.member** tests whether an element i is a member of the set.

The proof that a specification is **valid** [Schuman, 1989] begins by establishing that the subsystem state (characterised by the state schema) is **consistent**. That is, there is some state which satisfies the declarations and the invariants of the class. This is usually achieved by showing the existence of a (trivial) model wherein the overall state invariant derived from these constraints holds. For the class **bset**, it is necessary to show that:

\[ \exists m, n: \text{NAT}; S: \text{set}[\text{I}]; \] 
\[ n = \#S \land n < m \]

The proof is discharged, for example, by choosing m = 0, n = 0 and S = 0.

There are two separate aspects for proving that the queries and events associated with a class are consistent. Mutual constraints between event arguments and state components are again specified as simple predicates which are undashed. The effect of an event is a relation ranging over these arguments and both dashed as well as undashed state variables. By simple conjunction, composite predicates for the derived precondition and postcondition are obtained which include the state invariant. The inclusion of the dashed state invariant in the derived postcondition avoids the need to write explicit postconditions solely for the purpose of preserving the invariants. The two validation conditions for each query and event arise from these constructions. Each must be shown to be **applicable**. That is, there is a state which satisfies the derived precondition of the query or event. In the case of the event **bset.Include**, analysis is conducted for each schema separately, so that it is necessary to show:

**Case 1**: \[ \exists m, n: \text{NAT}; S: \text{set}[\text{I}]; i: \text{I} \rightarrow (i \notin S \land n = \#S \land n < m) \]

**Case 2**: \[ \exists m, n: \text{NAT}; S: \text{set}[\text{I}]; i: \text{I} \rightarrow (i \notin S \land n = \#S \land n \leq m) \]

Since the preconditions of these two cases are disjoint the event is completely deterministic. In addition, each query and event must be shown to be **effective**. That is, for any state and parameter values which satisfy the derived precondition of the event, there exists a possible new state which satisfies the derived postcondition of the query or event. For queries this proof is trivial since the derived postcondition is simply true. In the case of **bset.Include**, it is necessary to show that:
Chapter 4

Specification and Implementation of the Object Classes

Case 1: \( \forall m, n : \mathbb{N}; S : \text{set}[\mathbb{I}]; i : \mathbb{I} \cdot (i \notin S \land n = \#S \land n < m) \Rightarrow (\exists n' : \mathbb{N}; S' : \text{set}[\mathbb{I}] \cdot n' = \#S' \land n' \leq m \land i \in S') \)

Case 2: \( \forall m, n : \mathbb{N}; S : \text{set}[\mathbb{I}]; i : \mathbb{I} \cdot (i \in S \land n = \#S \land n \leq m) \Rightarrow (\exists n' : \mathbb{N}; S' : \text{set}[\mathbb{I}] \cdot n' = \#S' \land n' \leq m \land i \in S') \)

A further proof obligation is to show that it is always possible to establish a consistent initial state for instantiations of the class (effectiveness of the initialisation). For the class \text{BSET}, it is necessary to show that:

\( \forall m : \mathbb{N} \cdot (\exists n' : \mathbb{N}; S' : \text{set}[\mathbb{I}] \cdot n' = \#S' \land n' \leq m \land S' = \emptyset) \)

In summary, the necessary proof obligations required for showing that a specification is valid are:

- For the state schema:
  - showing that the subsystem class is consistent
  - showing that the initialisation is always effective;

- For each event schema:
  - showing the applicability of the event
  - showing the effectiveness of the event.

The complete proof that the specification is valid [Schuman, 1989] is given in Appendix B.

4.2 Correctness of Usages

The proofs concerning the usage of an object class rely solely upon the specification of the class and are independent of the implementation of that class. By way of example, consider the code segment given in Figure 4.2.

```
CALL EMPTY(B)
WHILE (.NOT.B) DO
  CALL ANYMEM(I)
  CALL EVAL(I)
  CALL EMPTY(B)
END WHILE
```

Figure 4.2 A simple usage of the set abstraction

The subroutine \text{EVAL} is called for every member of the set, and the item is removed from the set in the process. To prove that the usage of the set is correct in this case it is necessary to shown that:

\( \neg B \Rightarrow \text{pre[BSET.AnyMemb]} \)
Chapter 4 Specification and Implementation of the Object Classes

where \( \text{pre}[\text{C.E}] \) is the derived precondition of C.E. This is demonstrated trivially from the specification since:

\[
\text{pre}[\text{BSET}.\text{empty}] = (B = (S=\emptyset))
\]

therefore \( \neg B \Rightarrow (\exists i : I \cdot i \in S) \Rightarrow \text{pre}[\text{BSET}.\text{AnyMemb}] \)

This separation of proof concerns aids in modularising the reasoning involved in generating proofs of program correctness.

4.3 Correctness of an Implementation

The FORTRAN 77 multiple entry subroutine facility coupled with the \texttt{SAVE} statement to provide static variables encapsulates the implementation of simple data abstractions within a subroutine. The main subroutine body initialises the object class and each event is supported by an entry point into the subroutine.

The program listing for the implementation in FORTRAN 77 of a set object for base type \texttt{integer} is given in Figures 4.3a and 4.3b alongside the specification of the class. The subroutine \texttt{EXCEPT} handles the undefined states of the object. This routine should be supplied as an external subroutine.

![Figure 4.3a The implementation of the object class \texttt{BSET} in FORTRAN 77 (state and query schemas)](image-url)
The set object is initialised via a call to the `bset` subroutine. The integer array `a` holds the contents of the set and the variable `n` holds the index at which an element was last inserted into `a`. Therefore, the set consists of those elements of the array in the range `a(1)` to `a(n)`. Note that the parameter `m` is the upper bound for the index of array `a`. Each of the event schemas is implemented via an `ENTRY` statement into the subroutine `bset`.

The major drawback with the use of a static memory allocation technique is the constant `m`, which is introduced into the implementation of the object class. This corresponds to the maximum number of elements in the set and may only be changed at compile time.

```fortran77
ENTRY INCLUDE(I)
J=0
BF=.FALSE.
WHILE (J.LT.N).AND.(.NOT.BF)) DO
  J=J+1
  BF=A(J).EQ.I
END WHILE
IF (.NOT.BF) THEN
  IF (N.GE.M) THEN
    CALL EXCEPT
    RETURN
  END IF
  N=N+1
  A(N)=I
END IF
RETURN

ENTRY EXCLUDE(I)
J=0
BF=.FALSE.
WHILE (J.LT.N).AND.(.NOT.BF)) DO
  J=J+1
  BF=A(J).EQ.I
END WHILE
IF (BF) THEN
  A(J)=A(N)
  N=N-1
END IF
RETURN

ENTRY ANYMEM(I)
IF (N.EQ.0) THEN
  CALL EXCEPT
  RETURN
END IF
I=A(N)
N=N-1
RETURN

ENTRY EXCLUD(I)
J=0
BF=.FALSE.
WHILE (J.LT.N).AND.(.NOT.BF)) DO
  J=J+1
  BF=A(J).EQ.I
END WHILE
IF (BF) THEN
  A(J)=A(N)
  N=N-1
END IF
RETURN
```

**Figure 4.3b** The implementation of the object class `bset` in FORTRAN 77 (event schemas)

A valid formal specification provides a description against which an implementation of the specification may be proved correct. The implementation of a data abstraction defines how objects belonging to the class are represented. The relationship between the representation...
object and abstract object is classically defined by means of an abstraction function (representation object->abstract object) [Hoare, 1972a].

For the set data structure implemented by means of an array \( A \) and an index \( n \), the abstraction function is defined as:

\[
AXAXN \rightarrow SXmXN \cdot (S = \{A(i) \mid 1 \leq i \leq N\}) \land (m = M) \land (n = N)
\]

In addition, it is necessary to introduce a representation invariant – a statement of properties which all legitimate representation objects satisfy. This invariant captures the relationship between various data components used to represent the state of an object; it is needed in order to be able to reason about the implementation of each operation separately. The representation invariant \( (R_1) \) in this case is:

\[
(\forall i, j : 1..N \cdot i \neq j \Rightarrow A(i) \neq A(j))
\land (0 \leq N) \land (N \leq M)
\]

The verification involves showing:

- For the implementation of the state schema – providing the preconditions upon the parameter values are satisfied on entry, the initialisation predicate and representation invariant are satisfied on exit;
- For the implementation of the events – providing the derived preconditions and representation invariant are satisfied on entry, the derived postconditions and representation invariant are satisfied on exit.

The proof that the implementation, shown in Figure 4.3, satisfies the specification is given in Appendix C. In Appendix D the specifications and implementations of the stack, queue, abstract array and binary relation data structures are given.

4.4 An Alternative Representation

In Figure 4.3 the set data structure is represented as an unordered list of items (integers in this case) stored in the array \( A \). An alternative is to store the set items in order (assuming that the set \( S \) is totally ordered) and this representation is shown in Figures 4.4a and 4.4b.

The abstraction function for both representations is identical, however the representation invariant for the ordered case is:

\[
(\forall i, j : 1..N \cdot i < j \Rightarrow A(i) < A(j))
\land (1 \leq N) \land (N \leq M)
\]
Searching an unordered array during the Include, Exclude and member operations proceeds by the simple linear search where searching an n-entry list takes O(n) time. For an ordered sequence of items the search time for an n-entry list is O(log₂(n)). However in this case, the shifting of portions of the list which is required during the Include and Exclude operations results in execution times of O(n). In general, an unordered array is a superior representation when many more Include and Exclude operations than member operations are required while the ordered sequence performs best where more member operations are required.

**Figure 4.4a** The implementation of the object class `bset` as an ordered list (state and query schemas)
Figure 4.4b The implementation of the object class \texttt{BSET} as an ordered list (event schemas)
4.5 Multiple Instances and Refinement

The foregoing implementations allow for only one instance of the \texttt{bset} object. In many cases several instances of a class are required.

A new class \texttt{xset} (index set) is now specified and implemented, shown in Figure 4.5, which is a refinement of the class \texttt{fset}. \texttt{xset} generates distinct index values, which are drawn from the finite set \((1..p)\).

```
SUBROUTINE XSET
IMPLICIT LOGICAL (B)
IMPLICIT INTEGER (X)
SAVE
PARAMETER (P=100)
LOGICAL INDEX(1:P)
LOGICAL BF
DO 10,K=1,P
  INDEX(K)=.FALSE.
10 CONTINUE
RETURN
ENTRY VALID(X,B)
IF ((X.GT.P).OR.(X.LT.1)) THEN
  CALL EXCEPT
  RETURN
END IF
B=INDEX(X)
RETURN
ENTRY NEW(X)
K=0
BF=.TRUE.
WHILE ((K.LT.P).AND.BF) DO
  K=K+1
  BF=INDEX(K)
END WHILE
IF (BF) THEN
  CALL EXCEPT
  RETURN
END IF
X=K
INDEX(K)=.TRUE.
RETURN
ENTRY NULL(X)
IF ((X.GT.P).OR.(X.LT.1).OR.
    (.NOT.INDEX(X))) THEN
  CALL EXCEPT
  RETURN
END IF
INDEX(X)=.FALSE.
RETURN
```

\texttt{XSET} is implemented as a vector \((\text{INDEX})\) where every index value of the vector corresponds to an item in the universal set \((1..p)\). If item \(x\) is or is not a member of the
set represented by the vector then \text{INDEX}(x)\) is assigned true or false respectively. This is similar to the representation of a set as a bit vector [Hoare, 1972a].

Multiple instances of the bounded set can now be introduced by the subclass \texttt{MSET}, shown in Figure 4.6, which is a refinement of the \texttt{XSET} class. \texttt{BSET} objects are instantiated by the event \texttt{MSET.NewSet}. The parameter \(m\) of \texttt{BSET} and a unique index \(x\) are quantified by this event. An instance of \texttt{BSET} is removed by the \texttt{MSET.NullSet} event. The \texttt{XSET.valid} query is promoted together with all the events associated with the \texttt{BSET} class.

![Figure 4.6 The specification for multiple instances of sets](image)

The preconditions of a refined event are simply the conjunction of the preconditions of all the inherited events together with the precondition of the event itself. The postconditions are formed by a similar conjunction.

The program listing for the implementation of multiple instances of the set data structure for base type integer in standard FORTRAN 77 is given in Figures 4.7a, 4.7b and 4.7c. Before any set object may be created the class must be initialised via a call to the \texttt{MSET} subroutine. The entry point \texttt{NEWSET} in subroutine \texttt{MSET} takes as parameters two integers \(m\) and \(i\) and implements the \texttt{MSET.NewSet} event schema. A unique integer, \(i\), is returned, which indexes the new set. The integer array \(a\) holds the contents of all the defined sets and segments of this array are allocated to new set objects. The parameter \(m\) is the maximum number of set instances and corresponds to \(p\) in the specification. The array \(n\) holds the next index at which an element should be inserted into \(a\) and each set has an entry in the array \(\text{START}\) to indicate its starting index within \(a\), and an entry in \(\text{MAX}\) giving the maximum
size of the set. Calling \texttt{NEW} results in an area in the array \texttt{A} being reserved and if there is insufficient room available the \texttt{EXCEPT} routine is called. The entry point \texttt{NULSET} in subroutine \texttt{MSET} takes an \texttt{ID} and removes all space allocated to the set object referenced by it. The values stored in \texttt{A} are shifted when a set is removed so that the free space in \texttt{A} is a single segment. The logical array \texttt{REF} signifies whether an \texttt{ID} value is currently referencing a set object. The operations \texttt{INCLUDE}, \texttt{EXCLUDE}, \texttt{ANYMEM}, \texttt{EMPTY}, \texttt{CARD} and \texttt{MEMBER} are implemented in an identical fashion to that given in Figure 4.3. Far more efficient implementations of multiple instances are possible (for example, use of a "Buddy algorithm" [Knowlton, 1965] [Knowlton, 1966]).

Unfortunately, the identification of set objects by integer references or tags permitted the manipulation of the tags externally to the abstraction operations. The procedure creating an object returns a reference to the object rather than a representation of the object itself. This has consequences for the semantics of object reference. If there exists two integer variables, \texttt{i1} and \texttt{i2}, which represent references for two objects, \texttt{inst[1]} and \texttt{inst[2]}, then an assignment \texttt{i1=i2} causes \texttt{i1} to refer to the same object as \texttt{i2}, no changes are made to either \texttt{inst[1]} or \texttt{inst[2]}. Aliasing of this nature is unavoidable in \texttt{FORTRAN 77} without the introduction of language extensions and causes difficulties in program verification [Schwartz, 1979].

Fabry’s capability-based addressing technique [Fabry, 1974] was identified by Isner [Isner, 1982] as a more satisfactory alternative for identifying objects, but required extensions to the \texttt{FORTRAN 77} language. In this scheme a unique value becomes associated with an object when it is created, this value is termed a capability. To reference an object the capability must be supplied and extensions to the language are required to prevent these from being altered. The “limited private” types found in \texttt{Ada} [Ada, 1983] provide a similar facility but with compile-time enforcement (as opposed to run-time enforcement).

The implementation of a class may be derived directly from its specification through a process of refinement followed by extraction, which is described in the next chapter. This is an alternative to the classical approach of specification followed by implementation and verification.
SUBROUTINE MSET
IMPLICIT LOGICAL ( B )
SAVE
PARAMETER ( MR=100 )
PARAMETER ( MX=1000 )
LOGICAL INDEX(1:MR)
INTEGER M(1:MR)
INTEGER START(1:MR)
INTEGER N(1:MR)
INTEGER A(1:MX)
INTEGER POS
LOGICAL BF

DO 10, K=1, MR
   INDEX(K)=.FALSE.
10 CONTINUE
RETURN
ENTRY VALID(ID, B)
IF ((ID.GT.MR).OR.(ID.LT.1)) THEN
   CALL EXCEPT
   RETURN
END IF
B=INDEX(ID)
RETURN
ENTRY EMPTY(ID, B)
IF ((ID.GT.MR).OR.(ID.LT.1).OR.(.NOT.INDEX(ID))) THEN
   CALL EXCEPT
   RETURN
END IF
B=N(ID).EQ.START(ID)
RETURN
ENTRY CARD(ID, K)
IF ((ID.GT.MR).OR.(ID.LT.1).OR.(.NOT.INDEX(ID))) THEN
   CALL EXCEPT
   RETURN
END IF
K=N(ID)-START(ID)
RETURN
ENTRY MEMBER(ID, I, B)
IF ((ID.GT.MR).OR.(ID.LT.1).OR.(.NOT.INDEX(ID))) THEN
   CALL EXCEPT
   RETURN
END IF
J=START(ID)
B=.FALSE.
WHILE ((J.LT.N(ID)).AND.(.NOT.B)) DO
   B=A(J).EQ.I
   J=J+1
END WHILE
RETURN

Figure 4.7a The implementation of the multiple sets in FORTRAN 77 (state and query schemas)
Chapter 4  Specification and Implementation of the Object Classes

ENTRY NEWSET(ID, MV)
IF ((MV.LT.0).OR. (MV.GT. (MX-POS+1))) THEN
   CALL EXCEPT
   RETURN
END IF
K=0
BF=.TRUE.
WHILE ((K.LT.M) .AND. BF) DO
   K=K+1
   BF=INDEX(K)
END WHILE
IF (.NOT.BF) THEN
   CALL EXCEPT
   RETURN
END IF
ID=K
INDEX(K)=.TRUE.
START(ID)=POS
M(ID)=MV
N(ID)=POS
POS=POS+MV
RETURN

ENTRY NULLSET(ID)
IF ((ID.GT.MR).OR. (ID.LT.1).OR. (.NOT.INDEX(ID))) THEN
   CALL EXCEPT
   RETURN
END IF
INDEX(ID)=.FALSE.
DO 20, K=1, MR
   IF (INDEX(K).AND. (START(K).GT.START(ID))) THEN
      START(K)=START(K)-M(ID)
      N(K)=N(K)-M(ID)
   END IF
20 CONTINUE
POS=POS-M(ID)
DO 30, K=(START(ID)+N(ID)), (POS-1)
   A(K-N(ID))=A(K)
30 CONTINUE
RETURN

Figure 4.7b The implementation of the multiple sets in FORTRAN 77 (NewSet and NullSet event schemas)
ENTRY INCLUD(ID, I)
IF ((ID.GT.MR).OR.(ID.LT.1).OR.
(.NOT.INDEX(ID))) THEN
CALL EXCEPT
RETURN
END IF
J=START(ID)
BF=.FALSE.
WHILE ((J.LT.N(ID)).AND.(.NOT.BF)) DO
BF=I.EQ.A(J)
J=J+1
END WHILE
IF (.NOT.BF) THEN
IF ((N(ID)-START(ID)).GT.M(ID)) THEN
CALL EXCEPT
RETURN
END IF
A(N(ID))=I
N(ID)=N(ID)+1
END IF
RETURN
ENTRY EXCLUD(ID, I)
IF ((ID.GT.MR).OR.(ID.LT.1).OR.
(.NOT.INDEX(ID))) THEN
CALL EXCEPT
RETURN
END IF
J=START(ID)
BF=.FALSE.
WHILE ((J.LT.N(ID)).AND.(.NOT.BF)) DO
BF=I.EQ.A(J)
J=J+1
END WHILE
IF (BF) THEN
A(J)=A(N(ID)-1)
N(ID)=N(ID)-1
END IF
RETURN
ENTRY ANYMEM(ID, I)
IF ((ID.GT.MR).OR.(ID.LT.1).OR.
(.NOT.INDEX(ID))) THEN
CALL EXCEPT
RETURN
END IF
IF (N(ID).EQ.START(ID)) THEN
CALL EXCEPT
RETURN
END IF
I=A(N(ID)-1)
N(ID)=N(ID)-1
RETURN
END

Figure 4.7c The implementation of the multiple sets in FORTRAN 77 (Include, Exclude and AnyMemb event schemas)
CHAPTER 5  FORMAL DATA REFINEMENT

The object-oriented set-theoretic specification technique [Schuman, 1987] [Schuman, 1989] permits the implementation of a class to be derived through refinement [Pitt, 1990]. Refinement towards an implementation of a class involves adding new components of a state to a new class to include the components which form the basis of the implementation together with others which may be required to link the specification components with those of the implementation. Extra relations are added to the invariant, initialisation condition, precondition and postcondition as necessary. Without loss of generality all the relations associated to the actual implementation are included. This route is depicted in Figure 5.1 for the implementation of the BSET object class. A refinement is made to give the ASET object class. Components and relations are then extracted from this class to give the class CSET, an implementation of the original class BSET.

![Figure 5.1 Route from specification to implementation](image)

The process of refinement followed by extraction leading to an implementation of the BSET object class is presented in this chapter as an alternative to the classical approach of specification followed by implementation and verification which was given in Chapter 4 and Appendix C.

5.1 Class Behaviour and History

Given a class C with events C.Ei, the set of $\square$-histories (necessary histories) is the set of pairs $(\tau, P)$ where $\tau$ is a sequence of events drawn from:

$$\{ Ei \mid C.Ei \text{ is in the class C} \}$$

defined inductively as follows:
Chapter 5 Formal Data Refinement

\[ \langle 0, P \rangle \text{ is a } ◇\text{-history if } P = \{ p \mid \text{inv}[C] \land \text{init}[C] \Rightarrow p' \} \]

if \( \langle \tau, P \rangle \text{ is a } ◇\text{-history and } \text{pre}(Ei) \in P \)
then \( \langle \tau^{\Diamond}(Ei), Q \rangle \) is a ◇-history
where \( Q = \{ q \mid \exists p \in P \cdot (p \land \text{post}[C,Ei] \land \text{inv'}[C]) \Rightarrow q' \} \)

where inv[C] is the invariant of C.Ei, pre[C.Ei] is the derived precondition of C.Ei, post[C.Ei] is the derived postcondition of C.Ei and init[C] is a predicate characterising the effect of initialisation for an instantiation of C.

If \( \langle \tau, P \rangle \text{ is a } ◇\text{-history then } \tau \text{ is said to be a } ◇\text{-behaviour. A } ◇\text{-history is simply a } ◇\text{-behaviour paired with the set of all inferences which could be made about the state after that behaviour. Similarly a } ◇\text{-behaviour (possible behaviour) is a sequence of events each of which may be possible after its predecessors, and a } ◇\text{-history is simply a } ◇\text{-behaviour paired with the set of all inferences which could be made about the state after that behaviour. The set of } ◇\text{-histories is defined inductively as:} \]

\[ \langle 0, P \rangle \text{ is a } ◇\text{-history if and only if it is a } ◇\text{-history} \]

if \( \langle \tau, P \rangle \text{ is a } ◇\text{-history and } \neg \text{pre}[C,Ei] \in P \)
then \( \langle \tau^{\Diamond}(Ei), Q \rangle \) is a ◇-history
where \( Q = \{ q \mid \exists p \in P \cdot (p \land \text{post}[C,Ei] \land \text{inv'}[C']) \Rightarrow q' \} \)

If \( \langle \tau, P \rangle \text{ is a } ◇\text{-history then } \tau \text{ is said to be a } ◇\text{-behaviour.} \]

5.2 Refinement Towards Implementation

A class \( C_1 \) is said to implement a class \( C_0 \) if:

- every ◇-behaviour of \( C_0 \) is a ◇-behaviour of \( C_1 \);
- every ◇-behaviour of \( C_1 \) is a ◇-behaviour of \( C_0 \).

A refinement of a class \( C_0 \) to give a class \( C_1 \) is termed precondition-conservative if whenever the \( C_0 \) components of a valid state satisfy the precondition of \( C_0.E \) then the whole state satisfies the precondition of \( C_1.E \). Therefore the refinement does not change the applicability of any particular event. The refinement \( C_1 \) of \( C_0 \) is precondition conservative if and only if, for each event \( Ei \):

\[ (\text{pre}[C_0.Ei] \land \text{inv}[C_1] \Rightarrow \text{pre}[C_1.Ei]) \]

If \( C_1 \) is a precondition-conservative refinement of \( C_0 \) then \( C_1 \) implements \( C_0 \). By way of example the \texttt{BSET} object class, shown in Figures 5.2a and 5.2b, is refined to give the \texttt{ASET} object class, also shown in Figure 5.2.
Figure 5.2a The **BSET** and **ASET** object classes (state and query schemas)

Prior to components being added to C₀ to form C₁ a suitable representation of the set object class is chosen. In this case a set of n elements is represented by the first n elements of an array. Two components are therefore required, the index pointer n and the array (or partial function) A.
Assuming that the class \texttt{BSET} has already been shown to be valid, then if \texttt{ASET} is an implementation of \texttt{BSET} it is now necessary to prove that:

- \texttt{ASET} is a valid specification in its own right;
- \texttt{BSET} $\rightarrow \texttt{ASET}$ is a precondition-conservative refinement.

To show that \texttt{ASET} is valid it is necessary to prove that:

(1) The class is consistent
   There is a state which satisfies the state invariant;

(2) The initialisation is effective
   For any component values which satisfy the state invariant, there exists a possible initialisation state;
(3) Each event is effective
For any state and parameter values which satisfy the derived precondition of the
event, there exists a possible new state which satisfies the derived postcondition
of the event.

Note that in this case it is not necessary to prove the applicability of each event as the proof
to demonstrate that $\text{bset} \rightarrow \text{aset}$ is precondition-conservative shows that the refinement
does not change the applicability of any event. The remaining proofs proceed in the style
given in Appendix B.

By way of example, the proof that the Include event is precondition-conservative follows:

**Claim:**
$(\text{pre[BSET.Include]} \wedge \text{inv[ASET]}) \Rightarrow \text{pre[ASET.Include]}$

where
\[
\text{pre[BSET.Include]} = ((i \notin S) \wedge (n < m)) \lor (i \in S)
\]
\[
\text{inv[ASET]} = (n \neq \#S) \wedge (n \leq m) \wedge (\text{dom A}=1..m) \wedge (S=A[1..n])
\]
\[
\text{pre[ASET.Include]} = ((i \notin S) \wedge (n < m) \wedge (\forall j:1..n \cdot A(j) \neq i)) \lor
((i \in S) \wedge (\exists j:1..n \cdot A(j) = i))
\]
(the $\lor$ clauses are introduced because of the two cases for the Include event)

**Proof:**
\[
\text{pre[BSET.Include]} \wedge \text{inv[ASET]} = (((i \notin S) \wedge (n < m)) \lor (i \in S))
\]
\[
\wedge ((n \neq \#S) \wedge (n \leq m) \wedge (\text{dom A}=1..m) \wedge (S=A[1..n]))
\]
so
\[
\text{pre[BSET.Include]} \wedge \text{inv[ASET]} = ((i \notin S) \wedge (n < m) \wedge (n \neq \#S) \wedge (\text{dom A}=1..m) \wedge
(S=A[1..n]))
\]
\[
((i \in S) \wedge (\#S = n) \wedge (n \leq m) \wedge (\text{dom A}=1..m) \wedge (S=A[1..n]))
\]

**case analysis**
(consider each schema separately)

**case 1:** $(i \notin S) \wedge (n < m) \wedge (n \neq \#S) \wedge (\text{dom A}=1..m) \wedge (S=A[1..n])$

since $(i \notin S) \wedge (S=A[1..n])) \Rightarrow (i \in A[1..n]) \Rightarrow (\forall j:1..n \cdot A(j) \neq i)$

thus
\[
((i \notin S) \wedge (n < m) \wedge (n \neq \#S) \wedge (\text{dom A}=1..m) \wedge (S=A[1..n])) \Rightarrow
\]
\[
((i \in S) \wedge (n < m) \wedge (\forall j:1..n \cdot A(j) \neq i))
\]
case 2: (i ∈ S) ∧ (n ≠ S) ∧ (n ≤ m) ∧ (dom A = 1..m) ∧ (S = A\{1..n\})

since ((i ∈ S) ∧ (S = A\{1..n\})) ⇒ (i ∈ A\{1..n\}) ⇒ (∃j:1..n • A(j) = i)

thus
((i ∈ S) ∧ (n ≠ S) ∧ (n ≤ m) ∧ (dom A = 1..m) ∧ (S = A\{1..n\})) ⇒ ((i ∈ S) ∧ (∃j:1..n • A(j) = i))

combining these cases gives
(pre[BSE T. Include] ∧ inv[ASET]) ⇒ (((((i ∈ S) ∧ (n < m) ∧ (n ≠ S) ∧ (dom A = 1..m) ∧ (S = A\{1..n\})) ⇒ ((i ∈ S) ∧ (n < m) ∧ (∀j:1..n • A(j) ≠ i)) \lor
(((i ∈ S) ∧ (n ≠ S) ∧ (n ≤ m) ∧ (dom A = 1..m) ∧ (S = A\{1..n\})) ⇒ ((i ∈ S) ∧ (∃j:1..n • A(j) = i))))

thus
(pre[BSE T. Include] ∧ inv[ASET]) ⇒ pre[ASET. Include]

5.3 Extraction Towards Implementation

If a class C_1 is a refinement of a class C_2 then C_2 is termed an extraction from C_1. An extraction is called adequate if:

- (inv'[C_2] ∧ init[C_2]) ⇒ (∃e • inv'[C_1] ∧ init[C_1])
- for each event E:
  - inv[C_1] ⇒ (pre[C_2,E] ⇒ pre[C_1,E])
  - (inv[C_1] ∧ inv'[C_1]) ⇒ (post[C_2,E] ⇒ post[C_1,E])
  - (pre[C_2,E] ∧ inv[C_2] ∧ ∃e • inv[C_1] ∧ post[C_2,E]) ⇒ ∃e' • inv'[C_1]

where e is the components which are present in C_1 that are not present in C_2. If C_2 is an adequate extraction from C_1 then:

- any □-behaviour of C_1 is a □-behaviour of C_2;
- any ◊-behaviour of C_2 is a ◊-behaviour of C_1.

The cset object class, given in Figure 5.3, is an extraction from the aset object class, shown in Figure 5.2. To prove that cset is an implementation of aset (and hence of bset) it is necessary to shown that:

- (inv'[cset] ∧ init[cset]) ⇒ (∃s: s\{1\} • inv[aset] ∧ init[aset])
- for each event E:
  - inv[aset] ⇒ (pre[cset,E] ⇒ pre[aset,E])
  - (inv[aset] ∧ inv'[aset]) ⇒ (post[cset,E] ⇒ post[aset,E])
\[
- (\text{pre}[\text{CSET}.E] \land \text{inv}[\text{CSET}] \land \exists s : \text{set}[X] \land \text{inv}[\text{ASET}] \land \text{post}[\text{CSET}.E]) \\
\exists s' : \text{set}[X] \land \text{inv}'[\text{ASET}]
\]

**Figure 5.3 The CSET Object Class**

The proofs that the initialisation and Include event in CSET are adequate extractions from ASET follow:
Chapter 5 Formal Data Refinement

(1) **Claim** (initialisation):

\((\text{inv}'[\text{CSET}] \land \text{init}[\text{CSET}]) \Rightarrow (\exists S':\text{set}[I] \cdot \text{inv}'[\text{ASET}] \land \text{init}[\text{ASET}])\)

where

\(\text{init}[\text{CSET}] = (n' = 0)\)

\(\text{init}[\text{ASET}] = (n' = 0)\)

\(\text{inv}'[\text{CSET}] = (\text{dom } A' = 1..m) \land (n' \leq m)\)

\(\text{inv}'[\text{ASET}] = \text{inv}'[\text{BSET}] \land (\text{dom } A' = 1..m) \land (S' = A' \{1..n'\})\)

\(\text{inv}'[\text{BSET}] = (n' = #S') \land (n' \leq m)\)

**Proof:**

\(\text{inv}'[\text{CSET}] \land \text{init}[\text{CSET}] = (\text{dom } A' = 1..m) \land (n' \leq m) \land (n' = 0)\)

\(\text{inv}'[\text{ASET}] \land \text{init}[\text{ASET}] = (n' = #S') \land (n' \leq m) \land (\text{dom } A' = 1..m) \land (S' = A' \{1..n'\}) \land (n' = 0)\)

\((n' = #S') \land (n' \leq m) \land (\text{dom } A' = 1..m) \land (S' = A' \{1..n'\}) \land (n' = 0) \Rightarrow (S' = \emptyset)\)

so

\(\exists S':\text{set}[I] \cdot \text{inv}'[\text{ASET}] \land \text{init}[\text{ASET}] = (n' \leq m) \land (\text{dom } A' = 1..m) \land (n' = 0) \land (\exists S':\text{set}[I] \cdot S' = \emptyset)\)

thus

\((\text{inv}'[\text{CSET}] \land \text{init}[\text{CSET}]) \Rightarrow (\exists S':\text{set}[I] \cdot \text{inv}'[\text{ASET}] \land \text{init}[\text{ASET}])\)

(2.1) **Claim** (Include):

\(\text{inv}'[\text{ASET}] \Rightarrow (\text{pre}[\text{CSET}, \text{Include}] \Rightarrow \text{pre}[\text{ASET}, \text{Include}])\)

where

\(\text{inv}'[\text{ASET}] = (n = #S) \land (n \leq m) \land (\text{dom } A = 1..m) \land (S = A \{1..n\})\)

\(\text{pre}[\text{CSET},\text{Include}] = ((\forall j:1..n \cdot A(j) \neq i) \land (n < m)) \lor (\exists j:1..n \cdot A(j) = i)\)

\(\text{pre}[\text{ASET}, \text{Include}] = ((i \notin S) \land (n < m) \land (\forall j:1..n \cdot A(j) \neq i)) \lor ((i \in S) \land (\exists j:1..n \cdot A(j) = i))\)

(the \(\lor\) clauses are introduced because of the two cases for the Include event)

**Proof:**

\(\text{inv}'[\text{ASET}] \land \text{pre}[\text{CSET}, \text{Include}] = (n = #S) \land (n \leq m) \land (\text{dom } A = 1..m) \land (S = A \{1..n\}) \land ((\forall j:1..n \cdot A(j) \neq i) \land (n < m)) \lor (\exists j:1..n \cdot A(j) = i)\)

since

\((S = A \{1..n\}) \land (\forall j:1..n \cdot A(j) \neq i) \Rightarrow (i \notin S)\)

and

\((S = A \{1..n\}) \land (\exists j:1..n \cdot A(j) = i) \Rightarrow (i \in S)\)

so

\((\text{inv}'[\text{ASET}] \land \text{pre}[\text{CSET}, \text{Include}]) \Rightarrow ((i \notin S) \land (n < m) \land (\forall j:1..n \cdot A(j) \neq i)) \lor ((i \in S) \land (\exists j:1..n \cdot A(j) = i))\)
thus
\((\text{inv[ASET]} \land \text{pre[CSET}.\text{Include}]) \Rightarrow \text{pre[ASET}.\text{Include}]\)

thus
\(\text{inv[ASET]} \Rightarrow (\text{pre[CSET}.\text{Include}]) \Rightarrow \text{pre[ASET}.\text{Include}]\)

(2.2) Claim:
\((\text{inv[ASET]} \land \text{inv'[ASET]}) \Rightarrow (\text{post[CSET}.\text{Include}]) \Rightarrow \text{post[ASET}.\text{Include}])\)

where
\(\text{inv[ASET]} = (n=\#S) \land (n\leq m) \land (\text{dom A}=1..m) \land (S=A\{1..n\})\)
\(\text{inv'[ASET]} = (n'=\#S') \land (n'\leq m) \land (\text{dom A'}=1..m) \land (S'=A'\{1..n'\})\)
\(\text{post[CSET}.\text{Include}] = ((\forall j:1..n \cdot A(j)\not= i) \Rightarrow ((n'=n+1) \land (A'(n')=i))) \lor ((\exists j:1..n \cdot A(j)=i) \Rightarrow ((A'=A) \land (n'=n)))\)
\(\text{post[ASET}.\text{Include}] = ((i\not\in S) \Rightarrow ((i\in S') \land (n'=n+1) \land (A'(n')=i))) \lor ((i\in S) \Rightarrow (S'=S))\)

(the \lor clauses are introduced because of the two cases for the \text{Include} event)

Proof:
\(\text{inv[ASET]} \land \text{inv'[ASET]} \land \text{post[CSET}.\text{Include}] = (n=\#S) \land (n\leq m) \land (\text{dom A}=1..m) \land (S=A\{1..n\}) \land (S'=A'\{1..n'\}) \land ((\forall j:1..n \cdot A(j)\not= i) \Rightarrow ((n'=n+1) \land (A'(n')=i))) \lor ((\exists j:1..n \cdot A(j)=i) \Rightarrow ((A'=A) \land (n'=n)))\)

since
\((S=A\{1..n\}) \land ((\forall j:1..n \cdot A(j)\not= i) \Rightarrow (i\not\in S)\)
\((S'=A'\{1..n'\}) \land (n'=n+1) \land (A'(n')=i) \Rightarrow (i\in S)\)

and
\((S=A\{1..n\}) \land (S'=A'\{1..n'\}) \land (i\not\in S) \Rightarrow (i\in S)\)
\((S'=A'\{1..n'\}) \land (A'=A) \land (n'=n) \Rightarrow (S'=S)\)

so
\(\text{inv[ASET]} \land \text{inv'[ASET]} \land \text{post[CSET}.\text{Include}] \Rightarrow (i\not\in S) \Rightarrow ((i\in S') \land (n'=n+1) \land (A'(n')=i))) \lor ((i\in S) \Rightarrow (S'=S))\)

thus
\((\text{inv[ASET]} \land \text{inv'[ASET]} \land \text{post[CSET}.\text{Include}] \Rightarrow \text{post[ASET}.\text{Include}]\)

thus
\((\text{inv[ASET]} \land \text{inv'[ASET]} \Rightarrow (\text{post[CSET}.\text{Include}] \Rightarrow \text{post[ASET}.\text{Include}]\))
(2.3) **Claim**: 
\[(\text{pre[ASET. Include]} \land \text{inv[ASET]} \land \exists s : \text{set}(\mathcal{I}) \cdot \text{inv[ASET]} \land \text{post[ASET. Include]}) \Rightarrow \exists s' : \text{set}(\mathcal{I}) \cdot \text{inv[ASET]}\]

where
\[
\text{pre[ASET. Include]} = ((\forall j:1..n \cdot A(j) \neq i) \land (n < m)) \lor (\exists j:1..n \cdot A(j) = i) \\
\text{inv[ASET]} = (\text{dom } A = 1..m) \land (n \leq m) \\
\text{inv[ASET]} = (n \neq S) \land (n \leq m) \land (\text{dom } A = 1..m) \land (S = A \{1..n\}) \\
\text{post[ASET. Include]} = (\exists S : \text{set}(\mathcal{I}) \cdot ((n = S) \land (n \leq m) \land (\text{dom } A = 1..m) \land (S = A \{1..n\})) \land \\
((\forall j:1..n \cdot A(j) \neq i) \Rightarrow ((n' = n+1) \land (A'(n') = i))) \lor \\
((\exists j:1..n \cdot A(j) = i) \Rightarrow ((A' = A) \land (n' = n))) \\
\text{inv'[ASET]} = (n' \neq S') \land (n' \leq m) \land (\text{dom } A' = 1..m) \land (S' = A' \{1..n'\}) \\
\]

(the \lor clauses are introduced because of the two cases for the Include event)

**Proof**: 
\[
\text{pre[ASET. Include]} \land \text{inv[ASET]} \land \exists S : \text{set}(\mathcal{I}) \cdot \text{inv[ASET]} \land \text{post[ASET. Include]} = \\
((\forall j:1..n \cdot A(j) \neq i) \land (n < m)) \lor (\exists j:1..n \cdot A(j) = i) \land (\text{dom } A = 1..m) \land \\
(n \leq m) \land (\exists S : \text{set}(\mathcal{I}) \cdot ((n = S) \land (n \leq m) \land (\text{dom } A = 1..m) \land (S = A \{1..n\})) \land \\
((\forall j:1..n \cdot A(j) \neq i) \Rightarrow ((n' = n+1) \land (A'(n') = i))) \lor \\
((\exists j:1..n \cdot A(j) = i) \Rightarrow ((A' = A) \land (n' = n))) \\
\]

since
\[
(\exists S : \text{set}(\mathcal{I}) \cdot ((n = S) \land (n \leq m) \land (\text{dom } A = 1..m) \land (S = A \{1..n\})) \land \\
((\forall j:1..n \cdot A(j) \neq i) \Rightarrow ((n' = n+1) \land (A'(n') = i))) \lor \\
((\exists j:1..n \cdot A(j) = i) \Rightarrow ((A' = A) \land (n' = n))) \\
(\exists S : \text{set}(\mathcal{I}) \cdot (S' \neq S') \land (n' \leq m) \land (\text{dom } A' = 1..m) \land (S' = A' \{1..n'\})) \\
\]

so
\[
\text{pre[ASET. Include]} \land \text{inv[ASET]} \land \exists S : \text{set}(\mathcal{I}) \cdot \text{inv[ASET]} \land \text{post[ASET. Include]} \Rightarrow \\
(\exists S : \text{set}(\mathcal{I}) \cdot (n' = S') \land (n' \leq m) \land (\text{dom } A' = 1..m) \land (S' = A' \{1..n'\})) \\
\]

thus
\[
(\text{pre[ASET. Include]} \land \text{inv[ASET]} \land \exists S : \text{set}(\mathcal{I}) \cdot \text{inv[ASET]} \land \text{post[ASET. Include]}) \Rightarrow \\
\exists S : \text{set}(\mathcal{I}) \cdot \text{inv[ASET]} \\
\]

55
Chapter 5

Given that \textit{cset} is an implementation of \textit{bset}, the implementation of each schema of \textit{cset} in \textsc{fortran 77} is given in Figures 5.4a and 5.4b.

\begin{verbatim}
SUBROUTINE BSET
  IMPLICIT LOGICAL (B)
  SAVE
  PARAMETER (M=1000)
  INTEGER A(1:M)
  INTEGER N
  LOGICAL BF

  N=0
  RETURN

ENTRY EMPTY(B)
  B=N.EQ.0
  RETURN

ENTRY CARD(K)
  K=N
  RETURN

ENTRY MEMBER(I,B)
  J=0
  B=.FALSE.
  WHILE ((J.LT.N).AND.(.NOT.B)) DO
    J=J+1
    B=A(J) .EQ.I
  END WHILE
  RETURN
\end{verbatim}

\textbf{Figure 5.4a} The implementation of \textit{cset} in \textsc{fortran 77} (state and query schemas)

The code for the state schema includes the declaration of the constants and variables. The component names used in the \textit{cset} class are equivalent to the same names in upper case in the code. The initialisation \( n' = 0 \) is trivially coded as \( N=0 \).
Figure 5.4b The implementation of CSET in FORTRAN 77 (event schemas)

Take as an example of the code for the INCLUD entry block. The formal parameter \( i \) in the object class corresponds to \( I \) in the code. The while loop searches the array \( A \) between indexes \( I \) and \( N \) for the value of \( i \). If this search is successful then the routine terminates. Therefore, the terminating condition is \( (\exists J:1..N \cdot A(J) = I) \), corresponding to the precondition of the second CSET.Include schema in which no changes are made to the components. If the while loop terminates without finding \( I \) then the condition \( (\forall J:1..N \cdot A(J) \neq I) \) is satisfied, corresponding to part of the precondition of the first CSET.Include schema. Further, the condition \( n < m \) is checked by the conditional statement following the loop. If \( n < m \) then the except routine is called raising an exception and the INCLUD routine terminates. Finally, if the precondition \( ((N < m) \land (\forall J:1..N \cdot A(J) \neq I)) \) is
satisfied, the statements \( n = n + 1 \) and \( A(n) = i \) are executed and the postcondition for the event is satisfied.

## 5.4 An Alternative Representation

In the previous chapter the set data structure was represented both by an unordered list of items (as in the previous section) and by an ordered sequence. The ordered representation may also be derived through refinement. The \texttt{ASET} class for this case is shown in Figure 5.5 with the implementation being given in Figures 5.6a and 5.6b. The proofs proceed in an identical fashion to those for the previous case.

![Figure 5.5 The \texttt{ASET} object class for the ordered representation](image_url)
Chapter 5

Formal Data Refinement

SUBROUTINE MSET
IMPLICIT LOGICAL (B)
SAVE
PARAMETER (M=1000)
INTEGER A(1:M)
INTEGER N,L,U

N=0
RETURN

ENTRY EMPTY(B)
B=N.EQ.0
RETURN

ENTRY CARD(K)
K=N
RETURN

ENTRY MEMBER(I,B)
L=1
U=N
WHILE ((L.LE.U).AND.(I.NE.A((L+U)/2))) DO
  IF (I.LT.(A(L+U)/2)) THEN
    U=(L+U)/2-1
  ELSE
    L=(L+U)/2+1
  END IF
END WHILE
B=(L.LE.U)
RETURN

Fuzzy

Figure 5.6a The implementation of ordered \texttt{CSET} in FORTRAN 77 (state and query schemas)
Figure 5.6a The implementation of ordered \( \text{CSET} \) in FORTRAN 77 (event schemas)

Therefore, to derive an implementation of a class through refinement it is first necessary to make a precondition-conservative refinement of the class. This refinement should contain all the additional components, invariants, preconditions and postconditions required by the
implementation. An adequate extraction is then made from the refined class to yield a new class, which is an implementation of the original class.

Both implementation through refinement and the use of an abstraction function achieve the same results (formally correct software) but the former captures the intuition required in the latter as a structured procedure. The process of making a precondition-conservative refinement followed by an adequate extraction supports a systematic approach to deriving an implementation. The proof obligations are incremental and concerned with the preservation of the behavioural properties of the system. In contrast, the classical approach (detailed in Chapter 4 and Appendix C) relies upon observing a relation (the abstraction function) between an abstract specification and an implementation. Additionally in the new approach, both the specification and implementation are expressed within the same domain (the schema notation), which simplifies the reasoning involved.
CHAPTER 6 GENERATORS AND ITERATION

It was noted in Chapter One that iteration statements are an important point of interaction between the data and the control structure of a language. To support iteration over a data abstraction it is necessary to be able to access all elements in the structure in an efficient and non-destructive manner. Generators provide such access. They suppress the detail of how iteration over the data structure is implemented and allow loops to operate on abstract entities without explicit dependence upon the representation of those entities [Bishop, 1986] [Bishop, 1990]. Generators are supported as special types of semicoroutine in languages such as CLU [Liskov, 1981], Alphard [Shaw, 1981] and Icon [Griswold, 1983].

The results of an empirical analysis of the commonly used loop constructs in FORTRAN 77 are presented and two different generator constructs are identified. These are specified and implemented for the set object class.

6.1 Generators and Iterative Abstraction

A generator adheres to certain requirements of the semantics of iteration [Shaw, 1977] which will usually include:

- It must provide at least two functions (Init and Next). Init initialises the generator and Next returns a pair \( \langle v, f \rangle \) for each call, where \( v \) is the value bound to the loop variable and \( f \) is a flag value [Berztiss, 1988]. For a generator which goes through a finite iteration sequence, the final value of the flag is some finish indicator;

- Invocation of these functions in a prescribed order produces a sequence of values to bind to the loop variable.

Four actions are associated with the use of a generator, calling, resuming, yielding and returning [Liskov, 1986]. Calling corresponds to the initiation of the loop, whereas resuming refers to all successive calls after initialisation. When a generator completes a resumption call it yields a value until a point is reached where all values have been yielded, whereupon the generator returns. Control is transferred back from the generator to the loop statement via the yielding process, whereas returning signifies the end of the generator.

Generators and loop control constructs have been considered on several occasions in the literature. Shaw, Wulf and London [Shaw, 1977] outlined the problems of consistency which result from allowing changes to the data structure during iteration and showed that loop
proof rules were simplified if the restriction of not allowing changes to the data structure during iteration was enforced. Later it will be shown that this may be too severe a restriction to place upon the iterative constructs.

Berztiss [Berztiss, 1988] discussed the synchronisation problems associated with the need for more than one generator over a data structure. When two or more generators access a data structure it is necessary to be able to identify which is required when resuming. In addition, the generators must be able to operate independently over the data structure. The tagging process he outlined gave a simple mechanism for overcoming this problem and represented an improvement upon his previously proposed controlled iteration constructs [Berztiss, 1980].

An important distinction between what may be termed unconditional and set selective loops arises from the work of Pratt [Pratt78]. An unconditional loop over a data structure executes the loop body for every element of the structure in turn. A set selective generator executes the loop body for only those elements of the structure which fulfil a given condition. The unconditional loop is a special case of the set selective loop in which the selection condition for elements is simply true.

In their consideration of abstraction in the language Alphard, Shaw, Wulf, and London [Shaw, 1977] presented two distinct iterative constructs, *for* and *first*. The *for* statement was used in iterations over a complex data structure where several elements of the structure are to be yielded in turn. The *first* statement searched a data structure and returns the first element of the structure which satisfies a given condition. Both the *for* and *first* statements could be specialised for a given use via a generator.

Waters [Waters, 1979] presented four ways in which the logical structure of loops were built. These were the basic loop (all the computation in the body of the loop can effect its termination), the augmentation loop (a basic loop augmented by additional code fragments), the filter loop (takes a sequence of values and produces a restricted sequence of values) and the interleaving loop (two loops intermingled so that their execution is synchronised).

A number of generator constructs may be supported and it is important to determine which are required by a particular language. In the next section the results of an empirical analysis of FORTRAN 77 loop constructs are presented.

### 6.2 Loop Constructs in FORTRAN 77

The generator constructs required by a language are governed by the nature of the data access methods used in the iterative structures of that language. The classifications presented in [Shaw, 1977] [Pratt, 1978] [Waters, 1979] provide insight into the commonly occurring loop control and data structure access mechanisms which are present in
Chapter 6 Generators and Iteration

programming languages. To determine which mechanisms are commonly used for FORTRAN 77 an empirical analysis of some ninety six FORTRAN 77 files was carried out by the author. The files varied in length from twenty to around two thousand lines of code and were written by programmers with differing levels of experience. The analysis was a three stage process. Firstly, the loop control and data access mechanisms available were classified. Secondly, an analysis tool was developed, which automatically detected and classified the loop control mechanisms in a FORTRAN 77 source file. This tool generated a directed graph of the source file and detected loops by finding the back-arcs in the graph (a full description of the use of directed graphs in control-flow analysis can be found in [Paige, 1975] [Paige, 1977]). The only data structure present within FORTRAN 77 is the array. Once a loop control mechanism had been classified the usage of arrays within the loop was determined. This allowed the data access mechanisms within each loop to be identified. The empirical analysis of FORTRAN files has previously been reported [Knuth, 1971] [Robinson, 1976] but no other study had considered the data access mechanisms in loop constructs.

Four categories of array access were identified:

- Unconditional loops over an array
  The loop body is executed for every element of the array;

- Set selective loops over an array
  The loop body is only executed for those elements of the array which return true for a given predicate function applied to the index of the element and/or the element itself;

- Search loops over an array
  The loop searches the array until the first element is found which returns true for a given predicate function applied to the index of the element and/or the element itself;

- None of the above
  The loop does not iterate over an array or iterates over the array in a manner not previously described.

Some two hundred and forty one loops were analysed and the percentage falling into each of the classification of data access mechanisms was found to be:

- Unconditional loops 72.6%
- Set selective loops 12.5%
- Search loops 9.9%
- None of the above 5.0%
These results would indicate that unconditional, set selective and searching generators should be supported by a data abstraction mechanism for FORTRAN 77. (Note that since the unconditional generator is simply a special case of the set selective generator it is only necessary to implement the set selective and searching cases.)

6.3 A Set Selective Generator over the Data Class

In this section a set selective generator for FORTRAN 77 is developed using the object-oriented set-theoretic specification technique. Both the restrictions of FORTRAN 77 and the normal usage of iterative constructs in the language greatly influence the choice of a suitable abstraction.

The provision of a generator should reduce the complexity associated with selecting successive elements from the data structure and yield straightforward proof obligations for these iterations. In addition, the time and storage overheads incurred should be minimised. Although the loop proof rules are simplified if the restriction of not allowing changes to the data structure during iteration is enforced, the study of the data structure access mechanisms showed that some ninety six (37.2%) of the loops analysed made changes to the array over which they iterated. It is therefore unreasonable to exclude changes to the data structure during iteration and a solution should be adopted which allows changes whilst avoiding the problems associated with consistency. One such solution involves making a copy of the data structure for use by the generator and allowing changes only to the original structure. However, the time and storage overheads incurred when copying are unacceptable for all but the smallest data structures. A more satisfactory solution is to yield the elements of the data structure in order (assuming the relational operators are defined over the underlying element type). This allows a current position to be maintained. Only changes to elements of the data structure with values greater than the current position affect the generator operation. The specification of a set selective generator having this behaviour is given in Figure 6.1 and the corresponding refinements for the basic set operations are shown in Figure 6.2, along with the implementation of the generator. Only the case of a single generator over a set is considered.

A generator object is first instantiated by the state schema \textbf{iset} which introduces an additional state component \texttt{j}. The object is initialised by \texttt{iset.first}, which takes a predicate function \texttt{p} as a parameter and yields the first \texttt{<1,b>} pair. The predicate function \texttt{p} determines which members of the set will be yielded by the generator. Only those elements of the set for which the function \texttt{p} returns a value true are yielded. \texttt{iset.next} yields successive \texttt{<1,b2>} pairs and also requires the function \texttt{p} and a variable \texttt{b1} as parameters. If \texttt{b1} has the value false then the generator returns, allowing early termination from within the loop body. Note also that when \texttt{b2} (or \texttt{b} in the case of \texttt{first}) is false the generator has
returned (i.e., all elements satisfying \( p \) are exhausted) and the value of \( i \) is undefined. Item values from the set which satisfy the function \( p \) are yielded in increasing order with a component \( j \) holding the last item value yielded. Any subsequent values which are yielded must be greater than \( j \). If alterations are made to the set by the addition or removal of items with values less than the current value of \( j \) then these changes have no effect upon the operation of the generator.

![Figure 6.1 The specification of the set selective iterator](image)

The correct behaviour of the generator is strongly dependent upon the assumption that the intended calling conventions is adhered to, which are detailed in the next section. Even so, the approach suffers from certain drawbacks. Firstly, the predicate \( p \) is a parameter to the \texttt{ISET.First} and \texttt{ISET.Next} events (a requirement imposed by FORTRAN 77). This arises as a consequence of allowing changes to the data structure during the execution of the generator and makes it possible in principle to change the predicate for successive resumptions of the generator. In addition, the value of \( i \) is undefined in the final \(<i,b2>\) pair yielded. The requirements that the predicate function remains the same and that the undefined value of \( i \) is not used are thus obligations carried over into the user-code. Finally, the representation of the \texttt{BSET} object class has an effect upon the efficiency of the implementation of the generator. If the set is implemented by an unordered array the \texttt{First} and \texttt{Next} event implementations must search the whole set for the smallest legitimate item (if one exists). However, if the set items are stored in order then just a current position (as opposed to a current data value) can be maintained and no searching is required.
This generator implementation is given in Figures 6.2a, 6.2b and 6.2c. The first and next events are implemented by the first and next entry points respectively.

**SUBROUTINE ISET(MV)**

```
IMPLICIT LOGICAL (B,F)
PARAMETER (MX=1000)
EXTERNAL BP
SAVE
INTEGER A(1:MX), N, M, L, U
INTEGER J
IF ((MV.GT.MX).OR.(MV.LT.O)) THEN
   CALL EXCEPT
   RETURN
END IF
M=MV
N=0
J=0
RETURN
ENTRY EMPTY(B)
B=N.EQ.0
RETURN
ENTRY CARD(K)
K=N
RETURN
ENTRY MEMBER(I,B)
L=1
U=N
WHILE ((L.LE.U).AND.(I .NE. A((L+U)/2))) DO
   IF (I.LT.A((L+U)/2)) THEN
      U=(L+U)/2-1
   ELSE
      L=(L+U)/2+1
   END IF
END WHILE
B=(L.LE.U)
RETURN
```

**Figure 6.2a** The implementation of the object class **ISET** as an ordered list with the set selective generator (state and query schemas)
Figure 6.2b The implementation of the object class \texttt{iset} as an ordered list with the set selective generator (event schemas)
Chapter 6

Generators and Iteration

ENTRY FIRST(BP,I,B)
J=0
B=.FALSE.
WHILE ((J.LT.N) .AND. (.NOT.B)) DO
  J=J+1
  B=BP(A(J))
END WHILE
IF (B) I=A(J)
RETURN

ENTRY NEXT(BP,I,B)
IF (B) THEN
  B=.FALSE.
  WHILE ((J.LT.N) .AND. (.NOT.B)) DO
    J=J+1
    B=BP(A(J))
  END WHILE
  IF (B) I=A(J)
END IF
RETURN
END

Figure 6.2c The implementation of the object class ISET as an ordered list with the set selective generator (generator schemas)

FORTRAN 77 allows the parameters b1 and b2 of ITER.Next to be implemented as a single subroutine parameter b, which corresponds to b1 at input and to b2 at output. Note that since changes are permitted the value of J is incremented/decremented when a data item whose value is less than the last value yielded is included/excluded.

6.4 Use of the Set Selective Generator

Statements in programs are characterised by predicates, which relate the state of computation before invocation to the state afterwards. However, generators are not invoked in the usual sense. Consider the use of the generator shown in Figure 6.3.
Chapter 6

Generators and Iteration

IMPLICIT LOGICAL (B)

CALL FIRST(BP, J, B)
WHILE (B) DO
CALL EVAL(J, B)
CALL NEXT(BP, J, B)
END WHILE

Figure 6.3 Use of the generator

The assumption here is that the calling pattern always corresponds to that shown above. The loop calls an arbitrary subroutine EVAL for every element yielded by the generator. EVAL returns a boolean flag B which indicates whether iteration should be continued. If B is false then the loop terminates. (Note that if B is assigned the value false by EVAL then calling NEXT results in the generator returning and termination of the loop.) The call to EVAL may be replaced by any arbitrary loop body. In this example J is assumed to be an integer with the set being of base type integer. It is assumed that the function BP does not make changes to the set during iteration. In addition, calling the FIRST routine without calls to NEXT provides a searching generator in FORTRAN 77, with the calling pattern given in Figure 6.4.

IMPLICIT LOGICAL (B)

CALL FIRST(BP, J, B)
IF (B) CALL EVAL2(J)

Figure 6.4 Use of the searching generator

In the cases where the loop body (EVAL) does not make changes to the set, the proof rules associated with the loop will be greatly simplified. A further abstraction from this loop can then be made to provide a single routine. The ENTRY block for the implementation of this routine is given in Figure 6.5, and may replace the FIRST and NEXT routines. In this case the routine implements the loop given in Figure 6.5 as a single event encapsulated within the definition of the set and therefore avoids the drawbacks previously outlined. Note that this routine provides universal quantification ("for all") over the set, provided the early-termination option is not exercised.
EXTERNAL BP, EVAL
ENTRY FORALL (BP, EVAL)
J=1
B=.TRUE.
WHILE ((J .LE. N) .AND. (B)) DO
   IF (BP(A(J))) CALL EVAL(A(J), B)
   J=J+1
END WHILE
RETURN

Figure 6.5 A single routine with identical functionality to the generator loop

Although language restrictions increase the complexity of the generators a satisfactory compromise between language limitations and the generator constructs can be achieved. More sophisticated structuring, as discussed in [Bishop, 1990], cannot be achieved in the FORTRAN context.

6.5 Binary Operations

It is now possible to implement binary operations over sets in a procedural style. The subroutine given in Figure 6.6 implements set union for the sets indexed by x1 and x2 such that the set indexed by x1 is made equal to the union of x1 and x2. Multiple instances of sets with generators are assumed to be available in the style given in Chapter 4 and the external function `BDUMMY` is assumed to be return a value true to the set selective generator in all cases and does not change the contents of either set under consideration.

```
SUBROUTINE UNION(X1,X2)
IMPLICIT INTEGER (X)
LOGICAL B
INTEGER ITEM
EXTERNAL BDUMMY

CALL FIRST(X2,BDUMMY,ITEM,B)
WHILE (B) DO
   CALL INCLUD(X1,ITEM)
   CALL NEXT(X2,BDUMMY,ITEM,B)
END WHILE
RETURN
END
```

Figure 6.6 The set union operation

Similar implementations for set intersection (x1' = x1 \ x2), set difference (x1' = x1 \ x2), set equality (B=x2=x1) and subset (B=x2 \ x1) are given in Figure 6.7.
Figure 6.7 The set intersection, set difference, set equality and subset operations
6.6 A Larger Example

A subroutine \texttt{MERGE} is given in Figure 6.8, which takes the lowest \(n\) elements satisfying the predicate function \(BP\) drawn from either of two sets (identified by \(x1\) and \(x2\)) and inserts into a new set (identified by \(x3\)). The return value of \(n\) indicates the number of elements inserted into the set \(x3\).

```fortran
SUBROUTINE MERGE(X1,X2,NE,BP,X3)
IMPLICIT INTEGER (X)
IMPLICIT LOGICAL (B)
EXTERNAL BP
LOGICAL B1,B2
INTEGER ITEM1,ITEM2,COUNT

COUNT=0
CALL NEWSET(X3,N)
CALL FIRST(X1,BP,ITEM1,B1)
CALL FIRST(X2,BP,ITEM2,B2)
WHILE ((B1.AND.B2).AND.(N.GT.COUNT)) DO
  IF (ITEM1.LT.ITEM2) THEN
    CALL INCLUD(X3,ITEM1)
    CALL NEXT(X1,BP,ITEM1,B1)
  ELSE IF (ITEM2.LT.ITEM1) THEN
    CALL INCLUD(X3,ITEM2)
    CALL NEXT(X2,BP,ITEM2,B2)
  ELSE
    CALL INCLUD(X3,ITEM1)
    CALL NEXT(X1,BP,ITEM1,B1)
    CALL NEXT(X2,BP,ITEM2,B2)
  END IF
  COUNT=COUNT+1
END WHILE
IF (N.GT.COUNT)
  IF (B1) THEN
    WHILE (B1.AND.(N.GT.COUNT)) DO
      CALL INCLUD(X3,ITEM1)
      CALL NEXT(X1,BP,ITEM1,B1)
      COUNT=COUNT+1
    END WHILE
  ELSE
    WHILE (B2.AND.(N.GT.COUNT)) DO
      CALL INCLUD(X3,ITEM2)
      CALL NEXT(X2,BP,ITEM2,B2)
      COUNT=COUNT+1
    END WHILE
  END IF
N=COUNT
RETURN
END
```

Figure 6.8 A routine to merge two sets

The complexity of the loop computation is hidden with the two loop variables, which would be required if \texttt{MERGE} were implemented using the built-in types of FORTRAN 77, being eliminated. If the built-in types were used the efficiency of the routine would dictate the ordering of the data structures (arrays) prior to the merging process. Such detail may instead be included in the specification of the generator and removed from the user routines. The provision of generators over the data structure therefore reduces the program size, improves the code readability and eases the proof of program correctness.
The advantages of data abstraction methods may be obtained in FORTRAN 77 and the data independence provided allows parallel implementations to be considered. This is without changes to the original specification or any user-code. Various parallel implementations are described in the next chapter.
The mainstream use of computers is increasing in sophistication to a point where single processor systems are unable to cope with the demand. Parallel systems, in which several processing elements execute operations simultaneously, represent the most feasible avenue for achieving the increase in performance required. Within the field of parallel architectures, distributed memory multi-processor systems show increasing capability of successfully performing a varied number of computations [Hey, 1987] [Hey, 1988] [Otto, 1988]. Such systems comprise many processing elements, each with their own local memory, which communicate through message passing. They offer advantages over their shared memory counterparts, in that system expansion is not limited by the requirement for a common communications bus between the processing elements and a shared memory. The distributed memory approach allows very highly parallel systems consisting of many processing elements to be constructed.

The extension of sequential languages with new operations, which allow concurrency and synchronisation to be expressed, is the most viable means of migrating sequential systems onto distributed memory multi-processor architectures. These extensions are usually based upon a message passing synchronisation model such as CSP [Hoare, 1985]. A computing system is regarded as a collection of concurrently active sequential processes or tasks, which communicate through message passing over channels. Each process has access to only its own region of memory for code and data. All the code for a process is written in the ordinary sequential languages except for the added synchronisation features. These additional features provide a means of forcing a process to wait until it either receives a message on an input channel or is able to send a message on an output channel. The model requires a distributed data mapping for the complete program as no two processes are able to share the same area of memory.

For sequential systems, such as those written in FORTRAN 77, it is difficult to conceive a truly distributed data mapping for a complete program. For example, many large FORTRAN 77 applications contain data and data storage areas which are shared via the COMMON and EQUIVALENCE constructs [ANSI, 1978]. Support for shared data structures, concurrently accessible by a number of independent processes, is required. Such a structure allows update and access operations on the data to proceed concurrently (and potentially in parallel) whilst maintaining the consistency of the shared data. Abstraction provides a means of hiding such details from the parallel processes which access that structure.
Chapter 7 Parallel Data Structures

In this chapter methods of providing parallel data access are discussed. A server approach utilising a number of dedicated processors provides the required encapsulation and several processor architectures for implementing such a server are compared. A pipeline implementation of a search tree is shown to be an efficient and flexible means of providing parallel data structures for distributed memory architectures. The pipeline architecture proposed by Carey and Thompson [Carey, 1984] is presented for implementing a 2-3-4 tree and this structure is further developed to give a generalised $2^p-2^q$ search tree, which offers a number of advantages.

7.1 Concurrently Accessible Data Structures

Quinn’s study of dictionary style search operations [Quinn, 1987] gave guidance as to how increases in the throughput of data structure queries can be achieved on a parallel architecture. The sequential algorithm for a single search operation on a balanced B-tree [Comer, 1979] has logarithmic complexity [Knuth, 1969b]. If an $n$-element table is to be searched then the worst case for the sequential binary search is $\log_2 n + 1$ comparisons. The improvement in the response time which may be achieved by a parallel algorithm for a single search can be logarithmic only in the number of processors used (for $N$ processors the improvement in response time is $\log_2 N + 1$) [Quinn, 1987]. (The response time is the average time taken to process a single query.) Therefore, the strategy suggested by Quinn is to seek increases in throughput for a series of searches, insertions and deletions operating in parallel. (The throughput is the rate at which queries are processed by the system.)

Historically, much of the interest in parallel data structures has centred around shared memory multi-processor systems due to their earlier development and commercial availability. Samadi [Samadi, 1976] discussed the implementation of a B-tree [Comer, 1979] in a multi-user system and derived a simple locking protocol through the use of semaphores to avoid deadlocks. Held and Stonebraker [Held, 1978] noted that for the concurrent accessing and updating of a B-tree such a locking protocol was required. Ellis [Ellis, 1980a] proposed an algorithm for allowing concurrent search and insert operations to take place in an AVL-tree [Foster, 1965], and a locking protocol was again used to ensure consistency of the structure. Manber and Ladner [Manber, 1982] relaxed the balancing property of the AVL structure and allowed deletions. They hoped that insertions and deletions would maintain some form of balanced structure. However, Eppinger [Eppinger, 1983] showed that this would only be the case under certain restricted conditions. Ellis [Ellis, 1980b] also described a concurrent search and insertion algorithm, which extended the work of Bayer and Schkolnick [Bayer, 1980] in this area and a similar locking protocol was devised by Kung and Lehman [Kung, 1980].
In the case of distributed memory architectures, the data structure is partitioned between the local memory of the processing elements. Two schools of thought have emerged for providing concurrently accessible shared data structures in these architectures. The first relies on a single copy of the data structure being maintained in a number of dedicated processing elements. This may be termed a data structure server approach. An interface to the structure is provided allowing accessing and updates to take place in parallel, the control of which is hidden from the application environment. In these systems tree structures are a preferred storage mechanism. The increasing speed and density of very large scale integrated (VLSI) circuits has led to the development of several customised designs for database machines based upon tree structures [Bentley, 1979] [Song, 1980] [Ottman, 1984] [Bonuccelli, 1983] [Atallah, 1985] [Somani, 1985] [Chang, 1988], utilising O(n) processing elements to form a tree of n entries. Carey and Thompson [Carey, 1984] proposed a pipeline architecture using O(log n) processing elements to implement a 2-3-4 tree. A 2-3-4 tree is a tree in which each vertex which is not a leaf has two, three or four sons, and every path from the root to a leaf is of the same length. This was a similar architecture to that used by Tanaka, Nozaka and Masuyama [Tanaka, 1980] in their pipelined binary tree system. Fisher [Fisher, 1984] also proposed a pipeline system which used a pipeline length proportional to the length of the search key used to distinguish the data records. He demonstrated that the processor-profligate VLSI architectures are not always the best route to a high performance system and that O(log m) processor systems are a more viable option in many cases. In his concluding remarks, Fisher noted that Carey and Thompson’s design may be preferred in systems with relatively short keys and that tree machines can be expected to be of more value in applications where a query or update requires O(m) time on a uniprocessor. Walden and Sere [Walden, 1989] presented a survey of distributed memory processor architectures based upon transputers [Inmos, 1986] for implementing document retrieval systems. This was based upon the processor-farm paradigm in which each processor executed the same program on different data. They compared array, ring and tree structures and concluded that the tree was superior due to the shorter communications paths between processing elements, a result supported by Green and Paddon [Green, 1988].

In the server approach, as the number of accessing processors increased the requirement that all queries must pass through the root processor caused a single server bottle-neck performance degradation. This resulted in system expansion being limited. The alternative is to distribute the data structure across a number of processing elements in the system. The data structure is either divided into disjoint portions or some parts are replicated in several processing elements. Replication serves to increase availability by placing copies of heavily used information at several sites. This raises the issue of maintaining consistency to an appropriate degree and a number of general-purpose mutual consistency algorithms have been proposed [Stonebaker, 1979] [Thomas, 1979]. A distributed version of an extended hash
file was presented in [Ellis, 1985], which utilised replication, and a similar structure was
developed by Li and Atwood [Li, 1987]. Replication overcomes the bottle-neck constraint of
single server systems; however, the problem of consistency increases the number of
messages which are sent between the processing elements holding copies of the data
structure. For systems where many copies of the structure are required, this results in
performance degradation and limits system expansion.

The server approach, although constrained by a bottle-neck, provides encapsulation of the
data structure with the control of the concurrent access and updating of the structure being
carried out entirely by the dedicated processors. In the case of replication each application
processor holding a portion of the data structure has to be concerned with the consistency
of that portion. The encapsulation provided by the server allows the parallel implementation
of the data structure to be hidden from external processes.

The mathematical rigour of models such as CSP [Hoare, 1985] provides a framework for the
series of transputer-based programming paradigms which permitted a similar analysis of
system performance. The paradigms of interest in the context of server-based parallel data
structures are the processor-farm and algorithmic pipeline.

Processor-farms consist of a farmer processor that distributes independent packets of work
to a set of worker processors and receives back the results. Various topologies may be used
including linear chains and ternary trees. Algorithmic pipelines distribute the algorithm
associated with the computation over a linear array of processors. May and Shepherd [May,
1987] gave a number of considerations for choosing between farming and pipeline
topologies for an application. These included consideration of throughput, response time
and the memory capacity of the processors. Implementations of parallel data structures
which utilise one of these paradigms are contrasted in the next sections. It is assumed that
the data structure is to be accessed by a number of independent asynchronous processes.

7.2 Parallel Implementations of Data Structures

For the set data structure described in Chapters 4, 5 and 6 to be supported by a parallel
implementation, the implementations must:

- permit key values to be inserted and deleted from the structure and allow tests to
determine whether a given key value is contained within the structure;

- permit generators to iterate over the data structure.
These requirements exclude hash table algorithms from the set of possible solutions since generators cannot be supported (for data items extracted from a universal set of size #H, a generator over a hash table executes in \(O(#H)\) time [Aho, 1974]). However, several suitable alternative have been proposed in the literature:

- Chain and ring structures utilising a processor-farm [Walden, 1989];

- \(O(n)\) search trees utilising a processor-farm (where \(n\) is the number of key values stored) [Bentley, 1979] [Song, 1980] [Ottman, 1984] [Bonuccelli, 1983] [Atallah, 1985] [Somani, 1985] [Chang, 1988];

- \(O(\log_2 n)\) pipeline implementations of search trees utilising an algorithmic pipeline [Tanaka, 1980] [Carey, 1982] [Carey, 1984] [Fisher, 1984].

### 7.3 A Processor-Farm Implementation of Rings and Search Trees

The analysis presented in this section is based upon that introduced by Pritchard [Pritchard, 1988]. Two alternative processor topologies are considered, the processor chain (which is simply transformed into a ring) and the ternary tree, shown in Figure 7.1.

![Figure 7.1 Alternative Farm Topologies](image-url)
Three parameters were introduced by Pritchard:

- $T_{\text{calc}}$: time for one processor to complete one result;
- $T_{\text{comm}}$: time for one inter-processor transfer of one result;
- $T_{\text{setup}}$: time to set up each processor.

The linear chain was shown to have a maximised throughput ($S_N$) for an $N$ processor chain of:

$$
N \leq N_c : S_N = \frac{1}{2T_{\text{setup}}} \left[ 1 - \left( \frac{T_{\text{calc}}}{T_{\text{calc}}+T_{\text{setup}}} \right)^N \right]
$$

$$
N \geq N_c : S_N = \frac{1}{T_{\text{comm}}+T_{\text{setup}}}
$$

where $N_c$ is the critical value of $N$ for the largest useful chain. (There is a limit to the number of worker processors that can usefully be employed, owing to saturation of the communications bandwidth.) The value $N_c$ was given by:

$$
\left( \frac{T_{\text{calc}}-T_{\text{setup}}}{T_{\text{calc}}+T_{\text{setup}}} \right)^{N_c} = \left( \frac{T_{\text{comm}}}{T_{\text{comm}}+T_{\text{setup}}} \right)
$$

Hey [Hey, 1989] demonstrated that for a computationally intensive farming system both a triple chain and the ternary tree topologies have superior throughput over the linear chain in a transputer system. (Computational speedups of approximately 6.4 and 6.3 were achieved for the triple chain and ternary tree respectively as opposed to 4.8 for the linear chain. Computational speedup being measured with respect to the purely sequential implementation.) This was caused by the throughput of the linear chain being limited by the communication bandwidth from the farmer to the chain. However, Pritchard [Pritchard, 1988] noted that the linear chain was the simpler topology and in some cases was optimal.

The response time for the linear chain of $N$ processors was simply bounded by $N$. For a unidirectional ring (queries and replies travelling in one direction around the ring) the response time $T_R$ was given by:

$$
T_R = N(T_{\text{comm}}+T_{\text{setup}})+T_{\text{calc}}
$$

For the ternary tree the response time is $O(\log_2 N)$ (as opposed to $O(N)$ for the ring).

When a farming topology is used to implement a parallel data structure a further consideration is the optimum size of the data structure partition which should be stored at each processor. This effects the values of $T_{\text{calc}}$ and $N$. 

80
Although high throughputs and relatively low response times have been achieved with a farming topology it suffers from drawbacks when applied to parallel data structures. By way of example, consider the implementation of a set data structure on a linear ring topology. The set is partitioned between the workers in the ring. For a query (insertion, deletion or membership test) to be processed by a single worker processor the set should be partitioned into a number of contiguous subsets each of which is stored by a single worker. This is a straightforward task for those sets whose approximate size and shape is known in advance as workers can be allocated appropriate subsets. Unfortunately, in most cases the size and shape of the set fluctuates and is not well known in advance. To maintain an even distribution of key values between worker processors a periodic redistribution of data elements is required. Query access to the structure should be halted whilst the transfer of data elements between processors takes place. In most cases it is possible to restrict this transfer to processors which are physically connected (neighbours in the ring structure). Redistribution adds to the complexity of the inter-processor communication in the system and is a time consuming task (since it places a heavy burden upon inter-processor communication) so where possible should be avoided.

The allocation of worker processors to the farm may either be static (at compile-time) or dynamic (when worker processors are required by the structure). A static allocation is the simplest approach but may result in poor processor utilisation. A dynamic allocation gives improved processor utilisation but is somewhat difficult to achieve in current systems (Supernode for example [Harp, 1987]).

The ternary tree topology minimises the path length between the farmer and the worker processors in the system. However, the problem of data redistribution is more severe in this topology and may well involve the transfer of key values between processors which are not physically connected to one another.

If the approximate size and shape of the data structure is known prior to the decision concerning the choice of topology then a farming system is a good solution. However, the redistribution of data elements which results when the shape of the structure is not known in advance is a drawback to its application.

### 7.4 A Pipeline Implementation of Search Trees

Carey and Thompson [Carey, 1984] implemented a 2-3-4 search tree storing n key values in a linear pipeline of \([\log_2 n + 1]\) processing elements. A 2-3-4 tree is a tree in which each vertex which is not a leaf has two, three or four sons, and every path from the root to a leaf is of the same length [Aho, 1974]. Each processor holds a level of the tree structure in local
memory and the last processor N₁ stores the actual data items, as shown in Figure 7.2. The tree grows in an upwards direction from processor N₁.

![Diagram of parallel tree structure](image)

**Figure 7.2** Carey and Thompson’s parallel architecture for balanced tree maintenance

Data items consist of a primary key and an uninterpreted data field and the scheme allows insertions, deletions, exact-match searches and range queries. Each operation completes after $O(\log_2 n)$ delay and as many as $\lceil \log_2 n + 1 \rceil / 2$ operations may be at varying stages of execution. The searching operation is a pipeline version of the normal B⁺ tree searching operation [Comer, 1979] and the insert and delete operations are based upon the top-down node-splitting scheme presented by Guibas and Sedgewick [Guibas, 1978]. In this scheme transformations are applied during a single traversal of the tree for an update operation. There is no need to maintain a record of the tree structure during the traversal since no portion of the search path need be traversed again to restore the balancing condition. In this respect, the 2-3-4 tree has advantages over the 2-3 tree as manipulations can be performed in this top-down fashion [Guibas, 1978].

The insert operation performs node splitting on encountering a four branch tree node as depicted in Figure 7.3. The optional pointers are represented by the dashed lines and the search path pointer is indicated by the small filled circle. This transformation ensures that any future node splitting does not cause upward propagation in the tree structure thereby allowing the transformation to be applied in the top-down fashion. Deletion proceeds in a similar fashion with the appropriate transformation in Figures 7.4a, 7.4b or 7.4c being applied when a two branch tree node is encountered. Each deletion transformation ensures that the next node on the search path has at least three sons.
Since the pipeline operates on a request/reply paradigm, half of the processors can be processing requests at any given time. The scheme requires $O(\log_2 n)$ time per tree operation, but allows $O(\log_2 n)$ concurrency on the operations; one operation completes every $O(1)$ time. As queries enter the pipeline at the top and replies leave at the bottom the root bottle-neck problem normally encountered in tree architectures is removed. The
protocol between processors is simple and each processor needs only to communicate with its two neighbouring processors. Changes in the structure of a tree level are restricted to a pair of processors and are simply implemented as part of the protocol. The search tree is balanced and so provides optimal and predictable search times. In addition, no redistribution of key values between processors is required (a significant advantage of this approach). A potential drawback is the increasing memory requirement of the processors in the pipeline, given that a processor storing level i (i≥1) of the tree requires up to four times the amount of storage for tree nodes than the processor storing level i+1.

Conducting a similar analysis for this pipeline to that made by Pritchard for the farming topology gives a throughput of:

\[ S_N = \frac{1}{2(T_{\text{calc}} + T_{\text{setup}} + T_{\text{comm}})} \]

In this case \( T_{\text{calc}} \) is the maximum processing time for any of the pipeline stages. The response time for an N processor pipeline (\( N = \log_2 n + 1 \)) is:

\[ T_R = N(T_{\text{comm}} + T_{\text{setup}} + T_{\text{calc}}) \]

In general the theoretical throughput in this case is not as great as for the farm topology although the response times are comparable. In some pipeline algorithms it is possible to overlap the processing of the same query in several of the pipeline stages. This leads to improvements in both throughput and response time (in Chapter 9 it is demonstrated that this can be achieved in the pipeline implementation of these search trees). The significant advantage of the pipeline tree system is that no redistribution of the data between processors is required so, although maximum throughputs may not be as great as for farming topologies, system availability is unaffected by redistribution and the inter-processor communication is straightforward.

The 2-3-4 search tree may be developed further to give a \( 2^P - 2^P \) search tree, which is described in the next section.

### 7.5 The \( 2^P - 2^P \) Search Tree

A \( 2^P - 2^P \) tree (integer \( P \geq 3 \)) is a tree in which every vertex which is not the root or a leaf has between \( 2^P - 2 \) and \( 2^P \) sons and every path from the root to a leaf is of the same length. If the tree is not the singleton case (i.e. 2 nodes connected by a single edge) then the root has between 2 and \( 2^P \) sons, otherwise the root has 1 son. A tree may be empty.
A $2^{P-2} - 2^P$ search tree is a $2^{P-2} - 2^P$ tree where associated with a node $x$ is a key value $key(x)$ such that all the keys at the same level in the tree are distinct. In addition, for all nodes (except the root):

$$key(x) \geq key(father(x))$$

where the function $father(x)$ returns the father of node $x$ in the tree. Also, if $x$ and $y$ are any two nodes then:

$$(height(father(x)) = height(y)) \land (key(father(x)) < key(y)) \Rightarrow key(x) < key(y)$$

where the function $height(x)$ returns the height of node $x$ in the tree.

Data items are represented by the key values of the leaf nodes. The searching operation for a tree is again the normal B+ tree search operation [Comer, 1979] and the insert and delete operations follow the top-down node-splitting scheme. The insertion transformation is applied when an insertion operation encounters a node with $2^P$ sons, other than the root. The node is split to form two nodes each with $2^{P-1}$ sons, as depicted in Figure 7.5. This transformation ensures that any future node splitting does not cause upward propagation in the tree structure.

![Figure 7.5 The general insertion transformation for a $2^{P-2} - 2^P$ tree](image)

When a deletion operation encounters a node with $2^{P-2}$ sons, other than the root, one of the two general deletion transformations is applied. If the neighbouring node has less than or equal to $2^{P-1}$ sons then the transformation depicted in Figure 7.6a is applied, otherwise the transformation of Figure 7.6b is used. (Note, that the neighbour relationship used in the deletion algorithms relates a node to its right brother in the sub-tree or in the case of the rightmost node, to its left brother.)
Chapter 7 Parallel Data Structures

Figure 7.6a The general deletion transformation I (the neighbouring node having less than or equal to $2^{p-1}$ sons)

Figure 7.6b The general deletion transformation II (the neighbouring node having more than $2^{p-1}$ sons)

When the transformations are applied to a root node, the insertion transformation converts a root node with $2^p$ descendants into a double $2^{p-1}$ node configuration and a new root node, increasing the height of the tree. The deletion transformation I converts a root node with 2 descendants into a new root node formed by the merging of the root’s offspring, reducing the height of the tree. The proofs that these transformations preserve the properties of the $2^{p-2}-2^p$ tree follow the style given in [Carey, 1982].

Since the insertion transformation involves splitting a $2^p$-node into two $2^{p-1}$-nodes it will maintain the $2^{p-2}-2^p$ tree structure correctly as long as the father of a $2^p$-node it splits is not also a $2^p$-node. Therefore, to prove correctness it is sufficient to show the following:

Claim:
Suppose the tree segment of Figure 7.7 exists at the beginning of an insert operation, with the insertion path as indicated. By the time node X receives the Insert message, it will no longer be a $2^p$-node.
Figure 7.7 The unacceptable $2^P - 2^P$ tree configuration for insertion

**Proof:**
Noting that the insertion transformation has already been applied (if it was applicable) to the father of node $X$, we proceed by induction on the depth of the tree. For the basis, consider the insertion transformation when node $X$ is the root of the tree. The transformation applied will already have converted the $2^P$-node $X$ into a legal configuration involving two $2^{P-1}$-nodes and a new root node, increasing the tree height by one in the process. Hence, by the time $X$ receives its Insert message it will no longer be a $2^P$-node.

For the inductive step, assume that the claim holds for the first $k$ levels of the tree, $k \geq 1$, and consider a node at level $k+1$. By the inductive hypothesis its father must be of degree $2^P - 1$ or less, as the transformation applied at its own father guarantees this. Thus, even if the node is a $2^P$-node, the troublesome configuration cannot arise. Therefore, the top-down insertion algorithm preserves the tree structure.

The only deletion transformation that could possibly produce an incorrect $2^P - 2^P$ tree structure is transformation I. Therefore, to prove correctness it is sufficient to shown the following:

**Claim:**
Suppose one of the segments of Figure 7.8 exists at the beginning of a deletion operation, with the deletion path as indicated. By the time node $X$ receives the Delete message it will no longer be a $2^P$-node.

Figure 7.8 The unacceptable $2^P - 2^P$ tree configurations for deletion
Proof:
Noting that a deletion transformation has already been applied (if one was applicable) to the father of node X, we proceed by induction on the depth of the tree. For the basis, consider transformation I when X is the root node of the tree. The transformation will merge X's two sons into a new root, reducing the height of the tree by one in the process.

For the inductive step, assume that the claim holds for the first k levels of the tree, k≥1, and consider node X at level k+1. Assume that the claim does not hold for this node. Suppose X has a father XF at level k. There are only two possible cases to consider:

i) XF is not the root and has 2^{P-2} sons or XF is the root and has 2 sons. By the inductive hypothesis, the neighbour of X must have more than 2^{P-1} sons of its own. However, were this true, then transformation II would have been applied when the algorithm reached level k and X would have been transformed before it received its Delete message. Thus, this case is impossible;

ii) XF is not the root and has more than 2^{P-2} sons or XF is the root and has more than 2 sons. In this case, either of the transformations I or II would have been applied when the algorithm reached level k, and X would then be transformed into a node with more than 2^{P-2} sons before it received its Delete message. Thus, this case is impossible.

Since both these cases have led to contradiction, the claim must be true. Therefore, the top-down deletion algorithm preserves the tree structure.

It is possible to place bounds upon the maximum size of the data structure, n, which may be stored in a 2^{P-2}-2^P search tree of H layers. A layer contains the two sets of nodes from adjacent levels in the tree together with the associated edges and key values. In such a case:

\[ 2^{H(P-2)} ≤ n ≤ 2^{HP} \]  

Therefore, for a tree of m data items, the maximum number of layers H required in the tree is given by:

\[ H = \frac{\log_2 n}{P-2} \]  

A further property of the 2^{P-2}-2^P tree is worth noting at this point. The original insertion transformation for the 2-3-4 tree had a direct inverse in the deletion transformation I. Oscillations occurred when insertions and deletions were applied in succession resulting in transformations and their inverses being applied to the tree structure. The behaviour of the structure under these conditions increased the processing time for a query and placed a
strain upon the throughput of the system. A stabilising effect occurs in the $2^{P-2} \cdot 2^P$ tree since the insertion transformation leaves each node (which is not the root) with $2^{P-1}$ sons. Therefore, $2^{P-2}$ sons must be removed from such a node before a deletion transformation may be applied to it (since $P \geq 3$, $2^{P-2} \geq 2$).

In the next chapter a formal specification for the $2^{P-2} \cdot 2^P$ search tree is given and this demonstrates the suitability of the structure for parallel realisation.


Chapter 8  2P^-2-2P Search Trees

The previous chapter showed that a pipeline implementation of a search tree is an efficient means of providing parallel data structures for distributed memory architectures and a 2P^-2-2P search tree offers a number of advantages. A formal specification of the 2P^-2-2P search tree is developed in this chapter. Firstly, the invariants associated with the search tree structure are introduced. Then a semaphore is specified, instances of which are then combined to form a pipeline. This allows concurrent composition to be demonstrated. The operations upon a single layer of the search tree are then specified and the specifications are combined to give a search tree structure with parallel pipeline behaviour.

8.1 Parallel 2P^-2-2P Search Trees

Following on from the definitions given in the previous chapter, the components and invariants associated with a complete 2P^-2-2P search tree are shown in Figure 8.1. Refinements are made to the specification of a rooted tree (TREE) to give a balanced rooted tree (BALANCED_TREE), followed by a 2P^-2-2P tree (2P_TREE) and finally the 2P^-2-2P search tree (SEARCH_TREE). A rooted tree is a connected acyclic graph with a designated vertex (or node) termed the root (the single member of the set root, if one exists), there being a unique path from the root to every other node. The tree may be empty. This structure is specified as a set of nodes (node) formed by the union of the disjoint sets branch and leaf. The partial function father maps a node onto its father in the tree and the relation son relates a node to its sons in the tree. The domain of father is every node except the root since every node except the root has a father. The domain of son is the set branch. The set leaf contains all the nodes without sons, the leaf or terminal nodes of the tree. The inverse of the relation son is the partial function father. Finally, the non-reflexive transitive closure of the relation son intersected with the identity function is empty, giving an acyclic graph.

A balanced rooted tree is a rooted tree in which the lengths of the paths from the root to any two leaf nodes are equal. The specification of such a structure is formed by introducing the balancing condition into the rooted tree definition through refinement. The recursive function height maps a node onto its height in the tree and the predicate height {leaf} = {0} ensures that the tree is balanced.
A $2^P-2^P$ tree is a balanced rooted tree in which every node which is not the root or a leaf has between $2^P-2$ and $2^P$ sons. If the tree is not the singleton case (i.e., 2 nodes connected by a single arc) then the root has between 2 and $2^P$ sons, otherwise the root has 1 son. A $2^P-2^P$ search tree is formed by associating key values with nodes of the tree through the function $key$. Data items are represented by the key values of the leaf nodes.

![Diagram](image)

**Figure 8.1** The subclass `search_tree`

The subclass `search_tree` contains the invariant properties of the $2^P-2^P$ search tree structure. In order to capture the parallel behaviour of the structure a pipeline specification is now required.
A class LOCK is specified in Figure 8.2, which represents a boolean semaphore. An object of class LOCK may be in one of two operating states, idle or busy, shown in Figure 8.3.

```
LOCK(b)
    b, busy: BOOL
    busy' = b

LOCK.Flip(b2)
    b2: (~busy)
    busy' = b2
```

**Figure 8.2 The class GATE**

![State Diagram](image)

**Figure 8.3 The state diagram for an object of class LOCK**

An object of class LOCK is instantiated with a parameter b, which determines whether the LOCK is initially idle or busy. The event LOCK.Flip changes the operating state of the LOCK from idle to busy or vice versa provided that the parameter b2 has the value of the new state (false for idle or true for busy).

A simple bounded counter is now introduced, shown in Figure 8.4, which will be used to index stages in a pipeline.

```
LEVEL(L)
    L, H: NAT
    H ∈ {0..L}
    H' = 0

LEVEL.Incr
    H < L
    H' = H+1

LEVEL.Decr
    H > 0
    H' = H-1
```

**Figure 8.4 A simple bounded counter**
A pipeline can now be specified, shown in Figure 8.5, which is a refinement of LEVEL. LOCK objects are instantiated by the event PIPE.Grow and removed from the PIPE by PIPE.Shrink and a LOCK object is associated with every level of the PIPE.

**Figure 8.5 The class PIPE**

The pipeline is initially empty and only the first of the PIPE.Grow event is applicable. The operation Grow causes an additional LOCK to be added to the front of the pipeline. The first PIPE.Grow event is applied to the empty pipeline and generates a new instance of LOCK leaving it in the busy state. The second PIPE.Grow event adds a new instance of LOCK to an existing pipeline and leaves the new instance in the idle state. The LOCK which was at the front of the pipeline prior to the application of the event goes from being idle to busy. The Shrink operation causes a LOCK to be removed from the front of the pipeline. The first PIPE.Shrink event is applied to the pipeline consisting of a single instance of LOCK. The second PIPE.Shrink event is applicable only when the first two gates in the pipeline are idle.
Chapter 8

Under usual operation only the three events Enter, Down and Leave are applied to a PIPE object. Part of the decision tree for the behaviour of PIPE under these conditions is shown in Figure 8.6.

Figure 8.6 The decision tree for the behaviour of PIPE

The Enter operation causes stage[H] to change from the idle state to the busy state after which only a Down(H) operation may be applied, causing the stage to return to the idle state and stage[H-1] to become busy. At this point two operation are applicable, Enter and Down(H-1). Following the second application of Enter, both stage[H] and stage[H-1] are busy. Therefore, Down(H-1) must be applied before Down(H), which precedes any further application of Enter.

94
In general:

- following a Enter operation Down (H) must be applied before Enter is again applicable;
- following a Down (h) operation Down (h-1) must be applied before Down (h) is again applicable (for H≤h≤2);
- following a Down (2) operation Leave must be applied before Down (2) is again applicable.

This behaviour captures the "no overtaking" property of a pipeline. A further property is worth noting at this point. The object-oriented set-theoretic specification technique introduces conventions for concurrent composition [Schuman, 1989]. From these conventions, the following concurrent composition of two events are possible here:

- Enter with Down (h) (for h<H)
- Enter with Leave (for h≥2)
- Down (h₁) with Down (h₂) (for |h₁-h₂|≥2)
- Down (h) with Leave (for h>2)

When the Grow operation is applied to a non-empty pipeline the new instance of LOCK is left idle and stage[H] becomes busy. At this point, Enter, Down (H) or a further Grow operation are applicable. Following an application of Grow to an empty pipe, only Leave may be applied. Similarly, applying Shrink to a pipeline of length two or greater leaves only Down applicable. Finally, following an application of Shrink to a singleton pipe, only Grow may be applied.

It is now possible to specify the operations associated with a single layer of the tree structure. A layer contains the two sets of nodes from adjacent levels in the tree together with the associated edges and key values. The state schema for the class LAYER is given in Figure 8.7. LAYER also contains three sets of ordered pairs, i_set (insert set), d_set (delete set) and s_set (search_set). These are used to control the synchronisation of the class events and are all initially empty. Note that these sets are disjoint (specified in the invariant) and only a single ordered pair may be present in any of them at any one time.
**Figure 8.7** The state schema for the class `layer`

In Figure 8.8 the operations associated with inserting are specified. `ITransform` takes a node `x` and key value `i` and performs an insert transformation (if one is applicable) on the layer. If a transformation takes place then a new node `y` is created and this node together with a new splitting key is returned. Prior to an application of the event all of the synchronisation sets must be empty. Following the event, `i_set` contains the node-key pair to which the `Insert` event should be applied to in this layer.

`Insert` returns the node to which `ITransform` should be applied to the layer below. For the event to be applicable `i_set` must be non-empty (hence `ITransform` must be applied before an application of `Insert`). `Insert` takes the ordered pair in `i_set` `(x,i)` and determines to which of the sons of `x` `ITransform` should be applied.

`Reply` takes the reply from the `ITransform` event applied to the layer below and updates the components to take account of any transformations. `i_set` must again be non-empty.
but in this case is made empty by the event. The two cases of the event distinguish between performing and not performing a transformation.

\[
\text{LAY\text{ER}}.\text{ITransform}(x, i \rightarrow y, j)
\]
\[
x: \text{nodes}
\]
\[
y: \mathbb{N}\setminus(\text{nodes} \cup \text{leaves})
\]
\[
i, j: I
\]
\[
\#\text{sons}(x) = 2^p
\]
\[
i\_\text{set} = \emptyset
\]
\[
d\_\text{set} = \emptyset
\]
\[
s\_\text{set} = \emptyset
\]
\[
y \in \text{nodes}'
\]
\[
\text{keys}'(y) = j
\]
\[
\text{sons}'(x) \cup \text{sons}'(y) = \text{sons}(x)
\]
\[
\#\text{sons}'(x) = \#\text{sons}'(y)
\]
\[
\forall x_1: \text{sons}'(x): y \_1: \text{sons}'(y): \cdot
\]
\[
\text{keys}'(x_1) < \text{keys}'(y_1)
\]
\[
i < j \Rightarrow (x, i) \in i\_\text{set}'
\]
\[
i \geq j \Rightarrow (y, i) \in i\_\text{set}'
\]

\[
\text{LAY\text{ER}}.\text{Insert}(\rightarrow y)
\]
\[
y: \text{leaves}
\]
\[
x: \text{nodes}
\]
\[
i: I
\]
\[
i\_\text{set} = \{(x, i)\}
\]
\[
y \in \text{sons}(x)
\]
\[
\text{keys}(y) \leq i
\]
\[
\forall x_1: \text{sons}(x): x_1 \not\in y \Rightarrow
\]
\[
(\text{keys}(x_1) < \text{keys}(y)) \lor
\]
\[
(\text{keys}(x_1) > i)
\]

\[
\text{LAY\text{ER}}.\text{IR\text{e}ply}(x, i, y, j)
\]
\[
x: \text{leaves}
\]
\[
y: \mathbb{N}\setminus(\text{nodes} \cup \text{leaves})
\]
\[
i, j: I
\]
\[
i\_\text{set} \neq \emptyset
\]
\[
y \in \text{leaves}'
\]
\[
\text{keys}'(y) = j
\]
\[
\text{fathers}'(y) = \text{fathers}(x)
\]
\[
i\_\text{set}' = \emptyset
\]

\[
\text{LAY\text{ER}}.\text{IR\text{e}ply}(x, i, y, j)
\]
\[
x, y: \text{leaves}
\]
\[
i, j: I
\]
\[
i\_\text{set} \neq \emptyset
\]
\[
x = y
\]
\[
i\_\text{set}' = \emptyset
\]

\textbf{Figure 8.8} The \text{ITransform}, \text{Insert} and \text{IReply} events for the class \text{LAY\text{ER}}
In Figures 8.9 and 8.10 the operations associated with deleting are specified in a similar manner to those for inserting.

**Figure 8.9** The `DTransform` events for the class `LAYER`

```
LAYER.DTransform(x, y, i \rightarrow z, j)
x, y, z: nodes
i, j: I
#sons{x} = 2^{p-2}
#sons{y} \leq 2^{p-1} \Rightarrow (z=x) \land
(j = \min\{\text{keys}(x), \text{keys}(y)\})
#sons{y} > 2^{p-1} \Rightarrow (z=y) \land
(j = \text{keys}(\text{sons}{y}))
i_set = \emptyset
d_set = \emptyset
s_set = \emptyset

\begin{align*}
#sons{y} \leq 2^{p-1} & \Rightarrow (y \notin \text{nodes'}) \land (x, i) \in d_set' \\
#sons{y} > 2^{p-1} & \Rightarrow \left| \text{sons'}{x} - \text{sons}{y} \right| \leq 1 \\
\text{sons'}{x} \cup \text{sons'}{y} & = \text{sons}{x} \cup \text{sons}{y} \\
(y \in \text{nodes'}) \land (\text{keys}(x) > \text{keys}(y)) & \Rightarrow
(j = \min\{\text{keys'}{\text{sons'}{x}}\}) \land
((i < j) \Rightarrow (y, i) \in d_set') \land ((i \geq j) \Rightarrow (x, i) \in d_set')
\end{align*}
```

```
In addition to the events for inserting and deleting items, the operations for searching for a particular key value are given in Figure 8.11. The event Start Search sets the s_set to equal the node key pair to which Search should be applied. Search returns the node to which Start Search should next be applied in the layer below and changes the s_set back to being empty.
Chapter 8

Search Trees

2\(^{2P-2^P}\) Search Trees

It is now possible to combine all the previous specifications to model the pipeline implementation of the 2\(^{2P-2^P}\) search tree. In Figure 8.12 the subclass SEARCH.PIPE is introduced as a refinement of SEARCH_TREE, PIPE and LAYER. The component lay references \(h\) instances of LAYER. The invariants of SEARCH.PIPE relate the components of SEARCH_TREE, PIPE and LAYER so that a unique instance of LAYER is associated with each layer of the search tree.

The operations associated with inserting items into the search tree are given in Figure 8.13. APIPE.Grow is applied to begin an insertion. The first schema for this event corresponds to applying an insertion operation to a root node which has \(2^P\) sons. In this case a double \(2^{P-1}\) -node configuration is formed by applying Transform to the root. PIPE.Grow increments the length of the pipeline (and hence the height of the tree) and a new root node is formed. It returns \(h\), the layer of the tree to which the Insert operation should be applied. The second Grow event is applied to a root with less than \(2^P\) sons. In this case no transformation takes place. PIPE.Enter is used to start the insertion operation leaving the LAYER in which the root is stored in the busy state. The third Grow event is applied to the
empty tree and generates a singleton tree consisting of a root and a single leaf node. The key value of the leaf node is made equal to the key value being inserted.

Figure 8.13 The insertion event schemas for the subclass APIPE

The first SEARCH_PIPE.Insert event is applied to successive branch layers of the search tree. LAYER.Insert is applied to a particular level, followed by LAYER.Transform to the level below and finally Transform to the original level. The parameters of these events control this sequencing. The other SEARCH_PIPE.Insert events are applied to the layer holding the leaf nodes and either insert a new leaf node with the key value i or, if i is already associated with a leaf node, do not change the structure.
In Figure 8.14 the operations associated with deleting an item from the tree are specified.

```
SEARCH_PIPE.Shrink(i \rightarrow h)
  PIPE.Shrink
  lay[H-1].DTransform(x,y,i \rightarrow z,j)
  h: {H-1}
  x,y,z: node
  i,j: I
  H\geq2
  son(root) = {x,y}
  z=x
  y \in node'
  root' = {x}
  key'(x) = j

SEARCH_PIPE.Shrink(i \rightarrow h)
  PIPE.Enter
  lay[H-1].DTransform(x,y,i \rightarrow z,j)
  h: {H-1}
  x,y,z: node
  i,j: I
  H\geq2
  son(root) = {x,y}
  z\neq x

SEARCH_PIPE.Shrink(i)
  PIPE.Shrink
  i: I
  x: leaf
  son(root) = {x}
  key(x) = i

SEARCH_PIPE.Shrink(i \rightarrow h)
  PIPE.Enter
  i: I
  h (H)
  x: leaf
  \#son(root) = {x}
  key(x) \neq i

SEARCH_PIPE.Shrink(i \rightarrow h)
  PIPE.Enter
  i: I
  h (H)
  \#son(root) = leaf
  \#leaf > 1
```

Figure 8.14 The deletion event schemas for the subclass SEARCH_PIPE
Chapter 8

The operations associated with a searching are given in Figure 8.15. Start Search is applied to begin a search followed by Search being applied to successive layers. When Search is applied to the bottom most layer (lay[1]) a boolean value is returned, which indicates whether the key value i is associated with a leaf node in the tree.

![Event Schemata](image)

**Figure 8.15** The searching event schemata for the subclass SEARCH PIPE

The events of SEARCH PIPE may be composed concurrently:

- Grow with Insert (h), Delete (h), Search (h) (for h<H);
- Shrink with Insert (h), Delete (h), Search (h) (for h<H);
- Start Search with Insert (h), Delete (h), Search (h) (for h<H);
- Insert (h₁) with Insert (h₂), Delete (h₂), Search (h₂) (for |h₁-h₂|≥2)
- Delete (h₁) with Insert (h₂), Delete (h₂), Search (h₂) (for |h₁-h₂|≥2)
- Search (h₁) with Insert (h₂), Delete (h₂), Search (h₂) (for |h₁-h₂|≥2)

Therefore, if the search tree is implemented on an array of H processors then half the processors may be processing requests at any one time. Thus, the attainable level of concurrency is \( \frac{H}{2} \). Several examples of the \( 2^P-2^P \) search tree have been implemented on an array of transputer processors to demonstrate the flexibility and efficiency of the structure. The results are presented in the next chapter.
CHAPTER 9 IMPLEMENTATION OF PARALLEL DATA STRUCTURES

In the previous chapter the formal specification for the $2^{p-2}-2^p$ search tree was developed. The optimal search tree structure for a given architecture will vary and is largely dependent upon the speeds of the processor and of the communication links between processors. However, these underlying properties of the $2^{p-2}-2^p$ search tree structure remain:

- Efficiency – at most $\log_2 n/(P-2)$ processors are required to store $m$ search keys and as many as $[\log_2 n/(P-2)]/2$ operations may be executing concurrently;
- Flexibility – variations in both the number of processors allocated to the pipeline and the value of $P$ allow the optimal search structure for a given architecture to be determined;
- Performance – improvements in the query throughput may be achieved through parallel execution;
- Stability – the $2^{p-2}-2^p$ structure introduced a hysteresis behaviour when successive insertion and deletion operations are applied.

In this chapter the $2^{p-2}-2^p$ search tree is used to give a parallel implementation of the $\text{BSET}$ class. Various parallel sets are then implemented for a transputer architecture and their throughputs and response times are compared.

9.1 A Parallel Set Data Structure Implementation

A set data structure in which the base type is infinite or very large or where the number of items contained in the base type is very large compared to the size of the data structure, requires an alternative to bit-vector representation [Hoare, 1972]. A tabular representation, storing the data items as entries in an ordered table, permits logarithmic searches to proceed over the table. The $2^{p-2}-2^p$ search tree provides a parallel implementation of this search operation.

A precondition-conservative refinement of the class $\text{BSET}$ is made to give the subclass $\text{TSET}$ (tree set) shown in Figures 9.1a, 9.1b and 9.1c. $\text{TSET}$ inherits the class $\text{SEARCH_PIPE}$ in addition to $\text{BSET}$. From the upper bound placed upon the cardinality of the set, $m$, and equation (2) of Chapter 7, the maximum number of layers in the tree is determined and the parameter $L$ (from $\text{SEARCH_PIPE}$) is equal to this value by the invariant of $\text{TSET}$.
Figure 9.1a The BSET and TSET object classes (state schemas)

In order to reason about the correctness of the search-tree implementation of BSET, now ignoring parallel access by individual user-code segments, the queries and events of TSET are specified so as to sequentialise the parallel behaviour of the search tree.

The member (search) query for the TSET class is specified by stating that SEARCH_PIPE.Start_Search occurs together with SEARCH_PIPE.Search(1) applied to the lowest layer of the tree. All other search operations upon other levels are implied by the pipeline behaviour of the class. The Include and Exclude events are specified in a similar fashion. The event AnyMem applies the delete operation using an existing member of the set as the parameter i.

An adequate extraction is made from TSET to give an implementation of BSET in terms of the search tree. Set queries may then proceed in parallel. The object class BSET was implemented in this fashion on a transputer pipeline.

Figure 9.1b The BSET and TSET object classes (query schemas)
### Implementation Issues

A $2^P-2^P$ search tree may be implemented in a linear pipeline of $O(\log_2 n)$ processing elements. In this section the three operations associated with this algorithm, Search, Insert and Delete, are considered. In a processor pipeline of $H$ layers, processor $P_h$ ($1 \leq h < H$) is an index processor. Processor $P_1$ contains the data layer.

**Figure 9.6c** The `BSET` and `TSET` object classes (event schemas)

```plaintext
**BSET. Include (i)**
- $i: I \backslash S$
- $n < m$
- $i \in S'$
- `BSET.Include (i)`

**TSET. Include (i)**
- `BSET.Include (i)`
- `SEARCH_PIPE.Grow (i -> h)`
- `SEARCH_PIPE.Insert (l)`
- `i\notin\text{key(leaf)} \Rightarrow #\text{leaf}<m`

**BSET. Exclude (i)**
- `i: X`
- `i \notin S'`
- `TSET.Exclude (i)`

**TSET. Exclude (i)**
- `BSET.Exclude (i)`
- `SEARCH_PIPE.Shrink (i \rightarrow h)`
- `SEARCH_PIPE.Delete (l)`
- `\text{key(leaf)} \neq \{i\}`

**BSET. AnyMemb (→ i)**
- `i: S`
- `i \notin S'`
- `TSET.AnyMemb (→ i)`

**TSET. AnyMemb (→ i)**
- `BSET.AnyMemb (→ i)`
- `SEARCH_PIPE.Shrink (i \rightarrow h)`
- `SEARCH_PIPE.Delete (l)`
- `\{i\} \subseteq \text{key(leaf)}`

**TSET. AnyMemb (→ i)**
- `BSET.AnyMemb (→ i)`
- `SEARCH_PIPE.Shrink (i \rightarrow h)`
- `\text{key(leaf)} = \{i\}`

**Figure 9.6c** The `BSET` and `TSET` object classes (event schemas)
The search operation is a pipelined version of normal B+ tree searching [Comer, 1979]. When a processor \( P_h \) receives a search \((x, i)\) message (where \((x, i)\) are the parameters of LAYER.Start_Search in the specification) it does the following:

Case 1: \( P_h \) contains an index layer (it does not store the leaf nodes of the tree)
Using node \( x \) and key value \( i \), the appropriate son, \( y \), of \( x \) is selected. The message search\((y, i)\) is then sent to processor \( P_{h-1} \);

Case 2: \( P_h \) contains the data layer (it stores the leaf nodes of the tree)
If the key value of node \( x \) is \( i \) then true is returned, otherwise false is returned.

When processor \( P_h \) begins an insertion (following an application of \( \text{ITransform} \)) it does the following (Note that processor \( P_h \) already holds the node\(\times\)key pair \((x, i)\) to which insertion operation should be applied):

Case 1: \( P_h \) contains index nodes (it does not store the leaf nodes of the tree)
Using node \( x \) and key value \( i \), the appropriate sons, \( y \) and \( z \), of \( x \) are selected. Next \( P_h \) sends \( \text{ITransform}(y, i) \) to processor \( P_{h-1} \). \( P_{h-1} \) applies the insert transformation if it is applicable and stores the node\(\times\)key pair to which it should apply the insertion operation. It then sends the \( \text{IRply}(z, j) \) to \( P_h \) and begins the insertion. The reply informs \( P_h \) of the transformation (if any) which has taken place and \( P_h \) updates its node and edge details accordingly (Note that it is possible to overlap the processing of a single insertion between processors);

Case 2: \( P_h \) contains data nodes (it stores the leaf nodes of the tree)
If the key value of node \( x \) is not \( i \) a new leaf node is inserted with the key value \( i \).

When processor \( P_h \) begins a deletion (following an application of \( \text{DTransform} \)) it does the following (Note that processor \( P_h \) already holds the node\(\times\)key pair \((x, i)\) to which deletion operation should be applied):

Case 1: \( P_h \) contains index nodes (it does not store the leaf nodes of the tree)
Using node \( x \) and key value \( i \), the appropriate sons, \( y \) and \( z \), of \( x \) are selected. Next \( P_h \) sends \( \text{DTransform}(y, z, i) \) to processor \( P_{h-1} \). \( P_{h-1} \) applies a delete transformation if one is applicable and stores the node\(\times\)key pair to which it should apply the deletion operation. It then sends the \( \text{DReply}(y^1, z^1, j) \) to \( P_h \) and begins the deletion. The reply informs \( P_h \) of the transformation (if any) which has taken place and \( P_h \) updates its node and edge details accordingly (Note that it is possible to overlap the processing of a single deletion between processors);

Case 2: \( P_h \) contains data nodes (it stores the leaf nodes of the tree)
If the key value of node \( x \) is \( i \) then \( x \) is removed from the tree.
9.3 Implementation on a Transputer Architecture

Any building block for a distributed memory multi-processor architecture must offer a fast processor, memory which may be accessed efficiently and communications links for connecting the processors [Hey, 1987]. The Inmos transputer [Inmos, 1986] meets these requirements by providing:

- a fast (50ns), 32 bit RISC processor
- 4K bytes (T8) of on chip RAM and a 32 bit external interface
- 4 bidirectional communication links each running at up to 20M bits/s

Since all these elements are integrated onto the same silicon, interconnection is greatly reduced. This simplifies the construction of modular reconfigurable architectures such as Supernode [Harp, 1987] [Hey, 1988] [Nicole, 1989] (an architecture developed under Esprit Project 1085). The basic single Supernode architecture consists of sixteen T800 worker transputers each with 256Kb static RAM or 4Mb dynamic RAM local memory, a controller transputer, one or two T800 processors acting as disc servers and caches, a link switching network and a number of external links and devices. A complete supercomputer may be formed by combining up to sixty four Supernodes with an appropriate outer switching network, controllers and devices [Nicole, 1989]. The search tree structure examples presented here were developed at Thorn EMI Central Research Laboratories on a Supernode consisting of sixteen T800 worker transputers each with 4Mb dynamic RAM. The communication links between the processors were pre-set at 10M bits/s.

In any multi-processor architecture used for searching structures two critical criteria are the throughput and response time of the system. The throughput is the rate at which queries are executed and the response time is the time delay between sending a query and receiving the reply. The \(2^P-2^P\) search tree structure is highly flexible as it allows variations in the throughput and response time to be achieved by simple changes in the processor architecture or the value of \(P\).

Six tree structures were implemented using a parallel dialect of C [3L, 1988b]:

- six layer 2-3-4 tree
- four layer 2-8 tree (P=3)
- three layer 4-16 tree (P=4)
- two layer 8-32 tree (P=5)
- two layer 16-64 tree (P=6)
- one layer 1024-4096 tree (P=12)

It should be noted that the 2-3-4 tree was not a \(2^P-2^P\) tree but will be assumed to have a value of \(P=2\) in the results given herein. These structures were chosen because the maximum number of key values which could be stored in each was 4096 (except for the 8-32 tree in which could be stored only 1024 key values). The physical limits of the storage
available at each processor prevented tree structures storing more elements being investigated.

Each structure was implemented on a pipeline of transputers and the length of the pipeline was varied for each case. For example, for the six layer 2-3-4 tree pipeline lengths of one, two, three and six processors were used. This allowed the effect of assigning several layers of the tree structure to a single processor to be determined. Measurements were made for one hundred insertion queries to the tree structures using firstly random data keys, generated by the linear congruential method [Knuth, 1969a], and secondly sequentially ordered keys. In practice most key value distributions will lie somewhere between a random and sequentially ordered distribution. The keys were five digit integer values. The measured values of the throughput and response time were normalised for each case. Since the search time for a given tree structure was proportional to \( \log_2 n \), a normalising coefficient for each case was calculated as \( \log_2 n / (\log_2 n_{2.3.4}) \), where \( n \) is the number of elements in the tree prior to the one hundred queries being applied and \( n_{2.3.4} \) is this value for the 2-3-4 tree. Normalisation allowed comparisons between the values of throughput and response time for differing tree structures to be made without the need to consider the number of key values stored in each structure.

To implement generators in the processor pipeline the leaf process maintained a pointer into the data structure for each generator so as to identify the next key value to be yielded. A physical connection to the leaf processor provided an interface through which the processor can access the external predicate functions used by the generators.

To support multiple instances (a \( 2^p \)-2-2\(^p \) forest), refinements were made to the structure through the introduction of an additional process, termed the controller, along with minor changes to the existing processes and three additional operations upon the data structures. The controller process supplied \textsc{newset} and \textsc{nulset} operations which respectively generated and removed instances of the set data structure as specified in Chapter 4. The \textsc{newset} operation took a positive integer parameter, which placed an upper bound upon the cardinality of the new set structure and returned a unique tag for the set, used in all subsequent operations on that set. The \textsc{nulset} operation took the set identifier as a parameter and removed the data structure. The \textsc{newset} operation caused space to be allocated by the pipeline processes and a subsequent \textsc{nulset} operation freed this space. An additional operation for initialising the processes prior to the creation of any set data structures was also provided.

From the parameter supplied to the \textsc{newset} operation and Equation (2) of Chapter 7 the maximum height of the search tree was determined. This allowed the process used as the root to vary according to the maximum size of the tree. The controller process could be
connected to a number of pipeline processes, as indicated in Figure 9.2. A table mapping the set identifier onto the corresponding root process was maintained by the controller process and this minimised the query response time. Where there were more than three connections required the controller ran on multiple processors.

![Diagram showing implementation of parallel data structures](image)

**Figure 9.2** An example implementation of multiple set data structures with generators

In addition, the controller process acted as an interface between the programming environment and the data structure, the actual implementation of the data structure being hidden. In this respect, similar data structure implementations may be used for a variety of programming environments with only minor changes to the object interface. To date, concurrent data structures of this nature have been supported for the languages FORTRAN 77 and C on the Supernode architecture using a parallel C compiler [3L, 1988b].

It is worth noting at this point that the interface provided by the parallel implementation corresponds to that provided by the FORTRAN 77 code given in Chapter 4. The only difference between these two interfaces being that the controller process receives set operations via the message passing extensions provided by the extended sequential language as opposed to the subroutine call used in the purely sequential implementation. This allows the sequential implementation to be replaced by the parallel implementation in distributed memory parallel architectures.
### 9.4 Measured Results

The results for the various search tree are given in Tables 9.1 and 9.2. The throughput is expressed in terms of the number of replies received from the structure per second and the response time in terms of the time lapse between sending a query to the structure and receiving the corresponding reply from the structure.

<table>
<thead>
<tr>
<th>Tree Type</th>
<th>No. Levels</th>
<th>No. Processors</th>
<th>Throughput (replies/s)</th>
<th>Response Time (ms)</th>
<th>Coefficient</th>
<th>Normalised Throughput</th>
<th>Normalised Response Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-3-4</td>
<td>6</td>
<td>1</td>
<td>95.64</td>
<td>30.57</td>
<td>1.00</td>
<td>95.64</td>
<td>30.57</td>
</tr>
<tr>
<td>2-3-4</td>
<td>6</td>
<td>2</td>
<td>178.32</td>
<td>19.55</td>
<td>1.00</td>
<td>178.32</td>
<td>19.55</td>
</tr>
<tr>
<td>2-3-4</td>
<td>6</td>
<td>3</td>
<td>239.81</td>
<td>14.35</td>
<td>1.00</td>
<td>239.81</td>
<td>14.35</td>
</tr>
<tr>
<td>2-3-4</td>
<td>6</td>
<td>6</td>
<td>303.06</td>
<td>11.56</td>
<td>1.00</td>
<td>303.06</td>
<td>11.56</td>
</tr>
<tr>
<td>2-8</td>
<td>4</td>
<td>1</td>
<td>131.58</td>
<td>14.49</td>
<td>1.43</td>
<td>187.76</td>
<td>10.15</td>
</tr>
<tr>
<td>2-8</td>
<td>4</td>
<td>2</td>
<td>228.83</td>
<td>9.55</td>
<td>1.43</td>
<td>326.54</td>
<td>6.69</td>
</tr>
<tr>
<td>2-8</td>
<td>4</td>
<td>4</td>
<td>307.69</td>
<td>7.81</td>
<td>1.43</td>
<td>439.07</td>
<td>5.47</td>
</tr>
<tr>
<td>4-16</td>
<td>3</td>
<td>1</td>
<td>153.37</td>
<td>10.46</td>
<td>1.54</td>
<td>236.80</td>
<td>6.77</td>
</tr>
<tr>
<td>4-16</td>
<td>3</td>
<td>3</td>
<td>286.53</td>
<td>6.93</td>
<td>1.54</td>
<td>442.40</td>
<td>4.49</td>
</tr>
<tr>
<td>8-32</td>
<td>2</td>
<td>1</td>
<td>196.04</td>
<td>4.85</td>
<td>1.30</td>
<td>255.44</td>
<td>3.72</td>
</tr>
<tr>
<td>8-32</td>
<td>2</td>
<td>2</td>
<td>263.30</td>
<td>5.81</td>
<td>1.30</td>
<td>343.08</td>
<td>4.45</td>
</tr>
<tr>
<td>16-64</td>
<td>2</td>
<td>1</td>
<td>145.99</td>
<td>6.70</td>
<td>1.68</td>
<td>245.40</td>
<td>3.99</td>
</tr>
<tr>
<td>16-64</td>
<td>2</td>
<td>2</td>
<td>182.15</td>
<td>9.00</td>
<td>1.68</td>
<td>306.19</td>
<td>5.35</td>
</tr>
<tr>
<td>1024-4096</td>
<td>1</td>
<td>1</td>
<td>7.32</td>
<td>136.53</td>
<td>1.74</td>
<td>12.74</td>
<td>78.47</td>
</tr>
</tbody>
</table>

Table 9.1 The throughputs and response times for the various architectures using random key values

<table>
<thead>
<tr>
<th>Tree Type</th>
<th>No. Levels</th>
<th>No. Processors</th>
<th>Throughput (replies/s)</th>
<th>Response Time (ms)</th>
<th>Coefficient</th>
<th>Normalised Throughput</th>
<th>Normalised Response Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-3-4</td>
<td>6</td>
<td>1</td>
<td>96.60</td>
<td>30.26</td>
<td>1.00</td>
<td>96.60</td>
<td>30.26</td>
</tr>
<tr>
<td>2-3-4</td>
<td>6</td>
<td>2</td>
<td>182.48</td>
<td>19.41</td>
<td>1.00</td>
<td>182.48</td>
<td>19.41</td>
</tr>
<tr>
<td>2-3-4</td>
<td>6</td>
<td>3</td>
<td>247.52</td>
<td>14.20</td>
<td>1.00</td>
<td>247.52</td>
<td>14.20</td>
</tr>
<tr>
<td>2-3-4</td>
<td>6</td>
<td>6</td>
<td>311.53</td>
<td>11.28</td>
<td>1.00</td>
<td>311.53</td>
<td>11.28</td>
</tr>
<tr>
<td>2-8</td>
<td>4</td>
<td>1</td>
<td>134.05</td>
<td>14.06</td>
<td>1.43</td>
<td>191.29</td>
<td>9.85</td>
</tr>
<tr>
<td>2-8</td>
<td>4</td>
<td>2</td>
<td>238.10</td>
<td>9.07</td>
<td>1.43</td>
<td>339.77</td>
<td>6.36</td>
</tr>
<tr>
<td>2-8</td>
<td>4</td>
<td>4</td>
<td>301.20</td>
<td>7.83</td>
<td>1.43</td>
<td>429.81</td>
<td>5.49</td>
</tr>
<tr>
<td>4-16</td>
<td>3</td>
<td>1</td>
<td>174.52</td>
<td>8.80</td>
<td>1.54</td>
<td>269.46</td>
<td>5.70</td>
</tr>
<tr>
<td>4-16</td>
<td>3</td>
<td>3</td>
<td>304.88</td>
<td>5.95</td>
<td>1.54</td>
<td>470.73</td>
<td>3.85</td>
</tr>
<tr>
<td>8-32</td>
<td>2</td>
<td>1</td>
<td>259.84</td>
<td>3.66</td>
<td>1.30</td>
<td>338.57</td>
<td>2.81</td>
</tr>
<tr>
<td>8-32</td>
<td>2</td>
<td>2</td>
<td>375.00</td>
<td>3.42</td>
<td>1.30</td>
<td>488.63</td>
<td>2.62</td>
</tr>
<tr>
<td>16-64</td>
<td>2</td>
<td>1</td>
<td>244.50</td>
<td>3.92</td>
<td>1.68</td>
<td>411.00</td>
<td>2.33</td>
</tr>
<tr>
<td>16-64</td>
<td>2</td>
<td>2</td>
<td>364.96</td>
<td>3.66</td>
<td>1.68</td>
<td>613.30</td>
<td>2.18</td>
</tr>
<tr>
<td>1024-4096</td>
<td>1</td>
<td>1</td>
<td>396.83</td>
<td>2.03</td>
<td>1.74</td>
<td>690.48</td>
<td>1.17</td>
</tr>
</tbody>
</table>

Table 9.2 The throughputs and response times for the various architectures using sequentially ordered key values
The throughputs and response times of the $2^P-2-2^P$ trees for random keys are shown in Figures 9.3a and 9.3b. The value of L for a given search tree was the maximum pipeline length (or number of layers in the tree) used in the implementation of that tree.

The throughput for the single processor case improved with increasing P as far as P=5, the 8-32 tree. This was due to the reduction in context switching between processes on a single processor. The results for the 16-64 tree (P=6) demonstrated the effect of increasing the size of the data structure allocated to each process so that the correspondingly long processing times caused a degradation in throughput. When the pipeline length was increased to H processors an increase in throughput was seen on changing from a 2-3-4 (P=2) to a 2-8 (P=3) tree and then to a 4-16 (P=4) tree. The change to a 8-32 (P=5) and then to a 16-64 (P=6) tree resulted in a decrease in throughput.

The response time followed a similar pattern with improvements being seen for increasing values of P until the 16-64 (P=6) tree was reached. Note that the response time decreased as additional processors were added for the 2-4 (P=2), 2-8 (P=3) and 4-16 (P=4) trees. However, for the 8-32 (P=5) and 16-64 (P=6) trees the response time increased as additional processors were used since very little of the processing for a single query ran concurrently in these cases.

When the 1024-4096 tree with a single processor was considered, the burden of the single computational intensive process was clearly demonstrated by the poor values for throughput and response time, 12.74 replies/s and 78.47 ms respectively.

![Figure 9.3a] The normalised throughput for the search trees with pipeline lengths of one and L processors using random data
Figure 9.3b The normalised response time for the search trees with pipeline lengths of one and L processors using random data.

The results for sequentially ordered data are given in Figures 9.4a and 9.4b. The throughput improves as the value of P (in $2^P - 2P$) increases and the length of the processor pipeline increase. For the single processor case this improvement is almost linear. Significant reductions are seen in the response times, caused by the removal of the need to shift existing key values in any layer when an insertion takes place. New elements are always added to the end of an existing list.

Figure 9.4a The normalised throughput for the search trees with pipeline lengths of one and L processors using sequentially ordered data.
Figure 9.4b The normalised response time for the search trees with pipeline lengths of one and L processors using sequentially ordered data

It is worth considering why there should be, for the 2-3-4, 2-8 and 4-16 trees with random data, such a significant decrease in the response time for the tree architectures as the length of the processor pipeline is increased. For a processor pipeline, increasing the number of processors in the usually results in a degradation in the response time for a single query but an improvement in the throughput for several such queries. This leads to the so called throughput/response time trade-off. However, it was noted earlier in this chapter that it is possible to overlap the processing of a single insertion (or deletion) between processors. When a number of processors execute the same query in this fashion an improvement in the response time occurs.

These results demonstrate the flexibility of the $2^P-2^P$ search structure since both the value of $P$ and the number of processors allocated to the search structure may be varied in order to achieve the desired throughput and response time. For the architectures considered, the 4-16 tree with three processors in the pipeline may offer the best performance for random data. If a single processor was to be used, the 8-32 tree provides the greatest all round performance.
CHAPTER 10 SUMMARY AND CONCLUSION

FORTRAN 77 continues to be heavily used, commercial FORTRAN libraries have a wide distribution and it is still the most popular language for programming parallel systems. In Chapter 1 it was argued that a data abstraction facility should be integrated into existing FORTRAN 77 environments. In particular, the data independence supported would allow parallel implementation of data structures to be considered. The extension of sequential languages such as FORTRAN with new operations that allow concurrency and synchronisation to be expressed remains the most viable means for migrating sequential applications onto distributed memory parallel architectures. Where data is to be shared between processors, data abstraction permits the details of synchronisation, parallel access and consistency to be hidden.

A survey of the introduction of data abstraction into FORTRAN 77 showed that this may be achieved through the use of the encapsulation mechanisms supported by the language. By using the FORTRAN 77 multiple entry point procedure facility together with the SAVE statement to provide static variables within procedures it was possible to encapsulate the implementation of simple data abstractions. The specification and implementation of data abstractions is a non-trivial task and an unfamiliar one to the FORTRAN 77 programmer. For this reason a library of data abstractions was suggested, which might include sets, stacks, queues, abstract arrays and binary relations. It was argued that a state-based technique was most appropriate for the specification of such structures, due to the procedural nature of the abstractions, and the object-oriented set-theoretic specification technique was used throughout this thesis.

In Chapter 4 the specification and implementation of a bounded set was introduced and it was shown that the proof concerns for the implementation and usage of the data structure were separate. In particular, alternative representations were considered without changes to user-code. Multiple instances of the data structure were introduced through refinement of the original specification. The object-oriented specification technique's support for subsequent specialisation of a previously defined class through refinement made it especially appropriate.

The classical relationship between the implementation and specification was defined in Chapter 4, in terms of an abstraction function and a representation invariant. Alternatively, the implementation of a class may be derived directly from the specification by first making a precondition-conservative refinement followed by an adequate extraction. This yields a new class, which is an implementation of the original class. This approach was used in
Chapter 5 to derive representations for the bounded set. Both implementation through refinement and the use of an abstraction function achieved the same results, but the former was shown to capture the intuition required in the latter as a more structured method and made the reasoning steps involved more explicit.

Empirical analysis of FORTRAN 77 loop constructs indicated that set selective and searching generators should be supported so as to provide efficient non-destructive access to the data structures. A set selective generator was specified in Chapter 6 through refinement and, although language restrictions were shown to increase complexity, a satisfactory compromise between language limitations and generator constructs was achieved. It was then possible to introduce binary operations over the set data structure in a procedural style and the generators were shown to reduce program size and eased the proof of program correctness.

Concurrent data structures may be implemented using a server or replication approach. In Chapter 7 it was argued that the server approach allowed the implementation of the data structure to be hidden from external processes and appealed to farming or pipeline processor topology. The significant advantage of a pipeline system is that no redistribution of the data between processors is required so, although maximum throughputs may not be as great as for farming topologies, system availability is unaffected by redistribution and inter-processor communication is straightforward.

A $2^{P-2}-2^P$ search tree (integer $P \geq 3$) was introduced in Chapter 7 with a pipeline of up to $\log_2 m/(P-2)$ processors being used to implement a tree storing $m$ search keys. A top-down node-splitting scheme was shown to preserve the structure during insertion and deletion transformations without the need to re-traverse any portion of the search path. In Chapter 8 a formal specification of the search tree was developed, from which the concurrent composition of the operations and the pipeline behaviour of the structure were demonstrated. A precondition-conservative refinement was then made in Chapter 9 so as to give an implementation of the bounded set structure in terms of a search tree. Various parallel search trees were then implemented on a pipeline of transputer processors. The $2^{P-2}-2^P$ search tree offered advantages over similar proposed schemes because:

- variations in both the value of $P$ and the number of processors allocated to the pipeline allowed the optimal structure for a given architecture to be determined;
- improvements in both query throughput and response time occurred when moving from a sequential to parallel implementation;
- a stabilising (hysteresis) behaviour is introduced for successive insertion and deletion operations.
The introduction of data abstraction into FORTRAN 77 has been achieved without extensions to the language’s syntax and semantics and a library of data classes has been supported. The benefits of abstraction may now be exploited in a programming language designed before the widespread use of abstraction techniques.

In addition, the data independence provided by data abstraction allows a number of alternative implementations to be considered including parallel data structures. The provision of parallel data structures is a requirement for the effective migration of sequential systems onto distributed memory multi-processor architectures. Formally specified data abstractions provide an interface through which parallel data structures may be accessed by sequential code segments with the $2^p - 2^p$ search tree providing a means of realising an efficient representation for parallel access.

10.1 Directions for Further Research

The use of data abstraction methods in FORTRAN 77 should be pursued and this will enable the adequacy of the data structure library to be determined. It is anticipated that this library will be extended as a result of experience in use.

For the sharing of data structures between processors in a distributed memory multiprocessor architecture, all techniques so far proposed suffer from one of two major constraints which inhibit their use in highly parallel systems:

- **Bottleneck constraints** – When the number of processing elements requiring access to the data structure increases, bottlenecks in the access mechanism cause severe performance degradations and make the technique unacceptable for highly parallel systems.

- **Consistency constraints** – Duplicate copies of the data structure are held in different processing elements in an attempt to avoid bottlenecks. However, mutual consistency algorithms are required, which cause performance degradations when the number of duplications is large. Again, this is unacceptable for highly parallel systems.

To address the problem of sharing data in highly parallel distributed memory architectures a new approach is required and this will be the subject of further study. It is proposed that the sharing of data structures in highly parallel distributed memory architectures may best be achieved by a combination of dedicated data structure servers and replication schemes. In such a hybrid system the data structure may be distributed between a number of dedicated server sites each of which encapsulates some part of the structure and acts as manager for that partition. Replication between server sites may then be permitted with an
appropriate mutual consistency algorithm between sites. Each server is likely to consist of a number of dedicated processing elements and should store the data structure internally as a search tree. Concurrent access and/or updates to any individual server site will be possible. The number of processing elements accessing any one server may be restricted so as to avoid the single server bottleneck. However, if this upper bound is too low then a large number of servers will result, increasing the likelihood of the system being constrained due to the maintenance of data consistency. A trade-off is therefore anticipated between the upper bound placed upon the number of processing elements accessing a single server and the volume of communication required between server sites to maintain consistency. Further experimental investigation will be required to determine the appropriate balancing parameters within such an approach.
REFERENCES


References


References


References


References


References


References


References


References


References


References


References


References


References


References


During the course of his Ph.D studies, the author has published the following research papers:


134
The Author's Publications


The abstracts of these papers follow:

The emphasis of data organisation upon the specification, design and implementation of large systems raises the question as to whether data abstraction may be applied to programming languages designed before the widespread use of such techniques. The data abstraction facilities available in FORTRAN 77 are considered and it is shown that encapsulation is possible for simple data structures. An abstract array data structure is specified and further refined to allow the definition of iterators. An example of this structure is implemented in FORTRAN 77. The introduction of a series of abstractions is thus made possible which facilitates structured development and efficient maintenance.

The software maintenance process is presently consuming a large proportion of the software budget of many organisations. Various estimates have ranged as high as 85% of the total software budget being allocated to maintenance. Preventive maintenance techniques have been proposed, which modify systems in order to make them more amenable to later maintenance and therefore attempt to reduce the cost of maintenance. The restructuring of program control flow schemas has been outlined as one method of preventive maintenance with the subsequent development of several commercial restructuring packages. In many cases such tools fail to address the underlying problem which is poorly structured data. The aim here is to go some way towards facing and solving this problem. A technique is proposed which facilitates the re-engineering of the data in software to a more structured equivalent form and a software tool is outlined which will aid in the automation of this process. The result is a structured program both in terms of control flow and data structures together with the production of documentation in a step-wise and
formal manner. This leads to a reduction in the maintenance costs as more maintainable code is produced.

The emphasis of data organisation upon the specification, design and implementation of systems raises the question as to whether data abstraction may be applied to programming languages designed before the widespread use of such techniques. The data abstraction facilities available in FORTRAN 77 are studied and it is shown that encapsulation is possible for a simple data structure. A set data structure is specified and later refined to allow the definition of a simple iterator. The structure is implemented in FORTRAN 77 and the program proof for the use of the iterator in a loop construct is given. Such a facility allows the introduction of a series of abstractions into both new and existing FORTRAN 77 systems.

The re–engineering of program control flow schemas, although a widely advocated preventive maintenance technique, fails to address the underlying problems of software which contains poorly structured data. A technique is proposed which facilitates the retrospective introduction of abstract data types into existing systems and the corresponding software tool to aid this process is presented. The resulting source code is structured in terms of both data and control flow, thereby significantly promoting the ease of maintenance.

During the last five years the concept of software re-engineering has established itself as one of the most promising techniques for supporting the software maintenance activity. However, the most significant attention was, in the first instance, concerned with only control restructuring and more recently with data normalisation. The latter clarifies the static structure of the data and produces their entity life histories (its dynamic evolution). Both of these techniques are really concerned with the tidying of the software and its structure so that maintenance effort can be more easily supported. It does not address the issue of fundamentally poor design. The University of Surrey has recently completed work on both imperative and functional languages, including FORTRAN 77, COBOL 85 and STRAND®; the latter is Artificial Intelligence’s proprietary concurrent processing/functional language. This unique work has produced a uniform model for the fundamental alteration of source code by the retrospective introduction of data abstraction. It is the structure of this model, the Source Code Re-Engineering Reference Model (SCORE/RM), which the paper addresses. The introduction of such a reference model is essential if the re-engineering of software is to form part
of a full CASE environment, as this will permit the individual development of the appropriate tools whilst ensuring compatibility amongst the tools and other CASE systems.

A scheme for maintaining a balanced search tree on a distributed memory parallel architecture is described. A general \(2^P-2P\) (for integer \(P \geq 3\)) search tree is introduced with a linear array of up to \([\log_2 N/(P-2)]\) processors being used to implement such a search tree. As many as \([\log_2 N/(P-2)]/2\) operations can execute concurrently. Insertion and deletion transformations are described and several search trees are demonstrated on an array of transputer processors. Variations in both the number of processors allocated to the array and the value of \(P\) allow the optimal search structure for a given architecture to be determined.

A scheme for maintaining a balanced search tree on a distributed memory parallel architecture is described. A general \(2^{P-2P}\) (for integer \(P \geq 3\)) search tree is introduced with a linear array of up to \([\log_2 N/(P-2)]\) processors being used to implement such a search tree. As many as \([\log_2 N/(P-2)]/2\) operations can execute concurrently and insertion and deletion transformations are described. The \(2^{P-2P}\) search tree structure is shown to provide a means of implementing a set data structure and, through a series of refinements, allows the introduction of generators and multiple instances.

[8] “Concurrent Data Structures”
Several techniques for the storage of large data structures in main memory have been proposed and, although none is optimal in every situation, tree structures have become a commonly adopted algorithm. A scheme for maintaining a balanced search tree on a distributed memory parallel architecture is described. A general \(2^{P-2P}\) (for integer \(P \geq 3\)) search tree is introduced with a linear array of up to \([\log_2 N/(P-2)]\) processors being used to implement such a search tree. As many as \([\log_2 N/(P-2)]/2\) operations can execute concurrently. Several examples of \(2^{P-2P}\) search trees have been implemented on an array of transputer processors. The search structures developed were highly flexible allowing variations in the throughput and response time to be achieved by simple changes to the transputer architecture or the value of \(P\). Applications of these search structures presently in use or under consideration include neural networks, parallel simulation systems, distributed database applications and the migration of sequential systems onto parallel architectures.
The Author's Publications

[9] "A Scheme for Implementing Parallel Search Trees"
A general $2^P - 2^P$ (for integer $P \geq 3$) search tree developed at the University of Surrey is introduced, with a linear array of up to $\log_2 N/P - 2$ processors being used to implement such a tree. As many as $\log_2 N/P - 2$ operations can execute concurrently. Variations in both the number of processors allocated to the array and the value of $P$ allow the optimal search structure for a given architecture to be determined. A formal description of the structure allows reasoning concerning the possible concurrency in the system and the search tree structure is shown to provide a means of implementing a set data structure.

[10] "An Efficient Implementation of Search Trees on an Array of Transputers"
A scheme for maintaining a balanced search tree on a transputer architecture is described. A general $2^P - 2^P$ (for integer $P \geq 3$) search tree developed at the University of Surrey is introduced, which allows tree operations to execute concurrently. Several examples of these search trees have been implemented on the Supernode architecture using an array of transputers. Significant improvements in both query throughput and response time were demonstrated when moving from a sequential to a parallel implementation and the optimal search tree was identified. Applications of these structures include neural networks, parallel simulation systems, distributed database applications and the migration of sequential systems onto parallel architectures.

The process of software re-engineering must incorporate techniques for manipulating software which is imperative, declarative or functional in nature. This generality needs a mechanism for deriving the original requirements of the underlying data structures contained within the source code itself. A reference model is proposed from which it is possible to derive the necessary techniques required to implement such a mechanism. The reference model is programming language independent and as such is ideal for representing the general re-engineering process.

[12] "Data Abstraction, FORTRAN 77 and Concurrency"
The objective of this paper is to demonstrate the introduction of data abstraction into FORTRAN 77 without extensions to the language's syntax or semantics. Abstraction improves the constructability, reliability, understandability and maintainability of software systems as well as easing the application of formal program correctness proofs. In addition, the independence of the representation of the abstractions from their use allows parallel implementations to be considered. This aids in the migration of sequential systems onto parallel multi-processor architectures.
# APPENDIX A GLOSSARY OF MATHEMATICAL NOTATION

This appendix contains a glossary of the mathematical notation used throughout the thesis.

## Logic

Taking $P$ and $Q$ to be propositions...

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{true}$</td>
<td>True</td>
</tr>
<tr>
<td>$\text{false}$</td>
<td>$\neg (\text{true})$</td>
</tr>
<tr>
<td>$\neg P$</td>
<td>Negation, &quot;not $P$&quot;</td>
</tr>
<tr>
<td>$P \land Q$</td>
<td>Conjunction, &quot;$P$ and $Q$&quot;</td>
</tr>
<tr>
<td>$P \lor Q$</td>
<td>Disjunction, &quot;$P$ or $Q$&quot;</td>
</tr>
<tr>
<td>$P \implies Q$</td>
<td>Implication, &quot;$P$ implies $Q$&quot; or &quot;if $P$ then $Q$&quot;</td>
</tr>
<tr>
<td>$P \iff Q$</td>
<td>Equivalence, &quot;$P$ is logically equivalent to $Q$&quot;</td>
</tr>
<tr>
<td>$\forall x \cdot P[x]$</td>
<td>Universal quantification, &quot;$P$ holds for all $x$&quot;</td>
</tr>
<tr>
<td>$\exists x \cdot P[x]$</td>
<td>Existential quantification, &quot;$P$ holds for some $x$&quot;</td>
</tr>
<tr>
<td>$x_1 = x_2$</td>
<td>Equality (of variables)</td>
</tr>
<tr>
<td>$x_1 \neq x_2$</td>
<td>$\neg (x_1 = x_2)$</td>
</tr>
</tbody>
</table>

## Sets

Taking $U$ to be some given sets...

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \in U$</td>
<td>Set membership, &quot;$x$ is an element of $U$&quot;</td>
</tr>
<tr>
<td>$x \notin U$</td>
<td>$\neg (x \in U)$</td>
</tr>
<tr>
<td>$\emptyset[U]$</td>
<td>Powerset, the set of all subsets of $U$</td>
</tr>
<tr>
<td>$\text{set}[U]$</td>
<td>Finite subsets of $U$</td>
</tr>
<tr>
<td>$#U$</td>
<td>Cardinality of $U$ (U must be finite)</td>
</tr>
<tr>
<td>$\emptyset[U]$</td>
<td>The empty subset of $U$, where $[U]$ normally omitted</td>
</tr>
<tr>
<td>${x}$</td>
<td>The singleton subset, containing only the element $x$</td>
</tr>
<tr>
<td>${x_1, x_2, \ldots, x_n}$</td>
<td>The subset containing just those elements $x_1, x_2, \ldots$ and $x_n$</td>
</tr>
</tbody>
</table>
...and taking $U_1$, $U_2$, ..., $U_n$ also to be (given) sets...

$U_1 \times U_2 \times \ldots \times U_n$ Cartesian product, all $n$-tuples such that the $i$th component $x_i \in U_i$

$(x_1, x_2, \ldots, x_n)$ Ordered $n$-tuple, $x_1, x_2, \ldots, x_n \in U_1 \times U_2 \times \ldots \times U_n$

$\{x : U \mid P\}$ Comprehension, set containing exactly those $x \in U$ for which $P$ holds

$\forall x : U \cdot P$ Universal quantification $\equiv \forall x \cdot (x \in U \Rightarrow P)$

$\exists x : U \cdot P$ Existential quantification $\equiv \exists x \cdot (x \in U \land P)$

...and taking $S$ and $T$ to be sets such that $S \in \varnothing[U]$ and $T \in \varnothing[U]$...

$S \subseteq T$ Set inclusion $\equiv \forall x : S \cdot x \in T$

$S \subset T$ Strict set inclusion $\equiv S \subseteq T \land \exists x : T \cdot x \not\in S$

$S = T$ Set equality $\equiv S \subseteq T \land T \subseteq S$

$S \neq T$ $\equiv \neg(S = T)$

$S \cap T$ Set intersection $= \{x : U \mid x \in S \land x \in T\}$

$S \cup T$ Set union $= \{x : S \mid x \in S \lor x \in T\}$

$S \setminus T$ Set difference $= \{x : U \mid x \in S \land x \not\in T\}$

Natural Numbers

The inequality operators $<, >, \leq, \geq$ have their usual meaning

NAT The set of Natural Numbers

POS The set of strictly positive natural numbers, NAT \{0\}

$m..n$ The set of natural numbers from $m$ to $n$, $\{x : \text{NAT} \mid m \leq x \leq n\}$

$\min N$ Minimum of a non-empty set, $\min N \in N \land (\forall x : N \cdot x \geq \min N)$

$\max N$ Maximum of a non-empty set, $\max N \in N \land (\forall x : N \cdot x \leq \max N)$
Relations

Taking \( A, B \) and \( C \) to be sets, with \( a \in A, b \in B, c \in C \),...

\[ A \leftrightarrow B \]

The set of all relations from \( A \) to \( B \) = \( \emptyset[A \times B] \)

a relation is a set of ordered pairs; hence, all operators on sets apply to relations

...and taking \( R_0, R_1 \) and \( R_2 \) to be relations with the following “signatures”...

\[ R_0 : A \leftrightarrow A, R_1 : A \leftrightarrow B, R_2 : B \leftrightarrow C \]

\[ aR_1b \]

\( a \) is related by \( R_1 \) to \( b \) = \( (a,b) \in R_1 \)

\[ R_1^{-1} \]

The converse of relation \( R_1 = \{ (b,a) : B \times A \mid aR_1b \} \)

...and with \( A_1 \subseteq A \) and \( B_1 \subseteq B \),...

\[ \text{im } R_1 \]

The image of subset \( A_1 \) through \( R_1 \): \( \emptyset[A] \rightarrow \emptyset[B] \),

\[ b \in (\text{im } R_1)_{A_1} \Rightarrow \exists a : A_1 \cdot aR_1b \]

\[ \text{cod } R_1 \]

The codomain of relation \( R_1 = (\text{im } R_1)_{A} \)

\[ \text{dom } R_1 \]

The domain of relation \( R_1 = (\text{im } R_1^{-1})_{B} \)

\[ R_1(A_1) \]

The image of set \( A_1 \) through the relation \( R_1 = (\text{im } R_1^{-1})_{A_1} \)

\[ R_1\{a\} \]

The image of set \( \{a\} \) through the relation \( R_1 = (\text{im } R_1)_{\{a\}} \)

\[ R_2 \circ R_1 \]

Relational composition: \( A \leftrightarrow C, a(R_2 \circ R_1)c = \exists b : B \cdot aR_1b \wedge bR_2c \)

\[ R_0^n \]

The relation \( R_0 \) composed with itself \( n \) times

\[ R_0^* \]

Reflexive transitive closure of \( R_0 \)

\[ R_0^+ \]

Non-reflexive transitive closure of \( R_0 \)
Appendix A  Glossary of Mathematical Notation

Relations and Functions

General case:

\[ A \leftrightarrow B \]

The set of all (partial) relations from \( A \) to \( B \), \( \varnothing[A \times B] \)

Special cases:

\[ A \rightarrow B \]

The set of all total relations from \( A \) to \( B \), \( \{R:A\leftrightarrow B \mid \text{dom } R = A\} \)

\[ A \rightarrow B \]

The set of all partial functions from \( A \) to \( B \), \( \{F:A\leftrightarrow B \mid (\forall a:A; b_1,b_2:B \cdot aFb_1 \land aFb_2 \Rightarrow b_1=b_2)\} \)

\[ A \rightarrow B \]

The set of all total functions from \( A \) to \( B \), \( (A\rightarrow B) \cap (A\leftrightarrow B) \)

id\[A\]

The "identity function" on \( A \): \( A \rightarrow A \)

\[ a_1 \text{ id } a_2 \equiv a_1 = a_2 \]

A function is a relation, with the property that for each element in its domain there is a unique element related to it. All the operators defined on relations and sets apply also to functions. (Note, the union of two functions is not always a function).

Sequences

Taking \( A \) to be some set...

\[ \text{seq } [A] \]

The set of sequences whose elements are drawn from \( A \), \( \bigsqcup_{n:\text{NAT}} (1..n)\rightarrow A \)

\( \emptyset \)

The empty sequence

\( \langle a \rangle \)

The singleton sequence, consisting of just one element \( a \)

\( \langle a_1,a_2,\ldots,a_n \rangle \)

The sequence whose consecutive elements are \( a_1,a_2,\ldots,a_n \)

\( S_1 \land S_2 \)

Concatenation of \( S_1 \) and \( S_2 \)

A sequence is a finite function whose domain is \( 1..n \), where \( n \) is the length of the sequence. All operators on relations, functions and sets apply to sequences.
Appendix B  Proof of Consistency for the Set Specification

This appendix contains the proof that the specification of the bset object class, shown in Figure B.1, is consistent.

Figure B.1 The specification of the bounded set

An object class specification is consistent if and only if:

1. The class is consistent
   There is a state which satisfies the state invariant;

2. The initialisation is effective
   For any component values which satisfy the state invariant, there exists a possible initialisation state;

3. Each event is applicable
   There is a state which satisfies the derived precondition of the event;
Appendix B

Proof of Consistency for the Set Specification

(4) Each event is effective

For any state and parameter values which satisfy the derived precondition of the event, there exists a possible new state which satisfies the derived postcondition of the event.

The proof of consistency for the specification of the \texttt{Bset} object class follows:

(1) Proof of consistency of the class \texttt{Set}

\textit{Claim}:
\[ \exists m, n : \text{NAT}; \ S : \text{set}[I] \cdot n = \#S \land n \leq m \]

\textit{Proof}:
let \( m = 0, n = 0 \) and \( S = \emptyset \)
therefore \( n = \#S \land n \leq m \)
therefore \( \exists m, n : \text{NAT}; \ \text{cont} : \text{set}[I] \cdot n = \#S \land n \leq m \)

(2) Effectiveness of the initialisation

\textit{Claim}:
\[ \forall m : \text{NAT} \cdot (\exists n' : \text{NAT}; \ S' : \text{set}[I] \cdot n' = \#S \land n' \leq m \land S' = \emptyset) \]

\textit{Proof}:
let \( S' = \emptyset \) and \( n' = 0 \) (\( n' = \#S' \))
given \( n' = \#S' \Rightarrow n' \leq m \)
therefore \( n' = \#S \land n' \leq m \land S' = \emptyset \)
therefore \( \exists n' : \text{NAT}; \ S' : \text{set}[I] \cdot n = \#S \land n \leq m \land S' = \emptyset \)
therefore \( \forall m : \text{NAT} \cdot (\exists n' : \text{NAT}; \ S' : \text{set}[I] \cdot n = \#S \land n \leq m \land S' = \emptyset) \)

(3.1) Applicability of the include event

\textit{Claim}: (case analysis on each schema)

Case 1: \( \exists m, n : \text{NAT}; \ S : \text{set}[I]; \ i : I \cdot (i \notin S \land n = \#S \land n < m) \)

Case 2: \( \exists m, n : \text{NAT}; \ S : \text{set}[I]; \ i : I \cdot (i \in S \land n = \#S \land n \leq m) \)

\textit{Proof}:
Case 1 let \( m = 1, n = 0 \) and \( S = \emptyset \)
therefore \( i \notin S \land n = \#S \land n < m \)
therefore \( \exists m, n : \text{NAT}; \ S : \text{set}[I]; \ i : I \cdot (i \notin S \land n = \#S \land n < m) \)

Case 2 let \( m = 1, n = 1 \) and \( S = \{1\} \)
therefore \( i \in S \land n = \#S \land n \leq m \)
therefore \( \exists m, n : \text{NAT}; \ S : \text{set}[I]; \ i : I \cdot (i \in S \land n = \#S \land n \leq m) \)
(4.1) Effectiveness of the Include event

Claim: (case analysis on each schema)

Case 1: \( \forall m, n : \text{NAT}; \ S : \text{set}[\mathcal{X}]; \ i : \mathcal{X} \rightarrow (i \notin S \land n=\#S \land n<m) \Rightarrow (\exists n' : \text{NAT}; \ S' : \text{set}[\mathcal{X}] \cdot n'=\#S' \land n' \leq m \land i \notin S') \)

Case 2: \( \forall m, n : \text{NAT}; \ S : \text{set}[\mathcal{X}]; \ i : \mathcal{X} \rightarrow (i \in S \land n=\#S \land n \leq m) \Rightarrow (\exists n' : \text{NAT}; \ S' : \text{set}[\mathcal{X}] \cdot n'=\#S' \land n' \leq m \land i \notin S') \)

Proof:

Case 1 assume \( i \notin S \land n=\#S \land n<m \)
let \( n'=1 \) and \( S' = \{i\} \)
therefore \( n'=\#S' \land i \in S' \)
therefore \( \exists n' : \text{NAT}; \ S' : \text{set}[\mathcal{X}] \cdot n'=\#S' \land i \in S' \)
since \( (i \notin S \land n=\#S \land n \leq m \land \exists n' : \text{NAT}; \ S' : \text{set}[\mathcal{X}] \cdot n'=\#S' \land i \in S') \Rightarrow (\exists n' : \text{NAT}; \ S' : \text{set}[\mathcal{X}] \cdot n'=\#S' \land n' \leq m \land i \in S') \) (historical inference)
therefore \( \forall m, n : \text{NAT}; \ S : \text{set}[\mathcal{X}]; \ i : \mathcal{X} \rightarrow (i \notin S \land n=\#S \land n < m) \Rightarrow (\exists n' : \text{NAT}; \ S' : \text{set}[\mathcal{X}] \cdot n'=\#S' \land n' \leq m \land i \notin S') \)

Case 2 assume \( i \in S \land n=\#S \land n \leq m \)
let \( n'=1 \) and \( S' = \{i\} \)
therefore \( n'=\#S' \land i \in S' \)
therefore \( \exists n' : \text{NAT}; \ S' : \text{set}[\mathcal{X}] \cdot n'=\#S' \land i \in S' \)
since \( (i \in S \land n=\#S \land n \leq m \land \exists n' : \text{NAT}; \ S' : \text{set}[\mathcal{X}] \cdot n'=\#S' \land i \in S') \Rightarrow (\exists n' : \text{NAT}; \ S' : \text{set}[\mathcal{X}] \cdot n'=\#S' \land n' \leq m \land i \in S') \)
therefore \( \forall m, n : \text{NAT}; \ S : \text{set}[\mathcal{X}]; \ i : \mathcal{X} \rightarrow (i \in S \land n=\#S \land n \leq m) \Rightarrow (\exists n' : \text{NAT}; \ S' : \text{set}[\mathcal{X}] \cdot n'=\#S' \land n' \leq m \land i \in S') \)

(3.2) Applicability of the Exclude event

Claim:
\( \exists m, n : \text{NAT}; \ S : \text{set}[\mathcal{X}]; \ i : \mathcal{X} \rightarrow (n=\#S \land n \leq m) \)

Proof:

let \( m=0, n=0 \) and \( S=\emptyset \)
therefore \( n=\#S \land n \leq m \)
therefore \( \exists m, n : \text{NAT}; \ S : \text{set}[\mathcal{X}]; \ i : \mathcal{X} \rightarrow (n=\#S \land n \leq m) \)

(4.2) Effectiveness of the Exclude event

Claim:
\( \forall m, n : \text{NAT}; \ S : \text{set}[\mathcal{X}]; \ i : \mathcal{X} \rightarrow (n=\#S \land n \leq m) \Rightarrow (\exists n' : \text{NAT}; \ S' : \text{set}[\mathcal{X}] \cdot n'=\#S' \land n' \leq m \land i \notin S') \)

Proof:

assume \( n=\#S \land n \leq m \)
Appendix B
Proof of Consistency for the Set Specification

let $n' = 0$ and $S' = \emptyset$

therefore $n' = \# S' \land i \notin S'$

therefore $\exists n': \text{NAT}; \ S': \text{set}[\mathcal{I}] \cdot n' = \# S' \land i \notin S'$

since $(n = \# S \land n \leq m \land \exists n': \text{NAT}; \ S': \text{set}[\mathcal{I}] \cdot n' = \# S' \land i \notin S') =>$

$(\exists n': \text{NAT}; \ S': \text{set}[\mathcal{I}] \cdot n' = \# S' \land n' \leq m \land i \notin S')$ (historical inference)

therefore

$\forall m, n: \text{NAT}; \ S: \text{set}[\mathcal{I}]; \ i: \mathcal{I} \cdot (n = \# S \land n \leq m \land i \in S) \Rightarrow (\exists n': \text{NAT}; \ S': \text{set}[\mathcal{I}] \cdot n' = \# S' \land n' \leq m \land i \notin S')$

(3.3) Applicability of the AnyMemb event

Claim:

$\exists m, n: \text{NAT}; \ S: \text{set}[\mathcal{I}]; \ i: \mathcal{I} \cdot (n = \# S \land n \leq m \land i \in S)$

Proof:

let $m=1, n=1$ and $S=\{i\}$

therefore $n = \# S \land n \leq m \land i \in S$

therefore $\exists n, m: \text{NAT}; \ S: \text{set}[\mathcal{I}]; \ i: \mathcal{I} \cdot (n = \# S \land n \leq m \land i \in S)$

(4.3) Effectiveness of the AnyMemb event

Claim:

$\forall m, n: \text{NAT}; \ S: \text{set}[\mathcal{I}]; \ i: \mathcal{I} \cdot (n = \# S \land n \leq m \land i \in S) \Rightarrow (\exists n': \text{NAT}; \ S': \text{set}[\mathcal{I}] \cdot n' = \# S' \land n' \leq m \land i \notin S')$

Proof:

assume $n = \# S \land n \leq m \land i \in S$

let $n' = 0$ and $S' = \emptyset$

therefore $n' = \# S' \land i \notin S'$

therefore $\exists n': \text{NAT}; \ S': \text{set}[\mathcal{I}] \cdot n' = \# S' \land i \notin S'$

since $(n = \# S \land n \leq m \land i \in S \land \exists n': \text{NAT}; \ S': \text{set}[\mathcal{I}] \cdot n' = \# S' \land i \notin S') =>$

$(\exists n': \text{NAT}; \ S': \text{set}[\mathcal{I}] \cdot n' = \# S' \land n' \leq m \land i \notin S')$ (historical inference)

therefore

$\forall m, n: \text{NAT}; \ S: \text{set}[\mathcal{I}]; \ i: \mathcal{I} \cdot (n = \# S \land n \leq m \land i \in S) \Rightarrow (\exists n': \text{NAT}; \ S': \text{set}[\mathcal{I}] \cdot n' = \# S' \land n' \leq m \land i \notin S')$

(3.4) Applicability of the empty event

Claim:

$\exists m, n: \text{NAT}; \ S: \text{set}[\mathcal{I}]; \ b: \text{BOOL} \cdot (n = \# S \land n \leq m \land b=(S=\emptyset))$

Proof:

let $m=0, n=0, S=\emptyset$ and $b=true$

therefore $n = \# S \land n \leq m \land b=(S=\emptyset)$

therefore $\exists m, n: \text{NAT}; \ S: \text{set}[\mathcal{I}]; \ b: \text{BOOL} \cdot (n = \# S \land n \leq m \land b=(S=\emptyset))$
Appendix B  Proof of Consistency for the Set Specification

(4.4) Effectiveness of the empty event

Claim:
\[ \forall m, n : \text{NAT}; S : \text{set} \{X\}; b : \text{BOOL} \cdot (n \neq S \land n \leq m \land b = (S = \emptyset)) \Rightarrow (\exists n' : \text{NAT}; S' : \text{set} \{X\} \cdot \text{true}) \]

Proof: (the event postcondition is true)
therefore \[ \forall m, n : \text{NAT}; S : \text{set} \{X\}; b : \text{BOOL} \cdot (n \neq S \land n \leq m \land b = (S = \emptyset)) \Rightarrow (\exists n' : \text{NAT}; S' : \text{set} \{X\} \cdot \text{true}) \]

(3.5) Applicability of the card event

Claim:
\[ \exists m, n : \text{NAT}; S : \text{set} \{X\}; k : \{n\} \cdot (n \neq S \land n \leq m) \]

Proof:
let \( m = 0, n = 0, S = \emptyset \) and \( k = 0 \)
therefore \[ n \neq S \land n \leq m \land k = n \]
therefore \[ \exists m, n : \text{NAT}; S : \text{set} \{X\}; k : \{n\} \cdot (n \neq S \land n \leq m) \]

(4.5) Effectiveness of the card event

Claim:
\[ \forall m, n : \text{NAT}; S : \text{set} \{X\}; k : \{n\} \cdot (n \neq S \land n \leq m) \Rightarrow (\exists n' : \text{NAT}; S' : \text{set} \{X\} \cdot \text{true}) \]

Proof: (the event postcondition is true)
therefore \[ \forall m, n : \text{NAT}; S : \text{set} \{X\}; k : \{n\} \cdot (n \neq S \land n \leq m) \Rightarrow (\exists n' : \text{NAT}; S' : \text{set} \{X\} \cdot \text{true}) \]

(3.6) Applicability of the member event

Claim:
\[ \exists m, n : \text{NAT}; S : \text{set} \{X\}; i : X; b : \text{BOOL} \cdot (n \neq S \land n \leq m \land b = (i \in S)) \]

Proof:
let \( m = 0, n = 0, S = \emptyset \) and \( b = \text{false} \)
therefore \[ n \neq S \land n \leq m \land b = (i \in S) \]
therefore \[ \exists m, n : \text{NAT}; S : \text{set} \{X\}; i : X; b : \text{BOOL} \cdot (n \neq S \land n \leq m \land b = (i \in S)) \]

(4.6) Effectiveness of the member event

Claim:
\[ \forall m, n : \text{NAT}; S : \text{set} \{X\}; i : X; b : \text{BOOL} \cdot (n \neq S \land n \leq m \land b = (i \in S)) \Rightarrow (\exists n' : \text{NAT}; S' : \text{set} \{X\} \cdot \text{true}) \]

Proof: (the event postcondition is true)
therefore \[ \forall m, n : \text{NAT}; S : \text{set} \{X\}; i : X; b : \text{BOOL} \cdot (n \neq S \land n \leq m \land b = (i \in S)) \Rightarrow (\exists n' : \text{NAT}; S' : \text{set} \{X\} \cdot \text{true}) \]

Therefore, the bset object class is consistent.
APPENDIX C VERIFICATION FOR THE SET DATA STRUCTURE

This appendix contains the proof that the implementation of the BSet object class satisfies the specification of the class. For the set data structure the abstraction function, $\mathcal{A}_{(A,M,N)}$, is defined as:

$$\mathcal{A}_{(A,M,N)} : X \times X \times X \rightarrow X \times X \times X \cdot (S = \{ A(i) | 1 \leq i \leq N \}) \land (m = M) \land (n = N)$$

The representation invariant $(R_j)$ includes the invariant from the object class specification and is:

$$(\forall i, j : 1 \ldots N \cdot i \neq j \Rightarrow A(i) \neq A(j))$$

$$(0 \leq N) \land (N \leq M)$$

For verification it is necessary to demonstrate that:

1. For the implementation of the state schema – providing the preconditions upon the parameter values are satisfied on entry, the initialisation predicate and representation invariant are satisfied on exit;

2. For the implementation of the events – providing the event preconditions and representation invariant are satisfied on entry, the postconditions and representation invariant are satisfied on exit. (The event preconditions on the return values are included in the postcondition for verification).

Due to the rule of historical inference used in the specification, the postconditions for an event is the conjunction of the postcondition given in the specification with the central set for the event [Schuman, 1987] [Schuman, 1989].
Appendix C
Verification for the Set Data Structure

(1) Implementation of the State/Schema

```plaintext
1 SUBROUTINE MSET
2 IMPLICIT LOGICAL (B)
3 SAVE
4 PARAMETER (M=1000)
5 INTEGER A(1:M)
6 INTEGER N
7 LOGICAL BF
8 N=0
9 RETURN
```

The initialisation predicate is \( S' = \emptyset \)

Applying the inverse of the abstraction function to this gives:

\[ (N' = 0) \] 

**Claim:**

true \( (\text{BSET}) \ A^{-1}(S' = \emptyset) \land R_1 \)

where BSET symbolises the implementation of the state schema

**Proof:**

Augmenting the implementation with assertions (shown in the \{ \} brackets above) gives:

Line 8 since \( N' = 0 \Rightarrow \forall i, j: 1..N' \land i \neq j \Rightarrow A(i) \neq A(j) \)

also \( N' = 0 \land M = 1000 \Rightarrow (0 \leq N) \land (N \leq M) \)

therefore \( N' = 0 \land R_1 \)

(2.1) Implementation of the Include event

```plaintext
1 ENTRY INCLUDE\( (i) \)
2 \{ R_1 \}
3 J=0
4 WHILE \((J.LT.N).AND.(.NOT.BF)\) DO
5 J=J+1
6 BF=A(J).EQ.I
7 END WHILE
8 IF (N.GE.M) THEN
9 CALL EXCEPT
10 RETURN
11 END IF
12 BSET. Include\( (i) \)
13 N=N+1
14 A(N)=I
15 RETURN
```

The precondition is \( i \in S \lor (i \notin S \land n < m) \)
Appendix C Verification for the Set Data Structure

The postcondition is \( i \in S' \)

The postcondition for the event (including the central set) is \( S' = S \cup \{i\} \)

Applying the inverse of the abstraction function to these gives:

\[
A^{-1}(i \in S) = \exists j: 1..N \cdot A(j) = i
\]

\[
A^{-1}(i \in S \land n < m) = (\forall j: 1..N \cdot A(j) \neq i) \land n < m
\]

\[
A^{-1}(S' = S \cup \{i\}) = (\forall j: 1..N \cdot (\exists k: 1..N' \cdot A(j) = A'(k))) \land
\]

\[
(\exists j: 1..N' \cdot A'(j) = i) \land
\]

\[
(\forall j: 1..N' \cdot (A'(j) = i \lor (\exists k: 1..N \cdot A'(j) = A(k))))
\]

**Claim**: 

\[
(A^{-1}(i \in S) \lor A^{-1}(i \in S \land n < m)) \land R_I \{\text{INCLUD}\} \land A^{-1}(S' = S \cup \{i\}) \land R_I
\]

where **INCLUD** symbolises the implementation of the **Include** event

**Proof**: 

Augmenting the implementation with assertions (shown in the \{\} brackets above) gives:

**Line 4-7**  
define \( P[1..j] = \forall k: 1..j-1 \cdot S(k) \neq i \)  
and \( P[1..j] = \forall k: 1..j \cdot S(k) \neq i \)  
assume that \( P[1..1] = R_I \)  
\( S(j) = i \Rightarrow \) loop terminates  
therefore \( J = j \Rightarrow P[1..j] \)  
therefore \( P[1..N] \Rightarrow (\forall j: 1..N \cdot A(j) \neq i) \land BF = false \)

**Line 8-12**  
\( N \neq M \lor (\forall j: 1..N \cdot A(j) \neq i) \Rightarrow \) exception  
an exception is raised if the precondition upon the parameters is false  
if no exception is raised then \( N < M \land (\forall j: 1..N \cdot A(j) \neq i) \)

**Line 13-14**  
\( N' = N+1 \land A'(N') = i \land N' \leq M \)

**Line 15**  
reached from line 7 and line 14

**Case Analysis**

**Case 1 Line 7**  
necessary condition \( \exists j: 1..N \cdot A(j) = i \)  
therefore \( A' = A \land N' = N \land R_I \)  
therefore \( (\forall j: 1..N \cdot (\exists k: 1..N' \cdot A(j) = A'(j))) \land (\exists j: 1..N' \cdot A'(j) = i) \land (\forall j: 1..N' \cdot A'(j) = A(k))) \land R_I \)

**Case 2 Line 14**  
\( N' = N+1 \land A'(N') = i \land N' \leq M \)  
therefore \( (\exists j: 1..N' \cdot A'(j) = i) \)

also  
\( (\forall j: 1..N \cdot (\exists k: 1..N' \cdot A(j) = A'(j))) \)

therefore \( (\forall j: 1..N' \cdot A'(j) = i) \lor (\exists k: 1..N \cdot A'(j) = A(k))) \)

therefore \( (\forall j: 1..N \cdot (\exists k: 1..N' \cdot A(j) = A'(j))) \land (\exists j: 1..N' \cdot A'(j) = i) \land (\forall j: 1..N' \cdot A'(j) = A(k))) \land R_I \)

150
(2.2) Implementation of the Exclude event

```fortran
ENTRY EXCLUD(I)
{R_I}
J=0
BF=.FALSE.
WHILE ((J.LT.N).AND.(.NOT.BF)) DO
J=J+1
BF=A(J).EQ.I
END WHILE
IF (BF) THEN
A(J)=A(N)
N=N-1
END IF
{Vj: 1 ..N} (3k:1 ..N' • (A^' (k)=A(j) \& A^' (k) \# I))
RETURN
```

The precondition is true.
The postcondition is i \notin S'.

The postcondition for the event (including the central set) is \( S' = S \setminus \{i\} \).

Applying the inverse of the abstraction function to these gives:

\( A^{-1}(S' = S \setminus \{i\}) = (\forall j:1..N' \exists k:1..N' • (A^\prime (k)=A(j) \& A^\prime (k) \# I) \lor A(j)=I) \)

**Claim:**

\( R_I \{EXCLUD\} A^{-1}(S' = S \setminus \{i\}) \land R_I \)

where EXCLUD symbolises the implementation of the Exclude event.

**Proof:**

Augmenting the implementation with assertions (shown in the {} brackets above) gives:

Line 4-11 define \( P [1..j] = \forall k:1..j-1 \bullet A(k) \neq I \)

and \( P [1..j] = \forall k:1..j \bullet A(k) \neq I \)

assumed that \( P [1..1] = R_I \)

\( I = A(J) \Rightarrow A^\prime (J) = A(N) \land N' = N-1 \) (execute line 10)

therefore \( P [1..N] = (\forall k:1..N \bullet A(k) \neq I) \)

Line 12 reached from line 8 and line 11

case analysis

case 1 line 8

necessary condition \( \exists k:1..N \bullet A(k) = I \)

\( A(K) = I \Rightarrow A^\prime (K) = A(N) \land N' = N-1 \)

\( N \neq 0 \land N' = N-1 \Rightarrow N' < M \)

therefore \( (\forall j:1..N \bullet \exists k:1..N' • (A^\prime (k)=A(j) \land A^\prime (k) \neq I) \lor A(j)=I)) \land R_I \)

case 2 line 11

\( (\forall k:1..N \bullet A(k) \neq I) \)
Appendix C

Verification for the Set Data Structure

therefore $A'=A \land N'=N \land R_I$

therefore $(\forall j:1..N \cdot (\exists k:1..N' \cdot (A'(k)=A(j) \land A'(k)\neq I) \lor A(j)'=I)) \land R_I$

(2.3) Implementation of the \texttt{AnyMemb} event

\begin{verbatim}
1 ENTRY ANYMEM(I)
   [R_I]
2   IF (N.EQ.0) THEN
3      CALL EXCEPT
4      RETURN
5   END IF
6   I=A(N)
7   N=N-1
8   \{ (\exists k:1..N \cdot A(k)=I) \land \\
     (\forall j:1..N \cdot (\exists k:1..N' \cdot (A'(k)=A(j) \\
      \land A'(k)\neq I) \lor A(j)'=I)) \land R_I \}
9   RETURN
\end{verbatim}

The precondition from the specification is $\exists i:S$

The postcondition from the specification is $i \notin S'$

The postcondition for the event (including the central set) is $S'=S\setminus\{i\}$

Applying the inverse of the abstraction function to these gives:
$$A^{-1}(\exists i:S) = (N\neq 0)$$
$$A^{-1}(S'=S\setminus\{i\}) = (\exists k:1..N \cdot A(k)=I) \land (\forall j:1..N \cdot (\exists k:1..N' \cdot (A'(k)=A(j) \\
\land A'(k)\neq I) \lor A(j)'=I))$$

\textbf{Claim}:

$$A^{-1}(\exists i:S) \land R_I(\text{ANYMEM}) \land A^{-1}(S'=S\setminus\{i\}) \land R_I$$

where \texttt{ANYMEM} symbolises the implementation of the \texttt{AnyMemb} event

\textbf{Proof}:

Augmenting the implementation with assertions (shown in the \{\} brackets above) gives:

Line 2-5 $N=0 \Rightarrow$ exception

an exception is raised if the precondition upon the parameters is false

if no exception is raised then $N\neq 0$

Line 6-7 $I=A(N) \Rightarrow (\exists k:1..N \cdot A(k)=I)$

$N'=N-1 \Rightarrow (\forall j:1..N \cdot (\exists k:1..N' \cdot (A'(k)=A(j) \land A'(k)\neq I) \lor A(j)'=I)$

$A(j)'=I)$

$N\neq 0 \land N'=N-1 \Rightarrow N'\neq M$

$N>0 \land N'=N-1 \Rightarrow N'\geq 0$

therefore $(\exists k:1..N \cdot A(k)=I) \land (\forall j:1..N \cdot (\exists k:1..N' \cdot (A'(k)=A(j) \land A'(k)\neq I) \lor A(j)'=I) \land R_I$
Appendix C

(2.4) Implementation of the empty event

\begin{verbatim}
1 ENTRY EMPTY(B)
   (R₁)
2 B=N.EQ.0
   { (B=N=0) ∧ A'=A ∧ N'=N ∧ R₁ }
3 RETURN
\end{verbatim}

The precondition is true
The return value is \( b = (S = \emptyset) \)
The postcondition is true
The postcondition for the event (including the central set) is \( s' = s \)
Applying the inverse of the abstraction function to these gives:
\( \mathcal{A}^{-1}(b = (S = \emptyset)) = (B = (N = 0)) \)
\( \mathcal{A}^{-1}(s' = s) = A' = A ∧ N' = N \)

\textbf{Claim}:
\( R₁ \) \( \mathcal{A}^{-1}(b = (S = \emptyset)) ∧ \mathcal{A}^{-1}(s' = s) ∧ R₁ \)
where \texttt{EMPTY} symbolises the implementation of the empty event.

\textbf{Proof}:
Augmenting the implementation with assertions (shown in the \{ \} brackets above) gives:
Line 2 \( B = (N=0) \)
no changes made therefore \( A' = A ∧ N' = N ∧ R₁ \)
therefore \( (B = (N=0)) ∧ A' = A ∧ N' = N ∧ R₁ \)

(2.5) Implementation of the card event

\begin{verbatim}
1 ENTRY CARD(K)
   (R₁)
2 K=N
   { K=N ∧ A'=A ∧ N'=N ∧ R₁ }
3 RETURN
\end{verbatim}

The precondition is true
The return value is \( k=n \)
The postcondition is true
The postcondition for the event (including the central set) is \( s' = s \)
Applying the inverse of the abstraction function to these gives:
\( \mathcal{A}^{-1}(k = n) = K=N \)
\( \mathcal{A}^{-1}(s' = s) = A' = A ∧ N' = N \)
Appendix C

Verification for the Set Data Structure

Claim:
\[ R_I \{ (\text{CARD}) \ A^{-1}(k=n) \land A^{-1}(s'=s) \land R_I \} \]

where CARD symbolises the implementation of the \text{card} event

Proof:
Augmenting the implementation with assertions (shown in the \{\} brackets above) gives:

Line 2-7
\[ R^n = N \]
no changes made therefore \( A' = A \land N' = N \land R_I \)

therefore \( R = N \land A' = A \land N' = N \land R_I \)

(2.6) Implementation of the \text{member} event

```plaintext
1 ENTRY MEMBER(I, B)
2 \{R_I\}
3 J = 0
4 B = FALSE.
5 WHILE ((J.LT.N) .AND. (.NOT. B)) DO
6 J = J + 1
7 B = A(J).EQ.I
8 END WHILE
9 RETURN
```

The precondition is true
The return value is \( b = (i \in S) \)

The postcondition is true

The postcondition for the event (including the central set) is \( S' = S \)

Applying the inverse of the abstraction function to these gives:
\[ A^{-1}(b = (i \in S)) = (B \Leftrightarrow \exists j : 1..N \cdot A(j) = I) \]
\[ A^{-1}(s' = S) = A' = A \land N' = N \land R_I \]

Claim:
\[ R_I \{ (\text{MEMBER}) \ A^{-1}(b = (i \in S)) \land A^{-1}(s' = s) \land R_I \} \]

where MEMBER symbolises the implementation of the \text{member} event

Proof:
Augmenting the implementation with assertions (shown in the \{\} brackets above) gives:

Line 2-7 define \( P[1..j][ = \forall k: 1..j-1 \cdot A(k) \neq I \)
and \( P[1..j][ = \forall k: 1..j \cdot A(k) \neq I \)
assumed that \( P[1..1][ = R_I \)
\[ A(i) = I \Rightarrow \text{loop terminates} \]
therefore \( P[1..N] \Rightarrow \forall j : 1..N \cdot A(j) \neq I \)

Line 8 \( (\exists k : 1..N \cdot A(k) = I) \Rightarrow B \)
or $-B \land \forall j:1..N \cdot A(j) \neq i$

no changes made therefore $A' = A \land N' = N \land R_f$

therefore ($B \iff \exists j:1..N \cdot A(j) = i) \land A' = A \land N' = N \land R_f$)

Therefore, the implementation of the set data structure satisfies the specification.
In this appendix the specifications and implementations of the stack, queue, abstract array and binary relation are presented.

### D.1 The Stack Object Class

A stack is a list of items, which are accessed using a last-in first-out (LIFO) mechanism. The specification and implementation of a bounded stack is given in Figure D.1.

```plaintext
SUBROUTINE STACK
IMPLICIT INTEGER (D)
SAVE
PARAMETER (M=1000)
INTEGER S(1:M)
INTEGER N
N=0
RETURN

ENTRY DEPTH(D)
D=N
RETURN

ENTRY PUSH(I)
IF (N.GE.M) THEN
CALL EXCEPT
RETURN
END IF
N=N+1
S(N)=I
RETURN

ENTRY POP(I)
IF (N.EQ.0) THEN
CALL EXCEPT
RETURN
ENDIF
I=S(N)
N=N-1
RETURN

END
```

**Figure D.1** The specification and implementation of the bounded stack
The formal parameter \( m \) places an upper bound upon the number of items in the stack through the invariant. The sequence \( s \) is the internal representation of the stack. The operation \texttt{Push} adds an item to the top of the stack and the operation \texttt{Pop} removes and returns an item from the top of the stack. An item may not be pushed onto a full stack or popped from an empty stack. \texttt{Stack\_depth} returns the number of items on the stack.

### D.2 The Queue Object Class

A queue is a list of items, which are accessed using a first-in first-out (FIFO) mechanism. The specification and implementation of a bounded queue is given in Figure D.2.

```plaintext
SUBROUTINE QUEUE(M)
PARAMETER (M=1000)
SAVE
INTEGER Q(1:M)
INTEGER F, B
F=0
B=1
RETURN

ENTRY LENGTH(L)
IF (F.EQ.0) THEN L=0
ELSE IF (B.EQ.F) THEN L=M
ELSE IF (B.GT.F) THEN L=B-F
ELSE L=M-(F-B)
END IF
RETURN

ENTRY ENQ(I)
IF (F.EQ.B) THEN
CALL EXCEPT
RETURN
END IF
IF (F.EQ.0) F=B
Q(B)=I
B=B+1
IF (B.GT.M) B=1
RETURN

ENTRY DEQ(I)
IF (F.EQ.0) THEN
CALL EXCEPT
RETURN
END IF
I=Q(F)
F=F+1
IF (F.GT.M) F=1
IF (F.EQ.B) F=0
RETURN
END
```

**Figure D.2** The specification and implementation of the bounded queue
Appendix D The Stack, Queue, Abstract Array and Binary Relation Object Classes

The operation `enq` adds an item to the back of the queue and the operation `deq` removes an item from the front of the queue. An item may not be added into a full queue or removed from an empty queue. `queue.length` returns the number of items in the queue.

D.3 The Abstract Array Object Class

An abstract array is a partial function from the domain onto values of the codomain. The specification and implementation of a bounded abstract array is given in Figure D.3.

In the specification of the abstract array a generalisation has been made which simplifies the specification by taking the bounds to be between one and an integer constant, which is passed as the parameter m. The state schema contains the declaration of the domain, a, and A, the partial function itself. `array.size` returns the current cardinality of the d set. `array.isdef` allows a test to be made to determine whether a given index value is a member of the d set. `array.access` returns the base type value associated with a given member of the d set. The `array.assign` operation adds a value n to the d set and associates the base type value i with it (or changes the existing association to the value i). `array.delete` removes a value n from the set d.
**Appendix D**

The Stack, Queue, Abstract Array and Binary Relation Object Classes

### SUBROUTINES ARRAY

**IMPLICIT LOGICAL (B)**

**PARAMETER** (M=1000)

**SAVE**

**INTEGER** A(1:MX)

**LOGICAL** D(1:MX)

**DO 10,K=1,MV 20**

D(K)=.FALSE.

**10 CONTINUE**

**RETURN**

**ENTRY SIZE(N)**

N=0

**DO 20,K=1,M**

**IF (D(K)) N=N+1**

**20 CONTINUE**

**RETURN**

**ENTRY ISDEF(N,B)**

**IF ((N.GT.M).OR.(N.LT.1)) THEN**

**CALL EXCEPT**

**RETURN**

**END IF**

B=D(N)

**RETURN**

**ENTRY ACCESS(N,I)**

**IF ((N.GT.M).OR.(N.LT.1).OR.**

**(.NOT.D(N)) THEN**

**CALL EXCEPT**

**RETURN**

**END IF**

I=A(N)

**RETURN**

**ENTRY ASSIGN(N,I)**

**IF ((N.GT.M).OR.(N.LT.1)) THEN**

**CALL EXCEPT**

**RETURN**

**END IF**

A(N)=I

D(N)=.TRUE.

**RETURN**

**ENTRY DELETE(N)**

**IF ((N.GT.M).OR.(N.LT.1).OR.**

**(.NOT.D(N)) THEN**

**CALL EXCEPT**

**RETURN**

**END IF**

D(N)=.FALSE.

**RETURN**

**END**

### Figure D.3 The specification and implementation of the abstract array

<table>
<thead>
<tr>
<th>ARRAY ( (m) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m: \text{NAT} )</td>
</tr>
<tr>
<td>( d: \text{set}[\text{NAT}] )</td>
</tr>
<tr>
<td>( A: \text{NAT} \to \mathbb{I} )</td>
</tr>
<tr>
<td>( {1..m} \supseteq d )</td>
</tr>
<tr>
<td>( \text{dom} \ A = d )</td>
</tr>
<tr>
<td>( d' = \emptyset )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ARRAY. size ( (\to n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n: 0..m )</td>
</tr>
<tr>
<td>( n = #d )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ARRAY. isdef ( (n \to b) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n: 1..m )</td>
</tr>
<tr>
<td>( b: \text{BOOL} )</td>
</tr>
<tr>
<td>( b = (n \in d) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ARRAY. access ( (n \to i) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n: d )</td>
</tr>
<tr>
<td>( i: \mathbb{I} )</td>
</tr>
<tr>
<td>( A(n) = i )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ARRAY. Assign ( (n,i) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n: 1..m )</td>
</tr>
<tr>
<td>( i: \mathbb{I} )</td>
</tr>
<tr>
<td>( A'(n) = i )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ARRAY. Delete ( (n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n: d )</td>
</tr>
<tr>
<td>( n \in d' )</td>
</tr>
</tbody>
</table>
D.4 The Binary Relation Object Class

A binary relation $R$ is a subset of the Cartesian product of two sets $A, B$. The specification and implementation of a bounded binary relation is given in Figure D.4.

The binary relation is represented by the component $r$. The cardinality of $R$ is restricted by the parameter $n$. The query $\text{RELATE}.\text{Member}$ performs a membership test and $\text{RELATE}.\text{card}$ returns the cardinality of the relation itself. The $\text{RELATE}.\text{Add}$ event adds an ordered pair $(i, j)$ to $R$. The event is overloaded to take account of the two cases where $(i, j)$ is not or is a member of $R$ before the operation. The event $\text{RELATE}.\text{Remove}$ removes an ordered pair from $R$. $\text{RELATE}.\text{member}$ test whether an ordered pair is a member of $R$. 
Subroutine `RELATE`

```fortran
SUBROUTINE RELATE
IMPLICIT LOGICAL (B)
SAVE
PARAMETER (M=1000)
INTEGER R(1:2,1:M)
INTEGER N
LOGICAL BF

N=0
RETURN

ENTRY MEMBER(I,J,B)
K=0
B=.FALSE.
WHILE ((K.LT.N).AND.(.NOT.B)) DO
    K=K+1
    B=(I.EQ.R(1,K)).AND.(J.EQ.R(2,K))
END WHILE
RETURN

ENTRY CARD(K)
K=N
RETURN

ENTRY ADD(I,J)
K=0
BF=.FALSE.
WHILE ((K.LT.N).AND.(.NOT.BF)) DO
    K=K+1
    BF=(I.EQ.R(1,K)).AND.(J.EQ.R(2,K))
END WHILE
IF (.NOT.BF) THEN
    IF (N.GE.M) THEN
        CALL EXCEPT
        RETURN
    END IF
    N=N+1
    R(1,N)=I
    R(2,N)=J
END IF
RETURN

ENTRY REMOVE(I,J)
K=0
BF=.FALSE.
WHILE ((K.LT.N).AND.(.NOT.BF)) DO
    K=K+1
    BF=(I.EQ.R(1,K)).AND.(J.EQ.R(2,K))
END WHILE
IF (BF) THEN
    R(1,K)=R(1,N)
    R(2,K)=R(2,N)
    N=N-1
END IF
RETURN
```

**Figure D.4** The specification and implementation of the binary relation