University of South Wales

2059305

Bound by Abbey

Bookbinding Co.,
Cardiff, South Wales
Tel:(01222) 395882

The Refinement of Formal Specifications Using Reusable Software Components in Ada95

By

Stephen Alan Bale BSc (Hons)

A thesis submitted in partial fulfilment of the requirements of the University of Glamorgan/Prifysgol Morgannwg for the degree of Doctor of Philosophy.

School of Accounting and Mathematics
Division of Mathematics and Computing
The University of Glamorgan
Mid Glam
CF37 1DL

August 1998.

Contents

Acknowledgements i
Certificate Of Research ii
Declarations iii
Abstract iv

Chapter 1. Introduction

1.1 Background 2
1.2 The Z Formal Specification Language 3
1.3 Project Aims and Strategy 5
1.4 Construction of Reusable Components to Model Z 6
1.5 Abstract Data Types 6
1.6 Choosing the Target Language 7
1.7 Overview of Project 8
1.8 Overview of Thesis 8

Chapter 2. Literature Survey

2.1 Introduction 12
2.2 Refinement Calculus and Refinement Methods 12
2.3 Rapid Prototyping and Animation 19
2.4 Bridging the Gap Between the Specification and Target Languages 28
2.5 Conclusion 36

Chapter 3. Overview of Methodology

3.1 Introduction 40
3.2 The Birthday Book Specification 40
3.3 Translating the Birthday Book State Schema 41
3.4 Broad Overview of Method 46

Chapter 4. Construction of Reusable Components

4.1 Introduction 48
4.2 Outline of Reusable Components 49
4.3 Construction of Efficient Data and File Structures 53
4.4 Construction of Utility Packages Using Layering 54
4.5 Construction of Z Operators Using Utility Packages 57
4.6 Testing the Reusable Components 59

Chapter 5. State Schema Refinement

5.1 Introduction 63
5.2 Methodology 63
5.3 Refinement of State Schemas 64
5.4 The State Invariant 71
5.5 Multiple States and Inclusion 72
5.6 Schema Types and Schema Bindings 76

Chapter 6. Operation Schema Refinement

6.1 Introduction 83
6.2 Function or Procedure Bodies 83
6.3 Refining the Initial State Schema 83
6.4 The Declarative Part of Schemas 85
6.5 Refinement of Z Statements 88
6.6 Refinement of Preconditions 89
6.7 Comprehension Terms 101
6.8 Refining Complex Postconditions and Statements 111
6.9 Operations Involving Schema Types and Bindings 113
6.10 Schema Calculus 117

Chapter 7. Evaluation of Method

7.1 Introduction 130
7.2 Comparison of Code Produced Manually 130
7.3 Comparison with Code Produced from Concrete Design 137
7.4 Other Languages for Modelling Z 152
7.5 Scaling up to Industrial Sized Problems 163
7.6 Summary of Conclusions 195

Chapter 8. Advantages of Ada95 Over Ada83

8.1 Introduction 199
8.2 Improvements in the Generic Paradigm 199
8.3 Iterators 202
8.4 The Use of Child Packages 204

Chapter 9. Future Work and Conclusions

9.1 Introduction 210
9.2 Quality of the Reusable Components and the Code Obtained. 210
9.3 Viability of the Method 211
9.4 Advantages in Rapid Prototyping Terms. 211
9.5 Efficiency of the Obtained Code. 212
9.6 Scaling up to Industrial Sized Specifications. 213
9.7 Preserving the Style of the Original Specification 214
9.8 Non Functional Requirements. 215
9.9 The Use of Ada95 216
9.10 Criticisms of the Method 216
9.11 Advantages of Using Reusable Components. 218
9.12 Future work 219

References 220

Appendix 1 - List of Software Components and Implemented Specification Case Studies.

Appendix 2 - List of Z to Ada Translations.

Acknowledgements

I would like to express my sincere thanks to my supervisors at the University of Glamorgan and to Dr J. Hayward for his support, encouragement and advice throughout the duration of this research project.

I would also like to thank the Higher Education Funding Council for Wales (HEFCW) for their financial contribution to allow the University to fund this research under its DevR scheme.

Certificate Of Research

This is to certify that, except where specific reference is made, the work presented in this thesis is the result of the investigation undertaken by the candidate.

Candidate	....
Director of studies	....

Declarations

This is to certify that neither this thesis or any part of it has been presented or is being currently submitted in candidature for any other degree other than that of Doctor of Philosophy of the University of Glamorgan / Prifysgol Morgannwg.

Candidate ....

Abstract

The Refinement Of Z Specifications Using Reusable Software Components Written In Ada95

This thesis documents research that enables formal specifications, written in the specification language Z, to be turned into high level code in fewer steps than other refinement techniques and without the need for formal proof. This method helps to overcome one of the main stumbling blocks for formal methods, which is the difficulty of creating software from the formal specification. In the main, previous methods have either concentrated on creating an animation or prototype of the specification using functional languages or on converting the abstract structures found in specification into the less abstract structures found in high level programming languages, via a series of lengthy proofs.

The method presented in this thesis uses a different approach. Here, the target language is enriched with the abstract structures and operations available to the formal specification language. This is achieved by the construction of a series of reusable software components that model the main types and operations found in Z. The formal specification can then be translated into executable code by selecting the correct operations from the reusable components to implement each of the Z operations.

The research described in this thesis shows that the method is a viable one, as efficient executable code can be produced very quickly, without the need for formal proof, and with great confidence in its correctness. The components required to do this are available and have been written and constructed in such a way as to allow more complex components to be built from them.

Chapter 1

Introduction

1.1 Background

Over the past few decades, the number of complicated software systems in existence has grown enormously. As the complexity of these systems increases, they become more error prone due to the task of designing and coding the system. Many systems delivered to customers contain errors, do not meet the customer's requirements, are late and over budget. This problem was first recognised in the nineteen sixties and is known as the software crisis. Formal methods have arisen in recent years in an attempt to cure these problems. At present, most specifications are written in natural language, but this leads to specifications which are imprecise and ambiguous. Formal methods however, are techniques that use mathematical principles to develop software, the goal being to use the precision of mathematics to improve the quality of the software.

One very important aspect of formal methods is the process of turning the specification into executable code. This is called reification or refinement [McDe89]. The process is easier when functional and logic languages are the target language because they support sets or predicate calculus. For this reason, languages such as Miranda and Prolog are often used to animate specifications or prototype the software [Dill90,West92]. However, many specifications must be refined into third generation languages, which do not support sets or predicate calculus. Most of the work done in this area concentrates on methods that convert the abstract structures found in specifications into less abstract structures found in many high level languages [Woodk91,Woodk93]. The refinement calculus [Morg94] can be used with Z to provide a path from the abstract Z specification to executable code. The Z specification is successively refined and turned into the refinement calculus from which executable code can be derived with the aid of Dijkstra's guarded command language[Dijk75]. King [King90] gives rules for translating Z specifications into Morgan's refinement calculus and shows the development process. Other examples of this method can be found in the books by Woodcock [Woodc96] and Wordsworth [Word92]. A different approach is pursued by Valentine [Vale91]. Here, a

computational subset of Z named Z-- is used to move the specification through successive refinement stages to arrive at executable code.

An alternative to refinement within Z itself, is to bridge the gap between the formal specification and the target language. Wood et al [Woodw91] describe this latter method using Anna (annotated Ada) as the bridge between Z and Ada. Other methods exist for refining formal specifications, such as the B-method [Abri91], which has considerable tool support for refinement proofs.

1.2 The Z Formal Specification Language

The language used for this research project has been Z [Bard94,Spiv89,Word92] which is a formal specification language developed by the Programming Research Group at Oxford University. The notation is capable of presenting the mathematics and decisions made in the development of systems in a readable yet rigorous framework. The use of Z in writing specifications allows the developer to explore fundamental aspects of the system and can uncover design faults that may have been overlooked.

The language is designed to be read by humans, with the mathematics describing each operation or state enclosed in a box structure called a schema. Each schema is followed by English text to further describe the actions. Schemas allow large specifications to become manageable, as pieces of a system can be specified and then put into global context with schema promotion.

Z is based upon sets, and uses discrete mathematics and first order logic to fully describe each operation. It can describe both the static and dynamic aspects of the system. The static aspects include the state the system can take, and the invariant relationships that hold as the system changes. Dynamic aspects include the mathematical relationships between input and output, and the changes of state that are allowed to happen. However, Z lacks the features required for expressing real

time constraints and behaviour. It can be used to model real time systems, but there is little agreement on how to do it and no one method has been found to be superior [Fidg92]. Another feature that Z lacks is a modular structure (apart from the schema) for specifying software components or object orientated specifications; although extensions to Z to remedy these aspects have been proposed [Carr91].

Z is a specification language that has been used in large industrial applications. A good example of this is the IBM Customer Information Control System (CICS). Several case studies involving Z and CICS can be found in the book by Hayes [Haye87].

1.2.1 A Z Success Story

An experimental study, carried out by Goel and Sahoo [Goel91], compared the development time and the number of errors found during development, for software written from an English specification, with software written from a Z specification. The problem chosen was the NASA Launch Interceptor Problem, which is a simple but realistic representation of an anti-missile system. Programmers derived versions of the software from Z specifications using C and Ada. These were then compared with others versions derived from English specifications. It was noted that the versions developed from Z used less lines of code, and the development time was less. The main findings of the study were that the effort spent in writing Z specifications was compensated for by the reduced effort needed during later stages of development. Also, the number of errors found during development and testing were significantly smaller in versions developed from formal specifications compared with those developed informally.

1.2.2 Drawbacks

The method of turning formal specifications into code by successive refinement of Z specifications using the refinement calculus is a difficult process. Much work is required to prove that the new 'concrete' specification does not violate the original abstract specification. These proofs are carried out using predicate calculus and are

both difficult and time consuming, although there are now automated proof tools available. The difficulty of turning a formal specification into code is one reason why such a small percentage of software written in industry has been specified formally. A study by Smith [Smit91] indicates that formal refinement is currently impractical on an industrial scale. King [King90] states that there has been significantly less use of Z in the latter stages of development. Z has been used to document designs and specifications, but in very few cases have the proofs been completed.

1.3 Project Aims and Strategy

This project uses an alternative strategy to those described earlier. Instead of making the formal specification close to the target programming language, the programming language is enriched to model the structures found in the specification language itself. This is achieved by the construction of a number of reusable software components to model the types and operations found in the formal language. It should then be possible to translate a formal specification into executable code with high confidence in its correctness.

This thesis seeks to demonstrate that an imperative language can be enriched to provide a potentially viable method of translating a Z specification in order to arrive at code that can be executed. Other researchers have used functional languages mainly due to the fact that functional languages support sets.

This project is also concerned with reducing the number of refinement steps, making the refinement process simpler, and programming abstractly in Ada. This idea is not new [Jack85,Rann94], but the actual creation of a collection of reusable software components containing the operations found in Z, the implementation of the method, the problems arising from it, and ways to make the refinement progress smoothly have not been investigated in previous work.

1.4 Construction of Reusable Components to Model Z

Reusing software components is part of the object based methodology found in modern languages such as Ada and C++, and is aimed at turning software construction into an engineering discipline. Data structures and algorithms form the building blocks from which all software systems stem, and so make excellent candidates for reusable components. If there exists a rich supply of software components whose behaviour is known, then it is possible to construct more complicated systems with high degrees of confidence in their correctness. This will result in improved production, with less time required for development, integration and testing [Booc87].

The main types in Z are the set, function, relation and sequence. In order to turn a Z specification into code (using the method detailed in this thesis), it is necessary to have many components for each of these types to satisfy any performance or behavioural characteristics of the system. Each type of component must contain functions or procedures to model each of the operations available in Z for the type. Also, each component must have generic functions and procedures, parameterised over other functions, to implement quantifiers, comprehension statements, and iterators. The components themselves must also be parameterised to allow different instances of each package to be created when refining the state schema. The types in the package must be exported, but access to the underlying structure of the types must be prohibited. In order to satisfy these requirements, each component has been implemented as an abstract data type.

1.5 Abstract Data Types

An abstract data type is a structured type that can only be accessed by a well defined set of operations specified in an external interface [Booc87]. For example, the abstract data type stack can only be accessed by its operations e.g. pop, push, top etc. One way of visualising an abstract data type is as a black box. The user provides the correct input for an operation, and the black box provides the answer. The

abstract view is the view from outside the black box, and this is the only view necessary for its use. The internal workings of the black box are unimportant to its user. Abstraction is only possible if the internal workings of the black box are kept hidden. Hiding more information leads to more abstraction.

Abstract data types are seen by many people as a way of enriching a language with reusable software components [Booc87], which is the philosophy followed by this research project. Using techniques such as data hiding, encapsulation and the generic features of a language such as Ada, more abstract data structures can be provided, rather than the pre-defined arrays, pointers, and records etc. of high level languages. Ada provides the ability to define an abstract data type where the implementation is invisible to the outside world. Variables of this type can then be instantiated, and thought of as objects, while the abstract data type may be thought of as a class.

1.6 Choosing the Target Language

The Language Ada was chosen for this project because :-

- It is a very stable language established as an ANSI (American National Standards Institute) standard in 1983, and no subsets or supersets of Ada83 are allowed. Ada is subject to design reviews to keep up with developments in the software industry. This resulted in a new version, Ada95 which is also subject to ANSI standards. This version improved and added new features to Ada83. The next review is scheduled for the next century. Additionally all Ada compilers have to undergo a rigorous validation program before being certified. These measures improve the portability of Ada programs between machines.
- It allows a rich variety of packages and tasks.
- It is strongly typed and imposes constraints on the types of formal and actual parameters and those variables obtained from instances of generic packages.
- It is possible to parameterise software components over values, types, objects, and formal subprograms using Ada's generic mechanism. Ada95 now allows parameterisation of generic components over other components.

- It utilises modern software engineering philosophies of abstraction, information hiding, and code reuse.

1.7 Overview of Project

The project is at the stage where there is a complete set of components available enabling Z specifications to be translated into code directly. Moreover, a number of different components exist for each of the main types in Z, allowing users requirements for performance and storage to be met (a list of these components and the specification case studies already implemented can be found in appendix 1). The operations in the reusable components cover those operations found in the Z mathematical Tool Kit [Spiv89]. The translation of a Z specification provides a package that can be compiled but not executed. In order to execute the package it must be instantiated in other software that can then access and use the operations in the translated Ada package. The Ada packages obtained from the Z specifications used throughout this project have all been executed by writing programs to instantiate and test them (with the exception of the steam boiler and aircraft illumination specifications, due to the difficulty of simulating the steam boiler and aircraft illumination equipment and a lack of non-functional data).

1.8 Overview of Thesis

Chapter two is a literature survey chapter, discussing other work in the field of refining Z specifications to produce executable code.

Chapter three briefly highlights the program development method used when translating a Z specification into Ada code. A simple specification of the birthday book [Spiv89] is shown along with its equivalent implementation in Ada using reusable components. A program necessary to instantiate the Ada birthday book package implementation and use its operations is also shown. The details behind the derivation of the Ada code from the Z specification are explained in detail in chapters five and six.

Chapter four discusses the construction of the reusable components necessary for the method and describes a process of constructing components, that are built upon existing components, to provide efficient data structures by programming at a higher level of abstraction. This technique allows complex components to be constructed with much speed and ease. It also shows how some of the operators in Z that operate over complex types can be programmed in Ada95. As an example the code implementing the distributed union operator, defined by $\bigcup : \mathbb{P}(\mathbb{P}\ X) \rightarrow \mathbb{P}\ X$, is given.

Chapter five deals with the translation of the state schema to provide the state variables upon which the operations of the Z specification can act. The organisation of the Ada specification(s) implementing the Z specification is also discussed because some Z specifications use promotion and inclusion and cannot always be implemented in a single package. A novel way of implementing specifications that are based upon multiple states is presented. Schema types and bindings and the state invariant are also discussed.

Chapter six discusses the translation of operation schemas and the Z statements contained within them. It highlights the issues relating to the translation of the operation precondition (including quantifiers) and a technique to translate complex statements in the postcondition. Chapter six also introduces the operators of the schema calculus and their respective refinement in Ada. Schema calculus is important with regard to the construction of complex specifications that are built up using simpler specifications. The guidelines for the translation of Z operators for each of the main types are given in appendix 2.

Chapter seven evaluates this method of refinement by comparing the code produced using reusable components with the code that could be produced manually. Also, other languages such as C++ [Skan97] and the functional language Haskell

[Thom96] are examined to see how they could be used to produce the reusable components for implementing Z specifications. Finally, two case studies are included to examine how well real world Z specifications can be refined using reusable components in Ada95.

Chapter eight discusses the advantages that were brought to the method when Ada95 was used as the target language instead of Ada83. The advantages spanned the construction of the reusable components and the implementation of some of the more complex issues in Z.

Chapter nine contains the conclusions and recommendations for future work.

Chapter 2

Literature Survey

2.1 Introduction

The Z notation has been used successfully in the specification of many software systems [Crai95a]. However, specifications are not always seen as an end in themselves and whilst Z is good for specifying system behaviour, it is not so suitable for the refinement of these specifications into executable code. This is an important part of formal methods, which is not easy to achieve. As a result, many methods have been proposed to smooth the transition from the abstract specification to executable code.

The main methods examined in this chapter can be broadly classified in the following manner (although there is an overlap between classes in some methods discussed below) :-

1) those methods that successively refine the specification by stepwise refinements moving from the abstract specification to a more concrete specification.

2) those methods that produce an executable prototype of the specification by using a target language which is itself based upon sets and predicate calculus.

3) those methods that seek to bridge the gap between the formal specification and the target programming language by working in the Z domain.

2.2 Refinement Calculus and Refinement Methods

Many specifications must be refined into third generation languages which do not support the sets and predicate calculus found in the specification language. In these instances, it is necessary to convert the abstract structures found in the specification into the less abstract arrays, pointers etc. of the high level language. There are two processes involved in the refinement of model based specifications. The first is data refinement where the abstract mathematical representation of data is expressed by a

concrete equivalent and secondly, algorithm refinement where the operations of the specification are refined to produce an executable equivalent.

MacDonald and Sennet [MacD89] identify two aspects which may cause an initial specification to change even if all parties concerned agree that it meets the user requirements.

1. In moving to a design that will actually work on a computer some properties may need to be respecified in order to make the specification implementable.
2. There may be several equivalent ways of expressing the same properties, but some definitions may be easier to prove correct than others. The specification may be changed to utilise the definitions that make the proof easier.

The refinement methods and the refinement calculus are both based upon the successive stepwise transformation of specifications. In refinement methods [Word92, Woodc96], a concrete version of the specification is written by adding more detail to the abstract specification. This detail will include how the data is to be stored and how the abstract operations are made concrete by operating on the concrete state. As a result proof obligations are required to show that the new concrete specification is a valid representation of the original abstract specification.

The method involves the use of a simulation [Word92] or retrieve function [Litt92] to link the abstract state and the concrete state. This takes the form of a new schema which includes the abstract and concrete states and also a predicate which describes how their components are related.

When the simulation or retrieve schema is complete, the initial concrete schema must be defined which describes the concrete state when the system starts. Proof obligations are required to show that an initial state can be found and that any initial concrete state is correct with respect to the initial abstract state by the retrieve function. Mathematically the proofs [Word92] are stated as :-

$\vdash \exists$ ConcState′ • ConcInit and

ConcInit $\vdash \exists$ AbstractState′ • (AbstractInit $\wedge$ Retrieve′)

Retrieve is a function from the abstract state to the concrete state. Retrieve is primed because AbstractState is primed and the concrete state is primed in ConcInit (this sets the state to its initial value). When the concrete operations have been designed, they must also be shown to be correct with respect to the abstract operations. There are two obligatory proof obligations.

1) The Safety or Applicability proof is given by:-

pre AbstractOp $\wedge$ Retrieve $\vdash$ pre ConcOP

This states that if the current abstract state satisfies the precondition of the abstract operation, then the concrete state must satisfy the precondition of the concrete operation.

2) The Correctness proof is given by :-

pre AbstractOp $\wedge$ Retrieve $\wedge$ ConcOp $\vdash \exists$ AbstractState′ • (AbstractOp $\wedge$ Retrieve′)

This ensures that, when given a valid starting state for an abstract operation, performing the corresponding concrete operation will result in a state that corresponds to the abstract operation by the retrieve relation.

When the concrete design is complete and the proof obligations have been discharged for every operation schema, the algorithm refinement stage must follow in order to arrive at executable code. The approach taken is to use Djikstra's Guarded Command Language [Dijk75] to form a framework for the solution. Some statements may need further refinement here and they are written as subcomponents

for which specifications must be produced. The refinement is complete when each of the subspecifications are primitive statements in the target language. Wordsworth carries out safety and liveness proofs during these refinement stages also. However, Litteck [Litt92] says that the method need not be strictly followed (except in complicated situations) as refinements can be performed immediately producing algorithms straight from specifications.

A specification for a library system is given in [King89]. A design analysis is carried out to develop a concrete design and concrete data operations. The proofs that are required as discussed above, are stated, but left as an exercise for the reader. The concrete data designs are developed further towards executable code using Djikstras guarded command language, although the step from the guarded command language to the final implementation is not given. The work presented in this paper will be compared with an implementation using reusable components in chapter 7.3 to compare the code produced between the two methods.

In the refinement calculus [Back88,Morri87] the abstract specification is transformed by successive steps governed by the rules of the calculus which have been shown to provide valid refinements. The refinement calculus is based upon Djikstra's guarded command language extended with a specification construct to allow the formal use of specifications within programs. It has been designed as a wide spectrum language to support the whole development cycle from the abstract specification to code. The addition of the specification construct allows the distinction between programs and specifications to be banished. The specification construct is as follows :-

W: [pre,post]

where W - the frame listing the variable that may change

pre - describes the initial states

post - describes the final states.

As far as the refinement calculus is concerned all specifications are programs, and hence, not all programs are executable. Programs can include specifications and have constructions (called code) which are executable. In order for the entire program to be executable, the specifications must be removed from the program by the action of stepwise refinement, introducing more executability with each step. The book by Morgan [Morg94] introduces the thought behind the refinement calculus, its laws, the details and techniques of its use and provides case studies.

A number of papers have combined Z and the refinement calculus. King [King90] introduces a way of using Z specifications and the refinement calculus together in one development method. He describes the differences between Z and the refinement calculus and the reasons for those differences. He then describes a set of rules that allow a Z specification to be turned into the refinement calculus from which executable code can be derived. The development methodology is as follows :-

1. Write the specification in Z
2. Carry out data refinement in Z (as in the refinement methods by creating a concrete version of the specification).
3. Use the translation laws to turn the concrete specification into the refinement calculus.
4. Carry out algorithm refinement using the refinement calculus.

The aim of the work was to use each notation in the way that it is best suited and to provide a smooth path from the specification to executable code. Future work must be carried out to investigate how schema promotion (a very important aspect) fits into the method and King hints that future case studies may reveal small semantic changes that may be made to Z in order to make the development easier.

Wood [WoodK93] develops a method of using the specification properties of Z and the development capabilities of the refinement calculus *"without the need for any cumbersome translation between the two notations"* (presumably referring to

[King90]). This is achieved by dropping many of the standard features of Z and using the resulting streamlined Z with an unchanged refinement calculus. The method is described as the refinement calculus with a bit of Z (or R^Z). The Z specifications were written in such a way as to facilitate an easy transition from Z to the refinement calculus. Z schemas were only used to define compound types, whilst operation schemas were defined axiomatically to avoid them using schemas to change the state. Wood concludes that this restriction in the style of writing Z specifications did not impede the expressive power of the specifier and is justified due to the ease in which the refinement calculus can be introduced. He notes that one or two of the important features of the original method are lost but in addition to a seamless path from the specification to executable code, the approach provides encapsulation properties with are noticeably lacking in standard Z. This is due to the way in which schemas are used in this method which is significantly different to the use of schemas in standard Z. The Z specification is written in such a way as to describe a mathematical model of the problem domain where schemas define types but do not introduce any state to the system. The states are allocated when a variable of the appropriate schema type is declared in a scoped block of the refinement calculus. This lends encapsulation properties to the specification by reducing the prevalence of global data. A further restriction in the style of Z is that preconditions must be explicitly stated. The work detailed in this thesis also has this proviso, which, is deemed good practice [Good95a, Bard92]. Wood also notes that many of the operations will be specified as axiomatically defined functions which would lends themselves to rapid prototyping in a functional language, before the full refinement to imperative code is carried out. Other papers, discussed in section 2.3, also suggest that the full refinement product should be in an imperative and not a functional language. Another example of this refinement method is given in [WoodK91].

2.2.1 Critique of refinement methods and refinement calculi

The refinement methods described above have the advantage that a complete history of formal development is available from the specification through to executable code. This can enable errors to be easily traced and improves confidence in the systems correctness. However, formal proof is expensive, time consuming and requires a level of mathematical understanding greater than that needed for the formal specification and design. As a result very few proofs are actually carried out [Thom93]. Sceptics claim that formal methods are infeasible for any realistically size problem, whilst the less sceptical proponents claim that they should be applied selectively [Bowe93]. At Praxis, [Thom93] proof activity is limited because the proofs were judged not to be cost-effective for the systems developed at Praxis, which were rarely safety critical. Thomas also states that *"there seems to be widespread agreement amongst those with industrial experience of using formal methods that a lot of the benefits come from developing the formal specification (which takes relatively little time) compared with formal refinement and verification."* It was expected that a new generation of tools to assist with proofs would change this balance in the future. The lack of tool support is one reason cited in [Holl96, Gerh94] for the lack up uptake of formal methods in industry. Craigen et al [Crai95a] puts the lack of tool support down to:-

- The research and rapid prototyping nature of the tools make them unsuitable for industrial use.
- The steep learning curves associated with their use (up to 6 months as stated in [Crai95b]).

The methods described above pertain to be fully formal methods, but in reality, due to the complexities involved in using them, they are seldomly used in a fully formal manner.

2.3 Rapid Prototyping and Animation

The second class of refinement involves using functional or logic languages to produce a prototype or animation of the specification by translating it in a direct manner. A rapid prototype [Maud91] can improve software productivity by eliminating unnecessary rework. One of the major factors of rework are misunderstood specifications and interfaces [Boeh87]. A rapid prototype can allow the client to see and experiment with a system that is close to the final system to ensure that the requirements and behaviour are sound. This is an important activity because, even if the program is proven correct mathematically with respect to its specification, it does not imply that the specification describes what the customer wanted. The creation of a prototype has the advantage that the customer can have a demonstration of the systems intended behaviour at an early stage of the development cycle, allowing the developer to be confident that the customers requirements have been met by the formal specification.

Johnson et al [John90] describe a small example of how a Z specification can be turned into a functional language and shows that the resultant code cannot possibly be used as the final implementation because it is too slow. The program must be transformed using techniques such as those described in [Burn77] to produce a program whose run time behaviour is acceptable for a prototype of the system. Johnson proposes the following methodology.

1. Produce a formal specification of the system using Z
2. Refine the model into an explicit (constructive) representation and discharge proof obligations.
3. Use the formal transformation techniques to improve the speed of the program to make it suitable as a viable prototype.
4. Produce imperative programs that are implementations of the final functional program.

In step 2, the specification must be refined into a constructive representation because to quote Morgan [Morg94 pg 7] *"It can be proved that no computer, as the term is presently understood, can be built which could execute all specifications".* Many specifications can contain statements which are not implementable as they stand, as shown in the paper "Specifications are not (Necessarily) Executable" [Haye89]. Any method that seeks to take a specification containing such statements and then arrive at executable code using that specification as a base, must transform those statements in some way. Fuchs [Fuch92] criticises Hayes and Jones [Haye89] and argues that "Specifications are (Preferably) Executable". Furthermore he suggests that highly expressive specifications and executability are not mutually exclusive and that the examples given in [Haye89] can be made executable by adding a small number of constructive elements. Executable specifications generated in this way are transformations of their non-execuable counterparts, although they are slightly less abstract than the non-executable specification, but this is necessary for implementation purposes.

When the prototype, implemented as a functional program, is transformed to improve its efficiency, Johnson still advises a change to an imperative language. However, the author adds that in the future due to the advances in functional language compiler technology, this stage may be unnecessary in certain applications. Sherrel and Carver [Sher94] also state that the functional programming language Haskell will be usable for the construction of large software applications, when better compilers and improvements in computer architecture are made available. Utting [Utti95] states that even an inefficient animation is extremely useful for exploring the meaning of a specification. He adds that if more efficient animations are possible they may be used as a prototype of the final system, or even as the final implementation itself.

2.3.1 Z into Prolog(1)

Knott et al [Knot92] present a method of animating Z specifications using Prolog. The project developed a library of Prolog rules (the SuZan library) to match the built

in constructs of Z. Their method allows the Prolog code to execute at much faster speeds than other methods by carrying out program transformations at the Prolog level. An implementation of Morgan's telephone network in [Haye87] showed that the animation runs extremely slowly when translated into a functional language (in this case the animation took overnight to execute). A naive translation of the specification into calls of the Prolog library rules described in this paper, although faster than the functional form of Morgan's, still took three hours to execute. Some transformation rules such as placing the delta call of schemas at the end of the Prolog rules (instead of at the beginning as it appears in Z schemas) lowered the execution time to 70 seconds. Further work using some applications of the laws of Boolean algebra reduced the time to 0.65 seconds.

A major criticism of this work is that, for each schema, a predicate is built which generates all possible states defined by the signature part of the schema. The approach is completely impractical because often there will be no satisfactory answer to a query, due to a possibly infinite number of answers, or because the act of generating all possible states and testing them will take too much time. A simple example such as sorting the sequence [5,2,4,3,1] took 9 seconds with LPA MacProlog on a Macintosh SE/30 and a total of 11 seconds to verify that the solution was unique. The program generated all possible sequences containing the same elements as the sequence to be ordered and then chose the generated sequence (there should be only one) that had the correct order. Obviously this generate and test paradigm has the possibility to generate massive amounts of possible solutions through a combinatorial explosion. How long would the program take to sort even a small sequence of 100 numbers?

Another weakness of this method is that the system does not provide a means for handling quantifiers and generic schemas. The only schema calculus operators allowed are: negation ($\neg$), conjunction ($\wedge$), and disjunction ($\vee$) . These restrictions limit the expressiveness of the specifications that can be animated.

Finally, the author acknowledges that the variety of flow-modes allowable in the implementations of Prolog inhibits the portability of the SuZan library to other Prolog versions.

2.3.2 Z into Prolog(2)

West and Eaglestone investigate the animation of Z specifications using two approaches in Prolog.

1. The first method called 'formal program synthesis' arises from the fact that Z and Prolog are related in a mathematical way. The method realises this relationship by a direct translation of the Z schema into Prolog.
2. The second method is 'structure simulation' which results in a simulation of the Z schema by capturing its mathematical structure.

2.3.2.1 Formal Program Synthesis

This method consists of two parts. Firstly to express the higher order logic of Z as first order computer logic and secondly to turn the first order computer logic into Prolog. Some characteristics of schemas were identified in order to assess how a first-order, straight forward schema would be represented in Prolog. List operations in Prolog were used as a model for some set operations in Z. The method failed mainly due to the difficulty of deriving Prolog clauses from the first order representation of the Z specification. West et al concluded that, as yet, there is no suitable way of turning arbitrary logic specifications into logic programs and hence abandoned this approach.

2.3.2.2 Structure Simulation

This method required a 'flattening' of the Z specification into a form suitable for implementation in Prolog. A library of (finite) set theory operations was constructed using Prolog recursive predicates. These were used to animate two simple examples from [Haye87]. The advantages of the method were that the approach is simple and

the Prolog code is general so that it can be tailored to a particular Prolog implementation. However, this is clearly not an advantage in terms of the language used. Ideally Prolog code should work on all Prolog environments if the interpreters and the language conformed to a strict standard. The lack of data types in Prolog meant that Prolog sets had to be implemented as lists which was judged as a restriction on the subset of Z which is capable of being translated by this technique. In modelling the set as a list, the author does not say if the semantics of the set are preserved i.e. that it is finite, enumerable and no duplication of items are allowed. He does however say that the order of items in the list will be irrelevant. Another restriction of the technique involved some Z constructs whose suitability for implementation was unknown (at the time of writing the paper). These constructs included Lambda expressions and complex existentially quantified statements.

This approach has more merit than the approach of Knott et al, discussed previously, because the approach does not use the generate and test technique and so avoids combinatorial explosions. Instead data is directly input into the queries of the animation. Variables are instantiated with set values according to some test strategy to validate the specification and test the resulting animation.

2.3.3 Z into LISP

Morey et al [Morr92] present work on a project developing a method which enables Z specifications to be translated in an executable form in LISP. The more complex Z operations such as power set and distributed disjoint were prototyped in Miranda (Diller [Dill90] gives an example of animation using Miranda) to explore efficient designs before being coded in LISP. These were then tuned with specific LISP constructs. The project has concentrated on the functionality of Z schemas and hence on the implementation of their predicates rather than the declarations. The type of the variables were deemed to be outside the scope of the study. However, it is an aim of the project workers to extend the animation system to deal with a larger part of the Z language and to include the schema calculus operators and data typing. Morey criticises the implementation on the grounds of its inefficiency, but defends the

prototype by saying that once validated, program transformation techniques can be applied to speed up the implementation.

Morey et al identify the fact that a user would expect a prototype to support an on going state so that the post state values of one operation will become the pre state values of the next operation. However, in the example specification implemented by the method, the after states and output are defined as local variables in the functions implementing the operation schemas. Thus, there is no continuing state. This must make the animation difficult to use because the state must be set up individually for each test case because the after state of one operation test cannot be used in the next test.

This is one of the difficulties of using functional languages, which do not have the concept of a state, to implement state based specifications. The work of Goodman below attempts to address this problem without explicitly passing the state between functions.

2.3.4 Z into Haskell

Goodman criticises previous functional languages because they have been slow, and cumbersome. He also criticises attempts at animating or prototyping Z specifications with other functional languages because, the state has been handled clumsily, by being passed explicitly from function to function and input/output if handled at all has been done so with a series of continuations [Good95b]. He proposes the use of a Monad to simulate the action of input, output and state when implementing Z specifications using the functional language Haskell. This allows the programmer to deal with these features in a new way which is simpler than those used before. However, no mention is made of how to translate specifications that include multiple states, schema inclusion and schema bindings.

One benefit of using a functional language is that it is easier to reason about and formal proofs are more amenable than for an imperative language. However, incorporating the Monad introduces an imperative style of functional programming which then complicates matters with respect to reasoning and proving programs correct [Good93].

2.3.5 Critique of Rapid Prototying and Animations using Functional Languages.

The methods outlined above have the aim of showing that the formal specification is a valid representation of the customers requirements by providing a version of the software that can be shown to the customer at an early stage of development. The animation can be used to test operations and for reasoning about the specification. Indeed, quite commonly this is all they are used for because the code produced is not usable as a final implementation since its performance may not be satisfactory. The implementations are often acknowledged by the authors as being too slow. Another disadvantage is that many specifications are state based and the functional implementations do no handle the state well (although Goodman addresses this by moving the functional language closer to an imperative one, by adding some imperative features, through the use of a monad). Even carrying out an animation, without a continuing state (as in Morey), must be difficult as the state must be generated for each set of tests because the state obtained after an operation is not used in the next test.

Many functional languages do not have a standard implementation which would make it difficult to reuse the code in other applications. Also, the current state of compiler technology prohibits the use of functional programs in real world applications due to performance issues. However, many authors believe that functional programs will be in widespread use with future increases in compiler performance and advances in computer architecture. As Utting [Utti95] states, an

animation of a Z specification can be used in the final software provided it is efficient enough.

One other limitation with functional languages is the lack of facilities for interfacing with other languages and systems, such as networks and databases. Wadler [Wadl95] introduces one meagre (in his words) facility to enable the functional programming language Haskell to interface with C programs. At the time of writing, only values of base type could be passed to C. He states that this is a start in the right direction and current research topics include how to pass more complex structures between C and Haskell, how to add concurrency, how to integrate storage management and better support for graphical user interfaces. Once again, these are problems that are not faced when using a target language such as Ada.

Proponents of rapid prototyping using functional languages claim that the implemented code has the advantage that is closely matches the original specification, far more than is the case when imperative languages are used. The work outlined in this thesis has the advantage that the code produced is easily identifiable with the Z specification through the use of predefined function and procedure calls matching the Z operations, but because it has been constructed with an imperative language, the concepts of input, output and state are inherently available. The work of Goodman allows a functional language to be used in an imperative way, whereas the work described in this thesis allows an imperative language to be used in a more functional manner.

From the literature the following advantages of using functional languages to implement Z specifications can be summarised as follows :-

advantages

1. The prototype provides early validation of customer requirements.
2. A subset of Z is easily constructed in functional languages.

3. The code matches the original Z specification through similarly named function calls
4. The method provides an efficient means of testing model based specifications.
5. When using a functional language it is possible to make assertions about programs and prove these assertions to be correct.

disadvantages

1. Functional languages do not have the concept of a state, making it difficult to implement state based specifications.
2. Input and output are not handled well, particularly in the case of interactive input/output with purely functional languages
3. The resulting code is slow and often cannot be used as the final implementation.
4. The functional program must be translated into an imperative language if the prototype is not suitable for use.
5. The lack of a standard in many functional languages makes interfacing with other languages and portability difficult.

The work contained in this thesis has many of the advantages of using functional programs outlined above. However, with regards to point 2 and point 5, producing the code for the reusable components is more difficult and it is much harder to reason about imperative programs, although the use of abstract data types make reasoning about programs easier to do than when using the structures provided by the programming language [Haye96].

The work in the thesis uses a well established standard imperative language and therefore does not have the disadvantages inherent with using a functional programming language (such as state, input/output, speed and portability problems). Also, coding the specification in a high level imperative language allows for flexibility when integrating with operating systems, user interfaces and external modules [Jado89].

2.4 Bridging the Gap Between the Specification and Target Languages

The methods discussed in this section all seek to make the implementation process easier by refinement within the Z specification language, or by writing the specification in such a way that producing executable code is more easily achieved.

2.4.1 Structuring Z

Read [Read92] identifies the fact that Z has been used to good effect on large projects but it can be difficult to reuse parts of the system in future developments. The conventional way of writing Z specifications gives a good description of the behaviour of the whole system but does not define the behaviour of individual parts of the software. Read's paper is fundamentally about changing the way specifications are written. He suggests that specifications should be written in a style that allows parts of the system to be reused with more ease than at present. This can be achieved by improving the correspondence between Z and Ada at a structural level. Ada has several structuring mechanisms that have no equivalent in Z (i.e. packages, private types and tasks). Read shows how various classes of Ada packages can be modelled using Z. With these mechanisms in place a Z specification can be written in a more modular style with thought as to the final structuring of the Ada packages making up the implementation. The aim is to produce a specification that will have a one-to-one correspondence to the Ada packages that eventually implement it. In other words the formal specification must have the same structure as an Ada program. This means that parts of the system and their implementation can be more easily identified and plucked out for use in other specifications.

The main difference between Read's work and the work detailed in this thesis is that Read adds structure to Z using the structuring mechanism of Ada as a model. However, in this thesis, functionality is added to Ada by modelling the types and operations found in Z. Read uses Z to model service, restricted service, generic and state based Ada packages.

Read does not mention the process of taking a complete specification and arriving at code using only reusable components. In fact Read is not concerned about how a particular specification is turned into code at all, as long as the specification is modularised so that some components of the specification can be reused in other specifications. Read is concerned with facilitating the reuse of parts of an existing implementation in another system. The work presented in this thesis is concerned with arriving at the implementation.

A Z specification written using Read's method would undoubtedly be easier to translate into Ada code using the method presented in this thesis. This is because the Z specification will have been written in a structured manner and so the developer won't have to decide which states, types and operations should appear in which Ada package as would be the case in general. Read's method will have completed some design work at the beginning so that one does not have to be concerned with how the Z specification will map to the Ada implementation, one can simply translate each modularised Z component using the reusable Ada components. This will have advantageous results for scaling up the work to an industrial application because the industrial specification will be made up of small specifications, each concerned with a separate aspect of the systems functionality, which will easily map to Ada packages.

Lano et al [Lano92] also discuss the reuse and adaptation of specifications in Z and Object Z. Reusability of components has been seen as a central means to resolve the software crisis, but in order to reuse a component safely, there must be some formal description of its semantics. Lano et al identify Z specifications as potentially an excellent medium for reuse and adaptation since they combine a precise description of functionality with a design of the system. The paper addresses issues of how Z specifications can be structured and designed to facilitate reuse and adaptation.

Mitchell et al [Mitc94] also criticise the way in which formal specifications are written because the lessons learned from programming about effective

decomposition strategies are often not applied at the specification level. The authors conclude that care over decomposition in specifications is just as important as decomposition in computer programs. This is echoed by Sampaio and Meira [Samp90] who state that "*Although mechanisms for structuring and modularisation are more widely available in programming languages, they are vital to all phases of software development. Indeed, they are useful as a way to simplify any problem solving activity and software specification is no exception.*" Modularity reduces complexity, aids maintainability and increases system comprehension. Glass and Noiseux [Glas91] argue that modularising contributes far more to maintainability than structure does and is the most important factor in preventative maintenance, whilst Lientz and Swanson cite a study finding that 89% of code users reported improved maintainability with modular programming [Lien80].

A specification written with the methods described above should be more amenable to direct translation with reusable components. The structure of the Z specification will map to Ada packages easily because the Z specification will have been structured to specify coherent conceptual units in the application domain. The process of translating a large specification into code will then be equivalent to translating a series of smaller specifications each describing an aspect of the system into an Ada package. It is not necessarily the number of state schemas or the number of individual operation schemas, contained in a specification, that makes an implementation using reusable components difficult. It is the relationships between the states, operations and the packages housing them that can possibly cause difficulties if the specification is constructed in an ad hoc manner.

2.4.2 Transformation within Z

This section discusses those methods that work in the Z domain inorder to smooth the path from an abstract specification to executable code.

2.4.2.1 FunZ

Sherrel and Carver [Sher95] introduce FunZ which is described as an extension of Haskell with a Z like flavour, that preserves many of the notational conventions of Z. Software design with FunZ is similar to design with Z except that it provides a bridge between the Z specification and a functional implementation. Each step in the method has functional overtones to facilitate direct translation in a functional language. The Z specification is refined into a FunZ specification from which Haskell code can be derived. The paper uses a case study based upon the class managers assistant of Wordsworth [Word92]. Indeed, the whole process is very similar because Wordsworth develops a concrete specification using a retrieve function and data types found in an imperative language, whereas Sherrel and Carver develop the FunZ specification, again with a similar retrieve function, but targeting the types and operations available in Haskell. The work is based upon the fact that there exists a mapping between sets in Z and lists in Haskell. For example, considering lists without duplication, set difference, / , is mapped into range subtraction, ⩥, which in turn is mapped into list difference, // in Haskell. However, the paper does not describe the following important aspects.

- The example specification is simple and only uses Z operations which have direct counterparts in Haskell. No explanation of how operations in Z that have no counterpart in Haskell are specified in FunZ .
- Although the FunZ specification is now more readily turned into Haskell (as it is basically Haskell), no mention is made of how this is done with regards to the issues of input and output and state. The only examples shown are simple statements being translated into a FunZ equivalent. No information on how the final system works is given.

The work translates Z set operators into Z sequence operators, from which Haskell list functions are directly used to derive code. However, if Haskell supported the original set operations, then code could be translated directly, without rewriting the specification.

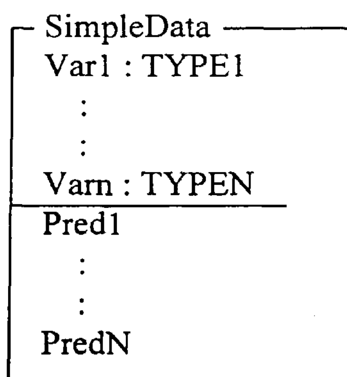
2.4.2.2 Anna

A similar method exists for linking Z specifications with Ada using Anna (ANNotated Ada) as an intermediate link [Woodw91]. Anna is a language extension to Ada that includes facilities for formally specifying the intended behaviour of Ada programs. The intention is to move from the abstract Z specification through refinement steps to arrive at a concrete Ada program. The method uses three notations during the development process :-

1. An implementation independent notation for the specification of the system (Z in this case).
2. An implementation dependant notation for the representation of a lower level specification (Anna).
3. A programming language for implementation (Ada).

The approach followed is to capture the system requirements in Z, refine the Z notation (for some number of steps) to both add detail to the specification and to bring the specification into a form suitable for transformation to Anna. Finally, the Anna specification may then be further refined and extended with Ada until the specification has been satisfied.

Some general rules for translating various types of schema are shown throughout the report, such as SimpleData :-



The variables in state schemas, such as SimpleData, are translated into functions as the corresponding Anna specification shows:-

```
Package SimpleData is

        function var1 return Type1;
        :
        :
        function varn return Typen;

        --| axiom
        --| pred1,
        --| :
        --| predn;

end SimpleData;
```

Where --| is the notation for an Anna annotation.

The approach of using functions to represent state variables has been chosen because the transformation works best when analysing the Z specification using the Anna specification analyser. However, it is clearly not a representation of what the Z specification intended. Var1 is a variable of type TYPE1, it is not a function returning an item of type TYPE1. The authors acknowledge this by saying that further work must be done to represent state variables as variables. A further criticism is that the method also fails when schema inclusion and the use of multiple states are required. Again the authors suggest that further work must be done to overcome these difficulties. The difficulties are not a result of moving from Z to Ada, as it will be been shown in this thesis (chapter 5) that state schemas, inclusion, schema types and bindings are easily translated when using reusable Ada components, but, the difficulties are caused by the requirement to use the Anna analyser.

In the paper, the only stage presented is the process of converting Z schemas into equivalent Anna specifications through the use of the Birthday Book case study [Spiv89]. In the state schema for Birthday Book, Known is defined as :-

Known : **P** NAME

In the Anna specification it is represented in the manner shown above as :-

```
Function Known return SET_OF_NAMES.SET;
```

Where SET_OF_NAMES is an instance of a package SET_CONCEPTS. The method translates operations by using generic packages such a SET_CONCEPTS and MAP_CONCEPTS which are assumed to be part of the Anna library, used by the Anna analyser, to show that the Anna specification is a valid representation of the Z specification. The Anna specification for AddBirthday is given as :-

procedure ADD_BIRTHDAY(ANAME : **in** NAME; ADATE : **in** DATE);
--| **where**
--| **in** (**not** SET_CONCEPTS.MEMBER(ANAME,KNOWN));
--| **out** (BIRTHDAY =

SET_CONCEPTS.UNION(MAP_CONCEPTS.MAPSTO(ANAME,ADATE),
in BIRTHDAY);

The development of this Anna specification to Ada code is not shown. In the appendices of [Woodw91], the original abstract Birthday Book specification is given with the corresponding Anna specification. However, the concrete version of the Birthday Book, also given in [Spiv89] is also presented, but the Anna specification for this concrete design is not given because, the authors say, it is not in a suitable form for translation due to infinite structures being used (Spivey developed the code using an (imaginary) infinite array structure to keep the example simple). This suggests that the Anna specification of the original Birthday Book specification is not used for the development of the Ada code at all. It is unknown why the authors were reluctant to provide an Anna specification for the new concrete design because

the Anna specification for the original Birthday Book specification also contains infinite structures. All that is required as Spivey says is to use schema calculus to specify a limit on the number of entries, with appropriate error reports if the limit is exceeded.

Finally, it is interesting to note that the Anna specification for Birthday Book is very similar to the actual code that would be derived if the operations contained within the reusable components described in this thesis were used. It is a shame that is it not possible to use the methodology described throughout this thesis to derive the Anna specifications (for reasons of using the analyser outlined above). If it were possible then code could be shown to implement the Z specification using the Anna analyser, but the code would be executable.

2.4.2.3 Z--

Valentine [Vale91] introduces Z--, which he describes as an executable subset of Z which can be treated as a typed functional programming language and executed reasonably efficiently. A Z specification is refined through successive steps and data refinements to arrive at high level Z--. Operational refinement moves the specification into base level Z--. The stages are highlighted by using Spivey's Birthday Book [Spiv89] as a case study. The description of Z-- is important with respect to the work presented in this thesis because Z-- ensures that all values it recognises can be stored in a finite computing machine. Whilst Z allows the finite and infinite data structures to be used in the same basis as each other programming languages do not (conceptually infinite structures can be defined in functional languages, but if a calculation is to be performed it must take place on a finite section of the data structure). Valentine transforms infinite sets, such as Known : $\mathbb{P}$ Name (from the Birthday Book) into alternate representations such as Known : $\mathbb{F}$ Name. This can then be held as an array, or in Z terms a sequence, with the range as the set. Valentine separates the use of sets into two concerns. Passive sets are those sets that are not altered and are only used for membership tests. These sets are left

unrefined in Z--. Active sets, are those sets that are enumerated and are therefore refined as sequences.

A concrete version of the specification is written replacing sets with sequences. The function Birthday : NAME $\twoheadrightarrow$ DATE is refined as follows :-

an ordered pair is declared	entry = = NAME × DATE
Birthday then becomes	BirthdayList : seq entry

One criticism of the work is that sequences allow duplication of items, where as duplication of items is forbidden in sets. BirthdayList, as declared above, allows duplicate Names, which was forbidden in the original specification. Also, the whole process is intended to refine the specification into a concrete form using sequences, from which an implementation can be created using a language that supports Z like sequences. However, if the target language supported sets, functions, relations and sequences then the transformation process to a concrete design based upon sequences would be unnecessary. This is the reasoning behind the construction of a series of reusable components written in Ada95 to model those types in Z and to allow a translation between the two. Valentine also identifies the fact that very large sets must be refined into some other data structure, apart from sequences, in order for them to be executed efficiently. This, presumably, would not be as easy to specify in the Z-- domain as no details of how to use more efficient structures are discussed. However, a specification written in Z-- will be more amenable to refinement with reusable components because it will be written in an executable manner.

2.5 Conclusion

No computer will be built that is capable of implementing all specifications. It is a fact that many specifications are produced that are not implementable as they stand. The refinement methods and calculi manoeuvre around these problems by constructing a design based on implementable constructs in a programming

language. Additional work is carried out to satisfy the non-functional requirements which do not appear in the original abstract specification such as performance, reliability, platform and storage means etc. The specification is heavily reworked and proofs are required to show that the new specification does not invalidate the original. When functional languages are used, some work must also take place to refine the Z specification. This work seeks to remove the non-executable statements by making them constructive, although non-functional requirements are often not considered because the intention is only to produce a prototype to validate the original specification. The work presented in this thesis also requires some work to be carried out in the Z domain, with regards to writing non executable statements in a constructive manner, removing non-determinism and considering performance issues before translation with reusable components can take place. A Z purist would argue that a specification only states a problem and does not offer a solution to implement it. This is a perfectly valid viewpoint and is the reason why every refinement method in the literature either adds information to the Z specification or rewrites it in some way in the process of developing code. When implementing a Z specification directly, it is necessary to blur this distinction between specifications and code by writing the specification in a functional manner, or transforming it to make it executable. A specification written in a functional style has the major advantage of being executable. The disadvantage is that some freedom in a specifiers expression is lost. However, to put this loss into context, Fuchs [Fuch92] points out that the main problem in software development is a lack of correctness and not the possible lack of expressive power in a specification language.

Refinement methods allow full formal development from specification to code in an imperative language, which can be used in an application. However, this method is impractical for all but the simplest specification and as a result very few proofs are carried out, although in many cases the cost of system failure does not justify the cost of full formal development. Here, usable code is produced but the method is, arguably, difficult to use. Animation or rapid prototyping of specifications using functional languages offers a more usable method, although less formal, but the

functional code produced is inefficient and is not often usable as a final system. The work presented in this thesis is comparably as easy to use as the functional methods because the translation process of Z predicates is similar, but the code produced is in a widely used, standard imperative language which has facilities to interface with other languages. The cost of producing the reusable components is higher than it would be if a functional language were used, but this can be offset because the functional implementation is often transformed into an imperative equivalent in any case.

Many authors have realised that Z specifications must be written in a more modular fashion to enable the lessons already learn't in software engineering for the construction of programs to be applied to the construction of specifications. At present, schema calculus incorporates many structuring facilities, but it does not modularise specifications. These methods will have advantages for the transformation of specifications to code using reusable components because the specification will map to an equivalent implementation in a linear fashion. A single module in a Z specification, i.e. a group containing its state, initialisation and operations, will be turned into a single package modelled as an abstract state machine. A specification that is modular will also enable each component specification to be tested against the informal requirements as it is implemented. The reuse of specifications will also be improved and a library of specifications can be built up from which other larger specifications can be constructed.

Chapter 3

Overview of Methodology

3.1 Introduction

The aim of this thesis is to show how Z specifications can be refined into code, which can be executed, using reusable components in Ada95. In this chapter a simple example of a Z specification of the Birthday Book [Spiv89] is given along with the Ada package specification and the Ada package body of the translation. A simple menu program is also highlighted which shows how the Birthday Book package is instantiated and how the operations are used.

3.2 The Birthday Book Specification

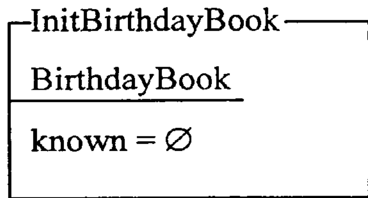
The Birthday Book records peoples birthdays and is able to issue a reminder when the day comes around. The system must deal with people's names and dates, however, for specification purposes the model for the names and dates does not matter. The sets of all names and all dates are introduced as basic types of the specification. These are :-

[NAME, DATE]

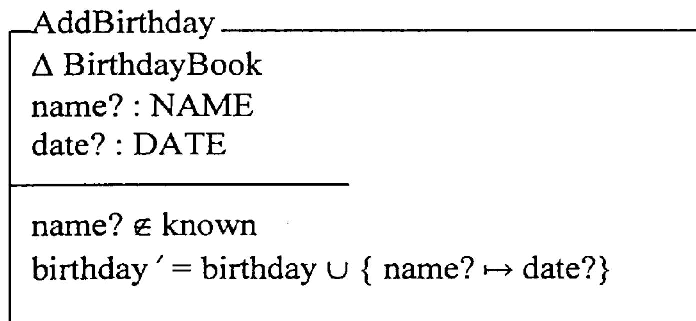
The Birthday Book state schema is as follows :-

```
┌─ BirthdayBook ─────────────────┐
│ known : P NAME
│ birthday: NAME ⇸ DATE
├──────────────────
│ known = dom birthday
└────────────────────────────────┘
```

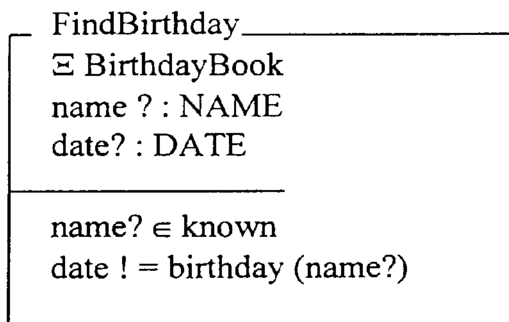
An initialisation schema specifies the state at which the system starts.



One operation of Birthday Book is to add birthdays to the system. It is specified as :-



Another operation could be to find the birthday of a person known to the system. It is specified as :-



3.3 Translating the Birthday Book State Schema

The state schema is translated into an Ada package specification modelling an abstract state machine. The state model is developed by instantiating reusable components according to the types specified in the state schema. This aspect is explained in detail in chapter 5. The operations specified in the Z specification require procedure or function definitions in the Ada package specification. The actual implementation of these Z specification operations are contained in the Ada

package body. The implementation of the Z operation schemas is discussed in detail in chapter 6.

The Ada specification for the Birthday Book is given below :-

```
with Many_to_one_G;Set_G;

generic
  type name is private;
  type date is private;


package b_book is

  procedure init_book;
  procedure add_birthday (n : name; d:date);
  function find_birthday (n : name) return date;
   --other operations in Z specification

private
    package name_set_pack is new Set_G(name);
    use name_set_pack;

    package date_set_pack is new Set_G(date);
    use date_set_pack;

    package birthday_map is new Many_to_one_G
                                     (name_set_pack,
                                      date_set_pack,

    use birthday_map;

    --the state model becomes
    known    : name_set_pack.set;
    birthday : birthday_map.map;

end b_book;
```

The Ada package body is given below. The details behind its derivation are explained in chapter 6.

```
package body b_book is

  procedure init_book is
  begin
    create_map(birthday);
  end;

  procedure add_birthday  (n:name; d:date) is
  begin
    if not is_a_member(n,known)
    then
      bind(n,d,birthday);
    end if;
  end;

  procedure find_birthday(n:name;d:in out date) is
  begin
    if is_a_member(n,known)
    then
      return (range_of(n,d,birthday));
    end if;
  end;

  -- other birthday book operations

end b_book;
```

The code given above is the Ada translation of the Z specification. As in the Z specification, no types are given for NAME or DATE. These are left as generic parameters in the Ada package. The code given above can be compiled, but in order to execute it, the birthday book package must be instantiated and used in another piece of software. This other software must provide the models for the generic types and the code for any generic procedures used. In this simplified example, there are no generic procedures used. In chapter 6 the use of generic procedures is explained in more detail. When the translated Ada package is instantiated the operations contained within the package become available for use. Other ancillary procedures can be included in the software using the translation, such as requesting input data. In order to instantiate the birthday book package the following code can be used, where its main body consists of a simple menu in order to call the operations available in the birthday book package (N.B. to cut down on the code in the thesis, this program is simplified and contains simple models, it does not for instance check that inputs are valid).

```
with b_book,text_io;
use text_io;

procedure birthday_main is

  --the options are the operations available to birthday book
  --quit program is used to end this program

  type options is (init_book,add_birthday,find_birthday,
                   quit_program);

  package options_io is new enumeration_io(options);
  use options_io;

  -- an integer package is instantiated because integer is used as
  -- a type in part of the model for DATE given below.

  package iio is new integer_io(integer);
  use iio;

  --the model for NAME
  subtype name is string(1..10);

  subtype day_number is  integer range 1..31;

  type months is (January,February,March,April,May,June,July,
                  August,October,November,December);

  package month_io is new text_io.enumeration_io(months);
  use month_io;

  -- a simple model for DATE
  type date is record
    day:day_number;
    month:months;
  end record;


  -- instantiation of birthday book package

  package birthday_book is new b_book(name,date);
  use birthday_book;

  --variables used in menu program
  n:name;
  d:date;
  ll:natural;
  choice : integer range 0..3;
  quit : boolean:=false;

   --simple procedure to request input of a date
   procedure get_date(d:out date) is
   begin
     put_line("enter day number 1..31 ");
     get(d.day);
     put_line("enter month eg July ");
     get(d.month);
   end;
```

```
begin --main part of menu program
  while not quit loop
    begin
      --print menu
      for i in 0..3 loop
        put(i,2);
        put("....");
        put(options'val(i));
        new_line;
      end loop;
      put("enter option : ");
      get(choice);skip_line;


     -- menu case statement

      case options'val(choice) is

        when init_book =>
          init_book;

        when add_birthday =>
          put("enter name ");get_line(n,11);
          put("enter date ");get_date(d);
          add_birthday(n,d);


        when find_birthday => put_line("enter name ");
          get_line(n,11);
          d := find_birthday(n);
          put(d.day);
          put("  ");
          put(d.month);


          -- other operations


        when quit_program =>
          quit := true;
      end case;

   end loop;

end birthday_main;
```

In the example above, the model for date does not check the number of days in a give month on input of a date. However, this simplified model was intended only as an example. It would be possible to use the model for date as given in the Ada package `Calendar`. The package Birthday Book is also reusable since many models for the given types can be used to instantiate the package.

3.4 Broad Overview of Method

- Examine the systems non-functional requirements to identify which reusable component to use for each of the types declared in the state schema, with regards to performance and method of data storage.
- Identify the Ada package hierarchy that will translate the Z specification (see examples in section 7.5.2.1 and section 7.5.3.3).
- Identify which Z operations will be contained in which Ada package.
- Translate state variables from state schema into an Ada package specification (see chapter 5) by using instantiations of the reusable components.
- Translate Z operations into Ada procedures or functions in the Ada package body (see chapter 6) using the operations contained in the reusable components.
- The translated Ada package(s) must be instantiated and used in the software system. This can be achieved when the system is built.
- Validate customer requirements using the complete software system using current techniques.
- Test the complete system against non-functional requirements using current techniques.
- If the system performance is not satisfactory then tune software (see section 7.3.5 paragraph 2).
- If the system performance is satisfactory and the system meets the customer requirements then it can be used in the final system.

Chapter 4

Construction of Reusable Components

4.1 Introduction

In order to translate Z specifications into Ada using reusable software components, it was necessary to construct reusable software components to model the main types found in Z; namely the set, function, relation and sequence. Each component was implemented as an Abstract Data Type using Ada95 and the Booch style [Booc87]. For each of these main types, a number of different components were developed based upon different data structures and using both internal and external memory to satisfy user requirements for efficiency and method of storage. The code extracts that appear throughout this thesis are written in Ada95.

The operations contained within each software component model the Z operations for that particular type. However, some combinations of Z statements occurred often and it was more efficient to create a separate operation to implement them. For example the statement $r \in ran\ F$ would involve applying the operation `range_of` to create the set of range elements for the function F and then using the function `is_a_member` to test the membership of r in this set. However the function `is_a_range_element` was included into function and relation components to model this situation using a single operation. A number of other operations have overridden counterparts so that the same operation can be applied on single items or sets of items and to either change an existing state variable or create a new variable. A full list of simple Z statements and operations are given in appendix 2, along with the Ada operation(s) used for their translation. Also contained in this section is a discussion on efficient data structures and a method of improving the productivity and efficiency of software components by the construction of utility packages. As an example, a set package is built on top of a package providing efficient external data storage, and then a function package is built on top of the set package. This method of construction also enabled software components modelling the Z operators that function over different types (such as distributed union) to be completed.

4.2 Outline of Reusable Components

This section describes the layout and structure for the Ada specification of a typical abstract data type, used to model a type in Z. Segments from the specification of a set are shown to highlight the main areas of interest. The data model is not given as many alternatives are possible (such as arrays, lists, trees and file structures etc.) and the operations shown are not exhaustive, in order to save on space. The generic formal part of a function package is also shown as an example because the other Z types (function, relation and sequence) require a set as a parameter to some operations. Here, with Ada95, the set can be imported as a generic package parameter, but, when using Ada83, the solution is not as concise (this is discussed in more detail in chapter 8 section 8.2).

It is possible for the set, function, and relation components to import an ordering function as a generic parameter, in order to improve the speed and efficiency of the implementation. It is valid to incorporate an ordering function for sets, even though a set is defined as having no order, because the order of items in the set is not available to the developer. The developer, when implementing the formal specification, can only access the set through the operations of the set abstract data type. The order is simply an underlying mechanism to improve the efficiency of the component and so the key abstraction of a set is not broken. In the case of the sequence component, there is no ordering function because the order of items in a sequence is of importance. Other components that use hashing functions or keys to achieve efficiency can also be constructed. In the case of a component utilising hashing routines, the component can be parameterised over a hashing routine supplied by the developer as required.

A typical set component is given as:-

```
generic

  type Item_Type is private; --allows a developer to instantiate set
  maxsize : in positive;     --over a given type and for a given
                             --number of items in the set
package Set_bounded_G  is

 --the type set is private so that the abstraction is safe

 type Set is private;

  --standard set operations
  procedure Create_Set  (S     :out Set);
  procedure Insert      (Item  :in Item_Type;     S  : out Set);
  procedure Union       (S1,S2 :in Set;           S3 : out Set);
  procedure Intersection(S1,S2 :in Set;           S3 : out Set);
      :
      :
  function Is_Equal_Set (S1,S2 : Set) return boolean;
  function Size_Of      (S1    : Set) return natural;

  -- Identity functions (used in quantifiers and set comprehension)

  function Ident        (I1:in Item_Type) return Item_Type;
  function Ident        (I1:in Item_Type) return boolean;

  -- Generic functions

  generic
    with function Predicate  (I1 :in Item_Type) return boolean;
    with function Expression (I1 :in Item_Type)
                                 return Item_Type is Ident;
  procedure Set_Comprehension(S1 :in Set;S2 : in out Set);

  --see section 6.7 for detailed explanation of set comprehension

  generic
    with function Post_Predicate(I1:in Item_Type) return boolean;
    with function Pre_Predicate (I1:in Item_Type)
                                  return boolean is IDENT;
  function There_exists(S1 : Set) return boolean;

  -- see section 6.6.2 for detailed discussion of There_exists

  generic
    with procedure Process(I1:in Item_Type;Continue : out boolean);
  procedure Iterate(S:Set);

  --see section 6.7.4 for detailed discussion of Iterate

  -- Exported Exceptions (see section 4.2.1 for use of exceptions)

  Set_Is_Full          : exception;
  Already_In_Set       : exception;
  Item_Not_In_Set      : exception;

  private
     -- DATA MODEL (this is not shown as many are possible)

 end Set_bounded_G;
```

The interface and some operations for a typical function component are as follows :-

```
with Set_bounded_G;

generic
  MSIZE:in positive;
  with package dset_pack is new Set_bounded_G(<>);--set of domain
  with package rset_pack is new Set_bounded_G(<>);--and range types


package One_to_one_G is

  type Map is private;

  procedure Create_Map(M:out Map);
  procedure Bind( D : dset_pack.item_type;      -- M′ = M ∪ {D ↦ R}
                  R : rset_pack.item_type;
                  M : in out Map);

  procedure Domain_Restrict(DS : dset_pack.Set;  -- M ′ = DS ◁ M
                            M1 : in out Map);

      :
      :

end One_to_one_G;
```

4.2.1 Use of Exceptions Within ADTs

Z was not designed to be executable and as a result it has no formal semantics for exceptions. However, since executable code is the goal, a design decision was made to incorporate exceptions into the reusable components. If attempting an operation would break the abstraction (such as attempting to add a duplicate to a set) then a message, in the form of an exception, is passed back to the application using the abstract data type. The application may then respond to the exception by repeating a process, trying an alternative, repairing the problem or even ignoring it. Exceptions can also be used to ensure that the system does not crash under certain circumstances, for example, when using dynamic memory allocation in a linked list. In dynamic data structures, it may be possible to use up all of the systems resources when adding data to the structure since computer memory is finite. In Ada operations that add data to the dynamic data structure, an exception can be raised

from within the component when the systems storage capacity is reached. Z allows infinite data structures, finite computing machines do not.

The exception `item_already_in_set` is raised in the Ada set operation `insert`. This operation models the statement $S' = S \cup \{I\}$, which is in effect made up of two statements, i.e. create a singleton set and union it with the set S. Since this statement is very common in Z specifications, a single operation was created to model it. When implementing the `insert` operation in an ordered set, the position of the item must be found. Having found the position of the item ready for insertion into the data structure, it is a very simple matter to determine if the item is already present and raise an exception if so. It is very common practice for an Ada package specification to contain a declaration of the exceptions that can occur in the package [Skan88 pg443]. An exception is not necessarily the result of an error in the program, it may be an event that happens rarely. By raising an exception within the package the user is given the opportunity to take appropriate action. Raising an exception here, also follows the style of Booch [Booc87] where exceptions are specific to a particular abstraction and may be raised if an operation is applied that may violate an object's integrity. The reusable components are designed to be usable in any implementation (not just for translating Z), and therefore it was deemed important that exceptions were included. Having said that, a Z specification should have a precondition (such as $I \notin S$) to check that one of the main abstractions of the set is preserved (i.e. that no item is duplicated) before a postcondition that adds an item to the set is reached. If this is true, then the exception `item_already_in_set` will not be raised. This is because the `insert` operation containing the exception will only be called if the item is not in the set. When translating a Z specification, the exception `item_already_in_set` can only be raised if an attempt is made to add a duplicate item and there is no Z precondition present in the specification which disallows it.

When implementing an unordered set component, checking for a duplicate item as part of the `insert` operation is not as efficient as with the ordered set component. In the unordered case, the `insert` operation may simply add the item to the end of the

data structure and checking for duplicity would require a traversal of the whole structure (in the worst case). It would be possible to write an Ada operation for inserting an item into an unordered set without first checking that it is not a duplicate and rely on a Z precondition to check first. This would mean that the operations from an abstract data type must be combined in a manner that preserves the abstraction (i.e. that `is_a_member` is used before `insert`, which is used only if `is_a_member` returns false.). This would then break a key concept of abstract data types, because, any order of operations should be applicable without breaking the abstraction. If the definition of the abstract data type is preserved, then the operation to insert an item should check that it is not a duplicate. As stated earlier, this is very easy when implementing `insert` in an ordered set component.

For the reasons above, a design decision was made to follow the Booch [Booc87] style of exporting exceptions rather than passing back a status parameter when an operation would possibly violate an objects integrity. The advantages of this were that error detection and recovery could be localised through the existing Ada exception handling mechanism.

4.3 Construction of Efficient Data and File Structures

Packages based upon trees have been constructed which use both internal and external storage, to provide efficient data structures. The tree package based on internal storage uses a tree with a balance condition to ensure that the height of the tree is always O (log n). This guards against cases where data is not random and could lead to a tree of height n at worst, which would result in an expensive list. The tree based upon external storage is an implementation of a B-tree [Baye72a,Baye72b,Weis93]. Accessing information from a disk is much slower than retrieval from main memory, and is the crucial factor in the performance of operations on the tree. The B-tree is organised so that each node in the tree contains n keys. When searching the tree an n+1 way decision is made in order to determine which direction to branch. This greatly reduces the number of disk operations. For example, a B-tree with a branching factor of 101 (i.e. 100 keys in each node) enables

any key in a tree containing one million keys to be found in at most two disc accesses if the root node is kept in main memory or three otherwise. These tree packages have been used as utility packages in the construction of two other set packages, and in keeping with the principles of reuse, and building utility packages upon other existing software packages, two more function packages have been built using the two set packages mentioned here.

4.4 Construction of Utility Packages Using Layering

It is possible to create software components which use the structure of trees or B-trees in the private parts of their Ada specification. Operations such as `insert` and `domain_restrict` for example would be implemented by direct manipulation of the underlying arrays, file structures and pointers contained within the specification. However, this would be complicated for many operations. The approach followed for this project was to implement the tree or B-tree as an abstract data type with enough primitive operations to facilitate the construction of other components as utility packages. This would improve the production of components by building them on top of existing components to make the most of reuse. The productivity is improved because the complexity is reduced by operating at a higher level of abstraction.

The crucial factor in the construction of utility packages is the generic iterator procedure which provides a mechanism for traversing the underlying abstract data type. The following code excerpts are taken from the completed versions and describe how a one to one function package was built upon a set package which was in turn built upon an underlying B-tree package.

The set is simply an instantiation of the B-tree package. The semantics of the set operations are constructed by appropriate use of the primitive operations available in the B-tree package.

```
package Set_bounded_G is
-----

 private
   package btree_set_io is new btree_G (msize,
                                        degree,item_type,lt_item);

    use btree_set_io;

   type set is record
     the_set : btree_set_io.btree;
   end record;
end Set_bounded_G;
```

An operation to open the set stored on file simply calls the appropriate primitive routine from the B-tree package. A list of possible exceptions for the B-tree will export exceptions relevant to the set.

```
procedure open_set(s:in out set;set_title,spaces_title:in string)is

begin
   btree_set_io.open_btree(s.the_set,set_title,spaces_title);

   exception
     when btree_set_io.btree_status_error  =>
       raise set_status_error;
     when btree_set_io.btree_name_error =>
       raise set_name_error;
     when btree_Set_io.btree_already_open =>
       raise btree_already_open;
     when others =>
       raise error_opening_set;
end;
```

A simple operation such as inserting an item into the set will call the primitive operation `insert` from the B-tree package. An exceptional condition such as attempting to insert an item that is already in the B-tree will result in an exception. Again, the exception can be caught here to export an exception relevant to sets.

```
procedure Insert(Item:in Item_Type;S:in out Set) is
 begin
   btree_set_io.Insert(Item,S.The_Set);
   exception
     when already_in_btree => raise already_in_set;
 end;
```

A more complex operation such as union will require the use of the iterator procedure from the B-tree package to traverse the B-tree structure. The union procedure takes two sets as input and places the union of the two sets into the first. The procedure iterates through the set s2 (contained in a B-tree) and checks whether each item in the set s2 is already in the set s1. If the item is not in the set s1 then it is inserted into s1 to form the union of the sets.

```
procedure UNION(S1:in out Set;S2:in Set) is

    Procedure Union_Process(Item:Item_Type;Continue:Out Boolean) Is
    begin
      continue:=true;
      if not Btree_Set_Io.Isin(Item,S1.The_Set)
      then
        Insert(Item,S1.The_Set);
      end If;
    end Union_Process;

    Procedure Union_Btree_Iterate Is New Btree_Set_Io.
                                          Btree_Iterate(Union_Process);

begin
   Union_Btree_Iterate(S2.The_Set);
end UNION;
```

A one to one function can be modelled as a set of ordered pairs. The one to one function can therefore be built by an instantiation of the sets package described above. The item type for the set must be an element that has a record structure containing domain and range fields.

```
package Bt_map

-----

   type element is record
      Dom :domain;
      Ran :Rainge;
   end record;


   Package Btree_Set_map_Io Is New  Set_bounded_G
                                (Msize,Degree,Element,Lt_Element);


   type Map is record
      A_Map: Btree_Set_map_Io.Set;
   end record;
```

A simple operation on the function would call the appropriate function from the instance of the set package.

```
function IS_EQUAL_MAP(M,M1:MAP) return boolean is
begin
    return IS_EQUAL_SET(M.A_MAP,M1.A_MAP);
end;
```

Complex operations will require an instantiation of the iterator procedure from the set package in the same manner as described earlier for the union operation. Each operation in the function package can be created via the operations available in the set package.

This method of construction allows software components that are based upon very complex structures to be completed with relative ease and in a fraction of the time. Increasing the degree of reuse of components by building on top of existing components has an effect on the productivity of software and should increase the quality of the software. However, there is no empirical evidence that this layering approach improves quality. A study by Zweben et al [Zweb95] concludes that there is certainly no loss of quality when layering is used and any loss in performance is minimal if the proper abstract functionality is encapsulated in the component. In the example shown above, the proper functionality certainly is encapsulated in each of the components. This is due to the fact that the main structure in Z is the set and each of the other structures depend upon sets.

4.5 Construction of Z Operators Using Utility Packages

It has been shown how components can be built upon other abstract data types to take advantage of the performance aspects of the underlying type. In this section, a utility component is shown which can house Z operations that operate over different set types. The distributed union operator has been used as an example, and its specification and the code for its body are given. The distributed union operator takes a set of sets as an input parameter and outputs a set of items.

It is defined by $\bigcup : \mathbb{P}(\mathbb{P}X) \rightarrow \mathbb{P}X$.

This operation is complicated by the fact that it operates upon sets of different types. Other operations that mix types such as relational composition can also be included in a utilities package. A version of the set comprehension procedure can also be placed in a utilities package to allow sets based upon different types to be created (see chapter 6 section 6.7). The following specification is complete for the distributed union operator.

```
with Set_bounded_G;

package Set_Utilities is

 generic

  size : natural;
  with package set_pack is new Set_bounded_G(<>);
  with package set_of_sets_pack is new Set_bounded_G
                                  (set_pack.set,size,gt_set);


  procedure dist_union (ss: in     set_of_sets_pack.set;
                        s : in out set_pack.set);


  --- other set utility operations

end;
```

This package imports a set package and another set package instantiated over the first to create a set of sets. The body is implemented by using nested iterators. The first iterator traverses the set of sets. Each set in the set of sets is then taken in turn, and the inner iterator is used to traverse each of them. Each item can now be taken and inserted into the output set which contains all the items present in the set of sets. The iterator procedure in this set package contains code that visits each item in the set until the last item in the set is reached or until the parameter `continue` has been set to false. In this case `continue` is always set to true because every item in each set in the set of sets must be visited.

The implementation for distributed union is as follows :-

```
package body Set_Utilities is

  procedure dist_union(ss:set_of_sets_pack.set;
                      s:in out set_pack.set) is
      continue:boolean;

      procedure process_set (constituent_set: in set_pack.set;
                                              continue:out boolean) is

         procedure process_item (i:in set_pack.item_type;
                                              continue:out boolean) is
         begin
           if not set_pack.is_a_member(i,s) then
             set_pack.insert(i,s);
           end if;
           continue:=true;
         end;

         procedure iterate_through_set is new
                              set_pack.iterate(process_item);

      begin -- process_set
        iterate_through_Set(constituent_set);
        continue:=true;
      end;

    procedure iterate_through_set_of_sets is new
                set_of_sets_pack.iterate(process_Set);

  begin -- dist_union
     iterate_through_set_of_sets(ss);
  end;

  ---- other set utility operations

end Set_Utilities;
```

4.6 Testing the Reusable Components

Reusable components require more testing than any other component [Barb94], since errors will spread to all software using the reusable components and confidence will be destroyed if an error is found in a 'safe' component. It would be possible to mathematically prove that each of the operations in the components preserve their respective Z semantics. However, proving programs correct is a very difficult and time consuming process. For the purposes of this research project and due to the time constraints involved, the main concern was in actually modelling the

Z types and operations using Ada and the translation of Z specifications into Ada code. The testing of components for this research project has not proven that they are free from errors, as the type of testing completed for this project can only demonstrate that the program is free from errors for the tests that were carried out.

The reusable components were tested in two ways:

1. The reusable components were tested by writing an event loop, with a menu, to test each operation in the components. This has been done for all the components to structurally test their correctness and usefulness. The approach is known as 'white box' testing [Basi87] since the source code is inspected and test cases are found to test each path through the operation for a variety of data. As the complexity and size of each operation is small complete coverage of the paths through the operation can be obtained for each operation in each component.

2. The abstract data types were also specified using an algebraic (or axiomatic) specification. Test programs were then derived from the semantics of the operations in the algebraic specification to test the validity of the operations in the reusable components. A technical report for the algebraic specification of the abstract data type set is given in [Hayw94]. A version of a bounded set component used to refine Z specifications is given in [Bale94] and a simple test program for the semantics is included.

The advantages of creating algebraic specifications for the abstract data types are:-

- Each operation has a well defined outcome for any data.
- The axioms can be used to test the software component.
- The ADT can be shown to be complete i.e. combinations of operations produce well defined results.
- Further results concerning the application of operations can be proved.
- The axioms contained within the Ada test program are independent of the data model.

A software component that implements such an algebraic ADT can be used with a high degree of confidence in its behaviour, since it effects are clear, its logical foundation is sound and it has been tested against its axioms. This method is akin to 'black box' testing where test data is constructed from the specification.

Chapter 5

State Schema Refinement

5.1 Introduction

This chapter discusses the method and details for the refinement of the state schema. An example of the Birthday Book [Spiv89] is given although it has been extended to include a higher order relation to show how complex types can be translated. A second state schema for a petrol station is refined to reinforce the method and to enable some of its operations to be used as examples in the next chapter. Other issues affecting design decisions are discussed and a third example departmental database specification, is used to highlight these areas. The areas include the refinement of schemas with multiple states, the use of schemas as types and the binding notation (θ).

5.2 Methodology

The state schema is refined into a generic Ada package specification by a series of instantiations of the reusable software components which model any sets, functions, relations and sequences found in the state schema. These create the state space upon which the operational schemas can act. The operations of the specification are translated and appear in the Ada package body (see chapter 6). A specification based upon a named state is translated into an Ada based Abstract State Machine (ASM). The parachuted or given types of the Z specification become generic types to the Ada package to give a developer control over their final format. This information is not present in the Z specification, but poses no problems for translation since the actual model to implement each type can be deferred to the package or system that uses (and therefore must instantiate) the translated package. Each of these given types will require an instantiation of a reusable sets package to create a set for each type. A complication occurs when it comes to modelling functions and relations because some of their operations can be parameterised over a set of the domain and range types. It is necessary, therefore, to instantiate set packages for the domain and range types before the function or relation statement in the schema can be modelled and instantiated as a package. A similar strategy exists for sequences, but in this case

the domain set of the sequence is the natural set, whilst the range set is a set of the item type making up the sequence. In some specifications, where functions, relations, sequences and combinations of these types are involved, the order of instantiation is important. This is because the instantiated package of one type can be used in the generic parameter list of another type (refer to function example in section 4.2). However, the order of instantiation is not difficult to determine because it is not possible to instantiate a package A that depends upon other packages B and C if the packages B and C have not been instantiated themselves. So, for example, when instantiating a function package, the set packages for domain and range types must be instantiated first. The instantiation of the function can then use these packages as generic package parameters.

5.3 Refinement of State Schemas

The rule for the refinement of the state variables contained within the state schema is that each different type introduced must have a set package instantiated for it, if it is used as a power set or in a function, relation or sequence declaration. Items that are declared as power sets require a set package instance in order to hold the state information. A minor refinement is made so that sets of type $\mathbb{P}$ are implemented as sets of type $\mathbb{F}$. The other types also depend upon sets since functions, relations and sequences require sets of domain and range items. Implementations using bounded data structures will require some extra information about the bounds.

5.3.1 Relations and Functions

In this section, relations and the various types of function are examined along with which type of reusable component should be used for each. For the discussion below, a set of domain items is known as the source set, whilst the set of range items is known as the target set.

5.3.1.1 Relations

The relation $X \leftrightarrow Y$ is defined in [Spiv89] as $X \leftrightarrow Y == \mathbb{P}\ (X \times Y)$.

Relations are modelled by instantiating a <u>many to many</u> map package.

5.3.1.2 Partial injections.

These are defined in [Spiv89] as :

$X \rightarrowtail\!\!\!\!\!\!\!\!\;\mapsto Y == \{ f : X \rightarrow\!\!\!\!\!\!\mapsto Y \mid (\forall x_1, x_2 : \mathrm{dom}\, f \bullet f(x_1) = f(x_2) \Rightarrow x_1 = x_2) \}$

This definition states that each member of the domain maps to a different member of the range. Functions of this type are therefore modelled by instantiating a one to one function package.

5.3.1.3 Partial functions

These are defined in [Spiv89] by:-

$$X \rightarrow\!\!\!\!\!\!\mapsto Y == \{ f : X \leftrightarrow Y \mid (\forall x : X; y_1, y_2 : Y \bullet$$

$$(x \mapsto y_1) \in f \wedge (x \mapsto y_2) \in f \Rightarrow y_1 = y_2) \}$$

This definition says that each member x in the domain of the function must relate to one member y in the range. However, it allows for duplicated range items provided that each of the duplicated range items has a different domain item. Functions of this type are modelled by a many to one function package.

5.3.1.4 Other Types of function.

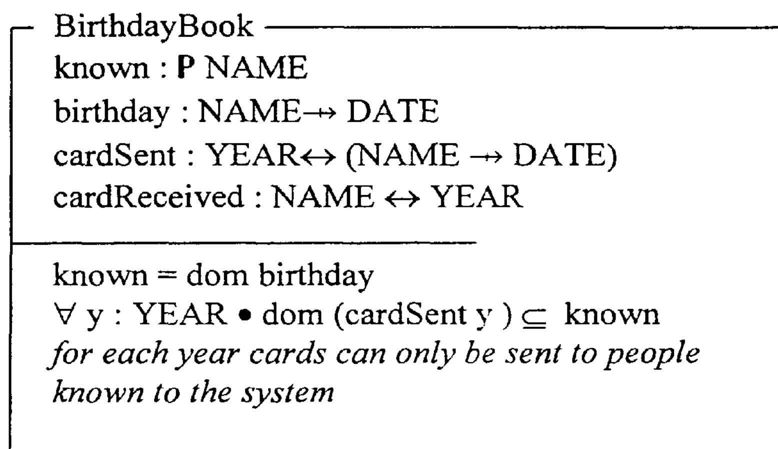
The functions described above were partial functions and partial injections. The other types of function are subsets of the main types given above. They are:-

- total functions - The domain of the function is the source set.
- total injections - An injective function which is also total.
- partial surjections - A partial function for which the range of the function is the target set.
- total surjections - A partial surjection that is also total. i.e. the domain of the function is the whole of the source set and the range of the function is the whole of the target set.
- bijections - An injective function that is both total and surjective.

A minor refinement is made where each of the types of function are created by instantiating a map component according to its basic type. Therefore all injections are modelled by instantiating a one to one package. All partial functions are modelled by instantiating a many to one package. The flavours of function effectively amount to whether the source or target sets cover the domain or range of the function.

5.3.2 The Birthday Book Implementation

The state schema for the Birthday Book specification is given as :-



The state schema is refined into a generic Ada package specification, where the parachuted types [NAME,DATE,YEAR] are generic parameters to the package, so that the developer has control over their final format. This information is not present in the Z specification, but the final format of the models used for each type can be determined by the non functional requirements of the system to be built. The following code segments are the instantiations of set packages for the parachuted types.

```
package Name_set_package is new Set_bounded_G (a_name,-------);
package Date_set_package is new Set_bounded_G (date,-------);
package Year_set_package is new Set_bounded_G (year,------);
```

The dotted lines in the generic parameter list above and throughout this section are intended to show that it is possible to include an ordering function or a hashing function (depending on the construction of the component) in the generic parameter list in order to improve the efficiency of the software, as discussed in chapter 4.2.

Each function or relation is essentially a mapping from a set of domain elements to a set of range elements, therefore, a set package of domain and range types is made generic to each package modelling them. In the example below, the set packages have been shown in bold type. The type Birthday is created by instantiating a many to one function as follows:-

```
package Many_names_to_one_date is new Many_to_one_G(
            Name_set_package,       --set package of domain types
            Date_set_package,       --set package of range types
            ------ );               --possible ordering functions
```

The relation cardSent : YEAR $\leftrightarrow$ (NAME $\twoheadrightarrow$ DATE) is a higher order relation. A many to many map package can be used, with the year as the domain and the partial function NAME $\twoheadrightarrow$ DATE as the range. A set of YEAR has already been instantiated. In order to instantiate the map package, a set of range items (of type NAME $\twoheadrightarrow$ DATE) is also required. This is done by utilising another set package instantiation. The item type for this set package will be the type `Many_names_to_one_date`, which is the NAME $\twoheadrightarrow$ DATE function instantiated previously. This enables the higher order relation to be created.

```
package Set_of_many_names_to_one_date is new Set_bounded_G(
      Many_names_to_one_date,        --Package of functions
      -------);                      --ordering functions

package Many_years_to_many_names_to_one_date is new Many_to_many_G(
      Year_set_package,
      Set_of_many_names_to_one_date, --set package of functions
      ------- );                     --ordering functions
```

The relation cardReceived : NAME $\leftrightarrow$ YEAR uses a straight forward instantiation of the many to many map package.

```
package Many_names_to_many_years is new Many_to_many_G(
        Name_set_pack,
        Year_set_pack,
        -------);
```

The state model presented in the private part of the Ada package specification becomes :-

```
        Known         : Name_set_pack.set;
        birthday      : Many_names_to_one_date.map;
        cardsent      : Many_years_to_many_names_to_one_date.map;
        cardreceived  : Many_names_to_many_years.map;
```

The type `Known` is redundant in this example, because it exists as the domain of the birthday function and can therefore be removed from the implementation. The other alternative is to keep `known` in the system and duplicate the data. Situations such as this can be common in Z specifications and the implementation of these situations can depend upon the available storage and speed requirements of the system. Duplicating data may speed up some operations in the system, but at the expense of its memory.

5.3.2.1 The Package Interface

Each of the instantiations that are used in the creation of the state space contain a number of parameters such as the type, and possibly size and ordering functions etc. These parameters must be declared as generic formal parameters to the package. The interface for the birthday book package is as follows:-

```
with Set_bounded_G,Many_to_one_G,Many_to_many_G;

Generic
      Map_size :  in positive;           --if bounded function
      type a_name is private;
      type date   is private;
      type year   is private;
      with procedure out_name(n:a_name); --procedures to output
      with procedure out_date(d:date);   --types if output required.
      with procedure out_year(y:year);   --in Z specification

      ---- possible ordering functions if ordered sets are used.
      ---- Sets are of type NAME, DATE and YEAR, so there would be
      ---- one ordering function for each.

package birthday_book is

      ----- available operations

      ----- exported exceptions

private
      ----- package instantiations creating state variables

      ----  the state model

end;
```

If unbounded components were used, the Birthday Book could grow dynamically and in this example `Map_size` would not be required as a generic parameter. The Z specifications ! notation does not contain information as to the type of output required from the specification. The generic output procedures allow this choice to be deferred to the actual package that instantiates the Birthday Book package. These output procedures can be constructed, by the developer, to take non functional requirements into account, such as outputting to a terminal, file, printer or using the output as the input to another routine.

The exported exceptions are used to implement operation schemas that have been made total by specifying what happens under certain 'error' conditions. This use of exceptions is discussed in section 6.6.1.

Functions for equality do not have to be included as generic parameters in the software components because they each export a private type. Ada semantics

therefore include the predefined operations for testing equality (and inequality) of types.

5.3.3 The Petrol Filling Station

The state schema for the Filling Station specification is specified in [Norc91] and given below. The set waitingToLeave models the situation where a car has filled up with petrol and the owner is about to pay. The car is finally removed from waitingToLeave by the operation LeavesHappy (not specified here). This operation enables the car to visit the petrol station again.

[Pump, Car]   The parachuted types are Pump and Car

FillingStation

queues : Pump ⤔ seq Car
waitingToLeave : **P** Car

∀ p : dom queues •
  #(queues p) = # (ran (queues p)) ∧
the length of a queue at that pump is equal to the number of different cars in the queue
  waitingToLeave ∩ ran (queues p) = ∅
a car cannot be at a pump and serviced

∀ p1,p2 : dom queues | p1 ≠ p2 •
  ran (queues p1) ∩ ran (queues p2) = ∅
a car can only be at one pump

The filling station has two given types namely pump and car. These will each require an instantiation of a set package. The type 'queues' is a partial injection, therefore a one to one function component is used. The domain of the function is the set of type 'Pump', whilst the range is the set of type 'Seq car'. An instantiation of a sequence package over the type car is therefore required. This can then be used as a parameter in a set package to create a set of the sequence of cars. The set of cars and set of pumps must be created first.

The car set is more suited to an unbounded set whereas the number of pumps in a filling station is small, finite and fixed so a bounded set package can be used. They are instantiated as follows:-

```
package car_set_pack is new  Set_unbounded_G(car, ---);
package pump_set_pack is new Set_bounded_G  (pump,max_no_pumps,---);
```

The sequence package is instantiated next, as it uses the set of cars as a parameter.

```
    package car_seq_pack is new Sequence_bounded_G(sequence_size,
                                          car_set_pack);
```

A set of car sequences is required for the range of the 'queues' function.

```
      package set_of_car_Sequences_pack is new Set_bounded_G(
                                          car_sequence,
                                          max_no_pumps );
```

The function 'queues' can now be instantiated. The domain type is the set of pumps package, whilst the range type is the set of car sequences package.

```
      package Queues_map is new One_to_one_G(max_no_pumps,
                                          pump_Set,
                                          set_of_car_sequences_pack
                                          -----);
```

The state model in the Ada package specification becomes :-

```
      queues              : Queues_Map.Map;
      waitingToLeave      : Car_Set_pack.Set;
```

5.4 The State Invariant

The state invariant describes a set of rules that must always be true of the state before and after operations are applied. A well written specification document should contain proofs to verify that the invariants are preserved by the initial state and that any state changing operations never violate the state invariants [Good95b]. If state changing operations never violate the state invariants, then the state

invariants will not require implementation in the Ada package specification modelling the state schema. As part of the established strategy for writing Z specifications, listed in [Bard92], it is recommended that the preconditions of partial operations are calculated. The description of the abstract operation is checked to ensure that the precondition is explicit in the operation schema; if not, it is added. Explicit preconditions should be used in operation schemas so that when any state changing operation is translated into Ada, its application will not violate the state invariants of the Z specification.

5.5 Multiple States and Inclusion

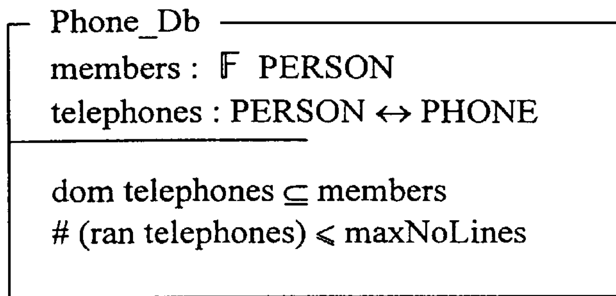
Many specifications are written which use schema inclusion. This can be in the form of state inclusion or operation promotion. It is vitally important, therefore, that using reusable software components for refining Z specifications succeeds with this type of specification. The specification described next contains a number of systems as specified in [Dill90]. The first system is a phone database that relates people to phone numbers. The second is a departmental database that relates people to departments and to rooms. Finally a third database extends the phone database with the departmental database, to create a full staff database. This section shows how child packages can be used to implement specifications that involve reuse and have multiple states. The work discussed in chapter 2 on refinement using functional languages eg. [Knot92,Morr92] has problems with this type of specification. Even the work of Goodman [Good95b,Good93] which handles the state in an 'imperative' manner through the use of a Monad does not mention how multiple states and state schema inclusion would be implemented.

5.5.1 The Phone Database State Schema

This database relates people to their phones.

[PERSON,PHONE] These are the parachuted types

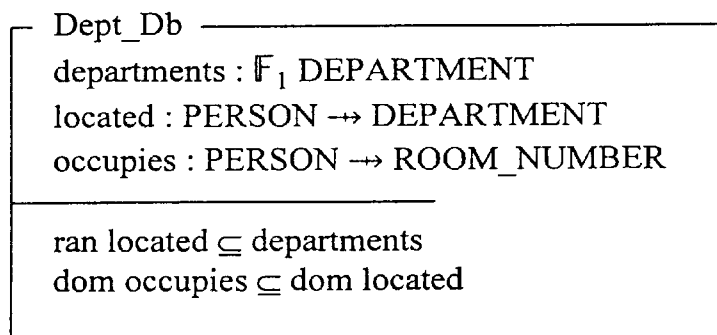
| $MaxNoLines : \mathbb{N}_1$



5.5.2 The Departmental Database State Schema

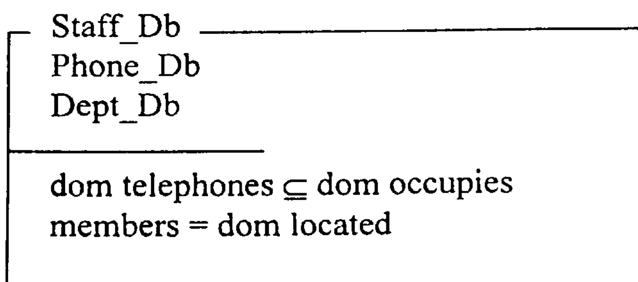
This database relates people to departments and room numbers.

[DEPARTMENT,ROOM_NUMBER]



The statement *ran located ⊆ departments* is included in the schema Dept_Db because operations exist that can modify the departments, by for instance merging two existing departments. The full staff database extends the phone database with the departmental one.

5.5.3 The Staff Database State Schema



This schema includes the Phone_Db and Dept_Db state schemas. Purely departmental operations could be defined for Staff_Db or defined for Dept_Db and then promoted to the full system. There are three possible ways of implementing this system :-

1. The private types would have to be made non private, so that they become visible to the package sharing them. This would allow the client package to have access to the underlying structure also, and would break the abstraction, resulting in a loss of information hiding. This should be avoided as information hiding is deemed as a good software engineering principle [Booc87].

2. Implement the specification using a single package incorporating all three system states. This would allow the types to remain private. However, this would create a large package, and in specifications that use schema promotion to a large extent this solution would be very impractical. Another disadvantage is that any additions to the system would require the package and all the client packages to be recompiled, even if the client packages do not use the new additions.

3. Ada95 allows the construction of a hierarchical library system that utilises child subprograms and child packages. When translating this specification the phone database can be implemented as a single package, and the staff database can be implemented by making the departmental database a child package of the phone database. A child package can be thought of as being declared within the declarative part of its parent, but after the specification part [Barn95]. This allows the private part of the parent to be visible to the child only. Child packages solve the problem of sharing private types amongst many packages and enable the package to be extended by 'bolting' on child units without recompilation of all client packages.

This project makes extensive use of reusable components. Reuse can also be employed in the construction of specifications that incorporate other specifications as seen in the staff database. Clearly, out of the three approaches outlined above,

child packages are the best means of ensuring that the visibility of private types is restricted to only those Ada packages that require it and that those packages can remain in individual packages.

5.5.4 Implementing the Phone Database

The state model can be constructed by the following instantiations of the reusable software components, where the types PERSON, PHONE and MaxNoLines are generic parameters to the package.

```
With Set_bounded_G,Many_to_many_G;

generic

   -------

Package Phone is

      ---- available operations

private

   package members_set_pack is new Set_bounded_G (person,msize,--);

   package phone_set_pack is new Set_bounded_G(Phone,MaxNoLines,--);

   package telephone_map is new Many_to_many_G (msize,MaxNoLines,
                                     members_set_pack,phone_set_pack);

   --The state model becomes :-

   telephones       : telephone_map.map;
   members          : members_set_pack.set;

end;
```

5.5.5 Implementing the Departmental Database

Staff_Db is implemented by making Dept_Db a child package of Phone_Db as discussed. Important aspects are shown in bold type as follows:-

```
with Set_bounded_G,Many_to_many_G,Many_to_one_G;

generic
   ---------

package phone.dep is          -- phone is the parent package

      --Available operations

private

   package department_set_pack is new Set_bounded_G(
                                department,size,---);

   package room_number_set_pack is new Set_bounded_G(
                                room_number,size,---);



   package located_map is new
                     Many_to_one_G(size,members_set_pack,
                                 department_set_pack);


 package occupies_map is new Many_to_one_G(size,members_set_pack,
                                            room_number_set_pack);

--The state model becomes :-

  occupies     : occupies_map.map;
  departments  : department_set_pack.set;
  located      : located_map.map;

end;
```

The package `members_set_pack` is visible here, even though it is in the private part of phone.ads.

5.6 Schema Types and Schema Bindings

Another way in which Z reuses specifications is by its use of schemas as types. A schema can be used as a type by creating a binding which allows the schema to be used as an object. The binding notation θ exists to create bindings and to abbreviate the making of associations. It can be used to identify the current component values of the schema and to equate the before and after state components of an operation.

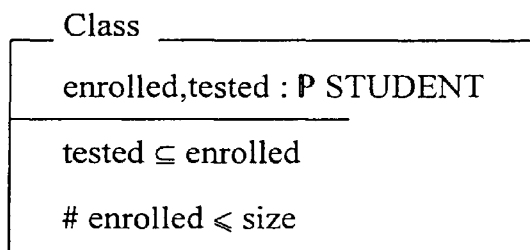
The next section looks at a specification for a class managers assistant [Word92] that uses schema types and the binding notation.

5.6.1 The Class Managers Assistant

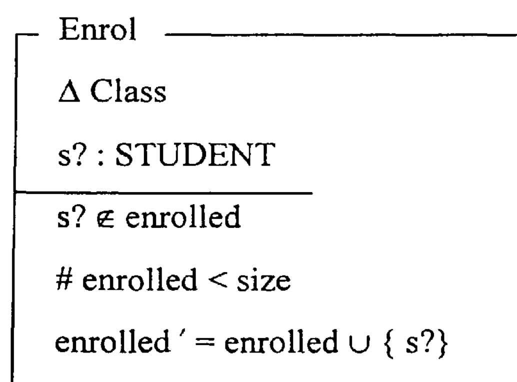
The specification that follows is taken from the Class Managers Assistant as specified in [Word92]. It has been used to show how schema types and bindings are implemented.

[STUDENT]

| size : $\mathbb{N}$



An operation can be defined to enrol a student into a class :-



The state schema Class could have been refined into an abstract state machine, but it is used as a type in the schema TwoClasses (following) and must therefore be implemented as an abstract data type. It is an abstract data type because the type

`Class` is exported from the package implementing the state schema Class (this package is given on page 79 and is named `a_class` due to different naming rules between Z and Ada). Exporting `Class` as a type enables many instances of `Class` to be used.

```
┌─ TwoClasses ──────────────┐
│ z_for_beginners : Class
│ z_advanced : Class
└───────────────────────────┘
```

Every operation on z_for_beginners contains the following predicates, so in [Word92], a schema is made of them, as follows:-

```
┌─ ZfbOperation ──────────────┐
│ ΔTwoClasses
│ Δ Class
├──────────────────
│ z_advanced′ = advanced
│ advanced doesn't change
│ z_for_beginners = θ Class
│ z_for_beginners′ = θ Class′
│ the changes in z_for_beginners are
│ identified with changes in a typical
│ class, component by component
└─────────────────────────────┘
```

A similar operation would exist to link changes in the class 'Two'.

An operation can be defined which reuses enrol and operates on the state TwoClasses. An operation to enrol a student into the class z_for_beginners would be:-

$$ZfbEnrol \mathrel{\widehat=} \exists \Delta Class \bullet Enrol \wedge ZfbOperation$$

The specification has the parachuted type student. Student will therefore be a generic type. The package implementing the state schema Class has been called `a_class` since the type Class is exported as a private type.

```
generic

  size:natural;
  type student is private;
  with function gt_student  (s1,s2: student) return boolean;
  with procedure out_student(s    :student);


package a_class is

-- The type class must be exported for use in TwoClasses, so it is
-- declared as a private type. Also, since this is an abstract data
-- type, each operation must act on the type class.


  type class is private;

  procedure Init     (c: out class) ;
  procedure enrol    (s:in student;c:in out class );
  procedure out_class(c:in class);

   enrol_error:exception;

private

     package set_pack is new Set_bounded_G
                         (student,size,-----);

-- The type class can now be defined as a record containing fields
-- for the set of enrolled and tested students.

    type class is record
       enrolled,tested : set_pack.set;
    end record;

end a_class;
```

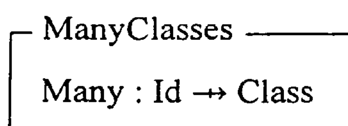
The operation enrol can now be implemented in the body of `a_class`. The state schema for TwoClasses is implemented by the following Ada specification (the generic parameters are the same as for `a_class`):-

```
with Set_bounded_G,a_class;

generic
       -------

package TwoClasses is

  type z_for_beginners is private;
  type z_advanced is private;

  procedure init_beginners(b: in out z_for_beginners);
  procedure init_advanced (a: in out z_advanced);

  procedure ZfbEnrol(s:student;b:in out z_for_beginners);
  procedure Za_Enrol (s:student;a:in out z_advanced);

   classtwo_enrol_error : exception;

private

    package class_pack is new a_class
                         (size,student,---------);

    type z_for_beginners is new class_pack.class;
    type z_advanced      is new class_pack.class;

end TwoClasses;
```

5.6.2 Extending the Class Managers Assistant

In the next example, the state upon which the operations act will not be given explicitly, but must be calculated within the Z specification. The simple class database can be extended to operate for many classes.

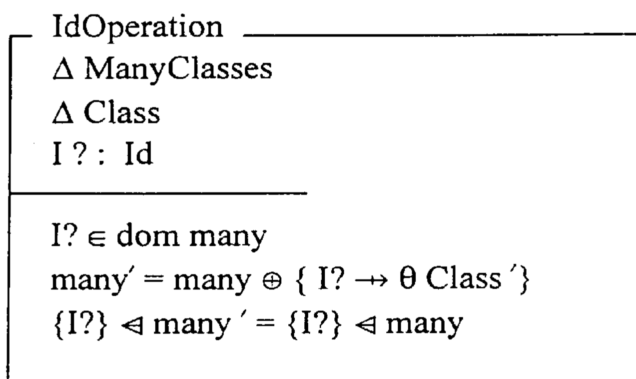
[Id] Each class will have a unique identifier.



The generic parameters to the package will be the same as for the TwoClass system, but extra parameters are required to create the set containing the unique identifiers. In order to create the state space for ManyClasses, the package `a_class` must be instantiated. A set package can then be instantiated using the type `class` from the instance of the package `a_class` to create a set of classes. This can, in turn, be used in the many to one function component along with the set of unique identifiers.

```
private

    package class_pack is new a_class
                            (size,student,-----);

    type the_class is new class_pack.class;

    package id_set_pack is new Set_bounded_G(id,size,-----);


    package set_of_classes_pack is new Set_bounded_G(
                                          the_class,size, -----);

    package many_map is new Many_to_one_G(size,
                            id_set_pack,set_of_classes_pack,-----);
    many:many_map.map;
```

To enrol a class the following linking schema is required :-



The refinement of this schema is dealt with in chapter 6 section 6.9.2.

Chapter 6

Operation Schema Refinement

6.1 Introduction

An operation schema from the Z specification will be refined into an exportable function or procedure in the Ada package body. The Z operations within the schema become function or procedure calls in the body of the Ada subprogram that implements the schema. In the case of the majority of operations in Z, the operators can be replaced by their equivalents from the reusable components. For example `is_a_member` replaces ∈ and `size_of` replaces #. In other cases a quantifier may be used which requires the use of a generic operation (see section 6.6.2). This chapter will describe the techniques that are required to turn the Z operation schemas into code and some examples will be given. A list of Z operations and their Ada counterparts are given in appendix 2.

6.2 Function or Procedure Bodies

A decision must be made as to whether functions or procedures are used to refine the schema. This decision depends on the type of the schema. Schemas that modify the state are implemented as procedures, whilst query operations that interrogate the system to return a value are implemented as functions.

6.3 Refining the Initial State Schema

The initial state schema specifies the state at which a system starts and can be used as an operation to set the system to the start values. There are a number of methods for initialising a system; this section will discuss each method in turn.

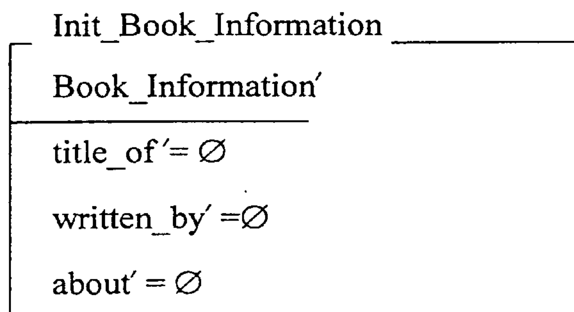
6.3.1 Initialising the Exported Type in the Ada Specification

The data structures used to implement the abstract data types modelling the types found in Z can be set to an initial value in the private part of the Ada specification. When a variable is declared as a type it will start its life at the initialised value. However, this method alone would not allow the variable to be reinitialised because the internals of the package where the initialisation took place will not be visible at

the level at which reinitialisation takes place. In order to refine an initial state schema this method alone is insufficient.

6.3.2 Exporting an Initialisation Procedure

For the reason mentioned above, each software component has a procedure which will create an item of a type by initialising the structures for the type. This allows the initial state schema to be refined by calling the appropriate create procedure. For example, the initial state schema for a library subsystem might be :-



This is refined by the following procedure :-

```
procedure Init_Book_Information is
begin
   create_set(title_of);
   create_set(written_by);
   create_set(about);
end;
```

Ada will automatically use the correct instance of the procedure `create_set` for the type given as a parameter. The operation `Init_Book_Information` is implemented as an operation in the abstract state machine package for `Book_Information`. Therefore, it is not necessary to include `Book_Information` explicitly in the parameter list of the Ada operation `Init_Book_Information`.

6.3.3 Initialisation in Body of Package Implementing Specification

In the case of most packages, a collection of operations are available for export, but no main program for the package is given. It is possible to include the initialisation of the package in the main body of the program. When the program is executed the system will take up this initial state. However, this method once again fails when it is necessary to reinitialise or clear state variables. In most cases the initial state will require the sets, functions and relations etc. modelling the state to be empty on start up. This method can be used where the state must be set up in a more detailed way. The specific state can be set up in the main part of the package body implementing the specification.

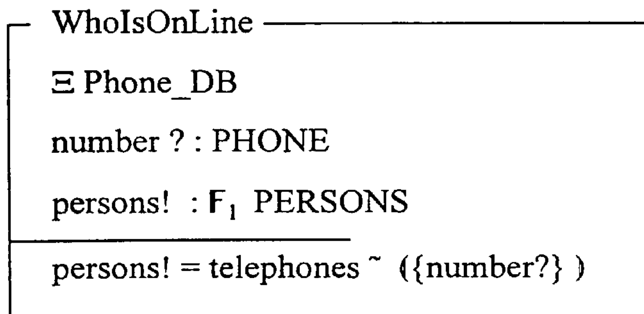
6.4 The Declarative Part of Schemas

The declarative part of schemas can consist of type declarations and a list of input and output variables. The refinement of the declarative part of schemas is now discussed.

6.4.1 Input and Output Variables

The schema will often have input and output variables in its declaration part. Since the package implementing the specification is turned into an abstract state machine, the state variables do not have to be listed in the Ada operations parameter list. Variables that use the '?' notation become input parameters to the procedure and can use the `in` mode in Ada. The use of the '!' notation in a schema requires a different technique as some important issues are raised. It is not sufficient to include an `out` mode parameter in the procedure or use a return statement if a function is used.

For example, consider the following schema taken from the Phone Database specification mentioned earlier (chapter 5.5.1).



This could be implemented as :-

```
function WhoIsOnLine (number:in Phone) return Person_set.set
```

or

```
procedure WhoIsOnLine(number:in Phone; Person : out
                                        Person_set.set)
```

In both cases, the package would have to export the type `Person_set.set` as well as the state that represents the specification. The package that uses the abstract state machine obtained from the Z specification would have to have visibility of the underlying representation of `set` in order to access each member of the set. In order to avoid this (since the underlying representation of set should be private for information hiding purposes), an operation to output types must be included in each of the reusable software components that model the main types. For example, in the set package, the following procedure is available.

```
Generic
  with procedure output_item(I:in Item);
procedure output_set( s: in set);
```

The `output_set` procedure is generic and can be instantiated over the procedure `output_item`. The procedure `output_set` is contained in the reusable set components and simply visits each item in the set and calls `output_item` for each item in the set. The code for the procedure `output_item` must be supplied by the developer. This code will appear in the package that instantiates and uses the abstract state machine derived from the Z specification. The code for outputting each

item must be written according to the non functional requirements that are not present in the Z specification. Output is specified in Z with the ! notation and this does not contain the actual details for the output. Using a generic procedure for output allows this information to be deferred to the actual instantiation of the abstract state machine. In this manner the abstract state machine implements the Z specification without saying how the output is to be achieved. The developer can decide whether the output is to be applied to a screen or a printer etc. or even both, because many `output_set` procedures can be instantiated by the developer by providing different code for `output_item`.

The operation WhoIsOnLine specified above can be declared in the Ada package specification as :-

```
-------
  generic
     with procedure out_person (P : in person)
   procedure WhoIsOnLine   (number : in Phone)
-------
```

In the Ada body of WhoIsOnLine an instantiation of the generic procedure from the set package must take place :-

```
i.e. procedure out_person_set is new members_set_pack.
                                     out_set(out_person);
```

The predicate in WhoIsOnLine can now call `out_person_set` to output the required set `persons!` (an example is shown in section 6.7.2). The code for `out_person`, provided in the software using the translated Ada package, could for example be simply :-

```
procedure out_person(P: in person) is
begin
   text_io.put(p);
end;
```

6.4.2 Declarative Types

On occasion, a type that is not a state variable will be declared in a schema. When types are declared that are state variables then they can be declared as being of the corresponding type. If a type does not exist as a state variable, then it must be created locally to the function or procedure implementing the schema. For example, consider the following schema fragment taken from the filling station specification.

```
┌─ TotalNumberAtEachPump ──┐
│ Ξ FillingStation
│ numbers ! : PUMP ⇸ ℕ
├──────────────────────────
│ ------
```

The type `numbers` is not a type derived in the state schema. The type will have to be created by instantiating a function package locally to the operation `TotalNumberAtEachPump`. However, since the type is only required locally it may not be necessary to create the type in the first instance. In the case above, it would be possible to simply access and output the pump and the relevant number of cars without creating the function.

6.5 Refinement of Z Statements

The next section will deal with refinement issues concerning predicates in operation schemas. Some of the issues discussed will cover the refinement of simple Z statements, preconditions, quantifiers as preconditions, complex Z postconditions and comprehension terms. Some of the operations in Z (e.g. quantifiers, set comprehension terms and mu expression) must be translated using generic units available in the software components.

6.5.1 Simple Z Statements

Most of the statements written in Z can be refined with a direct replacement of the relevant operation in the reusable software component. Many statements in Z such as

I? ∈ dom (M) are used commonly and have been implemented with single statements such as `is_a_domain_element(D,M)` for reasons of efficiency. An alternative would be to implement it as :-

```
whole_domain(M,domain_set);      -- get the domain of the map
is_a_member(I,domain_set);       -- test for set membership of I in
                                 -- temporary set (domain_set)
```

A full list of general Z statements, the equivalent Ada operations to translate them and some simple guidelines are given for each type of software component (see appendix 2). The general statements are also useful for refining complex statements as discussed in section 6.8.

Many operations such as restrictions and subtractions on the domain and range contain a number of overloaded procedures. This is because restrictions and subtractions can be defined as acting on a single item or on a set of items, and it is possible to change the existing state or create a new object. Care must be taken when using procedures that belong to different types. For example in a one to one function $f : X \rightarrowtail\!\!\!\!\rightarrow Y$, the Z statement $f^{\sim}(y)$ will return the domain for a given valid range input provided it is there. In the case of a many to one function $f: X \twoheadrightarrow Y$, $f^{\sim}(y)$ will return a <u>*set*</u> of domain elements for any given valid range element.

6.6 Refinement of Preconditions

The propositions in the precondition of a Z schema are refined by a nested '`if`' statement in Ada. Each proposition has an '`if`' clause and the program flow is arranged so that the postcondition appears in that part of the nested '`if`' structure for which all the '`if`' statements are satisfied. Each proposition in the precondition is evaluated and if satisfied, program control will move on to the next proposition, or if all propositions in the precondition are satisfied, program control moves onto the

postcondition. If any of the propositions are not satisfied then the program flow does not arrive at the postcondition. The following schema serves as an example:-

```
AddEntry_NewLine
Δ Phone_Db
name ? : PERSON
newNumber ? : PHONE
----------------------
# (ran telephones ) <  maxNoLines
name ? ∈ members
newNumber ∉ ran telephones
They must be a staff member, the line must be new
------
telephones′ = telephones ∪ {name? ↦ newNumber? }
members′ = members
```

The three propositions in the precondition for AddEntry_NewLine are translated as:-

```
------
if  size_of(range_of(telephones)) < maxNoLines
then
   if is_a_member(name,members)
   then
      if not is_a_range_element(newNumber,telephones)
      then
         -- DO POSTCONDITION
      end if;
   end if;
end if;
```

6.6.1 Robust Schemas

Many specifications use a style where the pre and postconditions of an operation are given without any details of error conditions. The error conditions are contained within a separate schema that can be combined with the original by schema disjunction to create the fully robust schema.

The schema for AddEntry_NewLine has three propositions, so there are three possible 'error' conditions. Schemas can be defined to handle what happens when

the propositions are not satisfied. Error schemas for the each of the three propositions are:-

```
┌── Addentry_Error1 ──────────────┐
│ Ξ Phone_Db
│ reply! : message
├─────────────────
│ ¬ (# (ran telephones ) < maxNoLines)
│ reply! = NotEnoughLines
└─────────────────────────────────┘
```

```
┌── Addentry_Error2 ──────────────┐
│ Ξ Phone_Db
│ reply! : message
├─────────────────
│ ¬ (name ? ∈ members)
│ reply! = NotAMember
└─────────────────────────────────┘
```

```
┌── Addentry_Error3 ──────────────┐
│ Ξ Phone_Db
│ reply! : message
├─────────────────
│ ¬ (newNumber ∉ ran telephones)
│ reply! = NumberAlreadyExists
└─────────────────────────────────┘
```

The robust schema operation for AddEntry_NewLine would be defined as :-

$$\text{RobustAddEntry_NewLine} = \text{AddEntry_NewLine} \lor \text{Addentry_Error1} \lor \text{Addentry_Error2} \lor \text{Addentry_Error3}$$

RobustAddEntry_NewLine would be translated as :-

```
-----
if  size_of(range_of(telephones)) < maxNoLines
then
   if is_a_member(name,members)
   then
      if not is_a_range_element(newNumber,telephones)
      then
         -- DO POSTCONDITION
      else
         put_line("NumberAlreadyExists");
      end if;
   else
      put_line("NotAMember");
   end if;
else
  put_line("NotEnoughLines");
end if;
```

In this example `put_line` statements have been used to print messages to the screen. This method of screen output is seen in other examples of animation found in the literature [Woodc96]. The problem is that the specification does not include details as to how the output is to be achieved. It may not be desirable to output to a screen. Indeed there may be no screen in the system.

In the next example of implementing RobustAddEntry_NewLine a decision was made to use exceptions to flag these 'error' conditions. This has the advantage that error detection and recovery can be localised through the use of the existing Ada exception handling mechanism [Booc87 pg 580]. The software using the Ada package of the translated Z specification can catch these exceptions and take appropriate actions according to requirements not specified in the Z specification. These actions could be, for example, outputting a message to the screen, attempting the same operation with different data, or even doing nothing when catching the exception. The code for RobustAddEntry_NewLine incorporating the exceptions is now:-

```
-------
if size_of(range_of(telephones)) < maxNoLines
then
   if is_a_member(name,members)
   then
      if not is_a_range_element(newNumber,telephones)
      then
         -- DO POSTCONDITION
      else
         raise NumberAlreadyExists;
      end if;
   else
      raise NotAMember;
   end if;
else
   raise NotEnoughLines;
end if;
```

In the error schemas, it can be noted that when each proposition is satisfied an output message occurs. These propositions are the negation of the original propositions in AddEntry_NewLine. It is possible to refine the robust operation in a similar manner by negating the propositions of AddEntry_NewLine and raising exceptions if the negated propositions are true. The operations implementing the propositions in the precondition return boolean values and so negating them does not alter the meaning of the code so long as the correct program control is used to arrive at the postcondition. In the example below, program control only arrives at the postcondition if all the negated propositions of AddEntry_NewLine are false. The equivalent to arriving at the postcondition when the precondition is true as in the first example of AddEntry_NewLine. Implementing the code in the manner shown next, enables the depth of indentation of `if` statements to be reduced and preserves the meaning of the specification.

```
-------
if not size_of(range_of(telephones)) < maxNoLines then
   raise NotEnoughLines;
elsif not is_a_member(name,members)then
   raise NotAMember;
elsif is_a_range_element(newNumber,telephones)then
   raise NumberAlreadyExists;
else
   -- DO POSTCONDITION
end if;
```

6.6.2 Quantifiers in Preconditions

Each type of software component has generic functions to handle both the existential and universal quantifiers. It is possible to use them over sets, functions, relations, or sequences. The existential quantifier traverses each item in the structure and is used to test if an item exists in the structure, according to specified rules. If an item obeying the specified rules is found then the traversal of the structure stops. The universal quantifier traverses the structure and tests that all the items in the structure of interest follow given rules. Their general forms are :-

$\exists$ **D | P•Q** and $\forall$ **D | P•Q** where:

$\exists$ is the existential quantifier

$\forall$ is the universal quantifier

D is some declaration

P is a predicate acting as a constraint

Q is a predicate being quantified

For example the statement $\exists\ \mathbf{n{:}S \mid n < 10 \bullet n^2 = 64}$ can be read as there exists a number in the set S where the number is less than 10 and its square is sixty four. The following operations are used to define the quantifiers for a set package. Functions would be declared for P and Q such as :-

```
function Lt_Ten (n : in S) return boolean is
begin
  return n < 10;
end;

function Nsq_equal_64 (n: in S) return boolean is
begin
  return n*n = 64;
end;
```

The Ada specifications for the quantifiers is now given. The `ident` functions are used in the cases where P from the above definition is missing (e.g. $\exists\ \mathbf{n{:}S \mid n^2 = 64}$).

```
-------
  function  IDENT(I : in ITEM_TYPE) return ITEM_TYPE;
  function  IDENT(I : in ITEM_TYPE) return boolean;
```

```
-------

generic
    with function P(I:in ITEM_TYPE) return boolean;
    with function Q (I:in ITEM_TYPE) return boolean is IDENT;
function THERE_EXISTS(S : SET) return boolean;

generic
   with function P(I:in ITEM_TYPE) return boolean;
   with function Q (I:in ITEM_TYPE) return boolean is IDENT;
function FOR_ALL(S:SET) return boolean;
```

6.6.3 Refinement of Universal Quantifiers

As an example of the use of the universal quantifier, the following schema is presented. It is also taken from the petrol station specification, and it describes what happens when a car arrives at the station and joins the queue at a pump.

```
Arrives
Δ FillingStation
c? : Car
p? : Pump
--------------------
p? ∈ dom queues
∀ p : dom queues • c? ∉ ran (queues p )
c? ∉ serviced
queues′ = queues ⊕ { p? ↦ queues p? ⁀ <c?>}
serviced′ = serviced
```

The statement '$\forall$ p : dom queues • c? $\notin$ ran (queues p)' says that the car arriving at the pump must not already be in a queue at any pump. A universal quantifier is translated following the form **Quantifier : Type • Predicate** and the following three point process can be used :-

I. The quantifier is the universal quantifier which requires an instantiation of a generic function.

II. The type of the set is dom(queues). Therefore ∀ must be instantiated from `pump_set_pack`. A set over which ∀ iterates must be enumerated. This is achieved with `whole_domain(pump_set,queues)`, where `pump_set` is defined locally to `Arrives` as type `pump_set_pack.set`.

III. The predicate $c? \notin \text{ran}(\text{queues } p)$ follows the general statement $I \notin \text{ran } S_q$
This is implemented by **not is_a_member(c,S_q)**
where S_q = queues p which follows the form M(D), which is refined by
range_of(p, S_q , queues)

The refinement process has stopped, as all the operations in the general statements have been fully decomposed, and all the information necessary for implementation has been collected. However the parameter S_q is still unknown and so becomes a local variable of type `car_sequence`. Reading upwards to collect the refined statements allows the full implementation for the predicate part of the universal quantifier construct. In the code below, the variable `c` is in scope as it is an input parameter of the procedure implementing the schema Arrives.

```
Procedure Arrives(c: in car; P in Pump) is


  function is_car_in_sequence (P:pump) return Boolean is
     Sq : car_sequence;
  begin
     range_of (p,Sq,queues);
     return not is_a_member(c,Sq);
  end;

  function Car_not_in_queue is new pump_set_pack.
                                   for_all(is_car_in_sequence);

begin
   if Car_not_in_queue then

     ---  other statements in implementation of predicate

end;
```

6.6.4 Refinement of Mu-Expressions

The Mu-expression is defined as follows :-

$\mu\, D \mid P \cdot E$ Where D- Denotes declarations.

P- Denotes a predicate constraining the values.

E- An expression denoting a term.

The value that makes the property of P true is used in the expression E to give the result of the Mu-expression. The Mu-expression is defined only if there exists a unique element to make the property of P true. If no expression is supplied then the value of the characteristic tuple is used. A Mu-expression is similar to a set comprehension procedure that produces a singleton set. Unfortunately the set comprehension procedure cannot be used to translate a Mu-expression because it returns a set instead of an item. For this reason a separate generic function is used in the refinement of Mu-expressions :-

```
generic
   with function Predicate (I: Item_type)return boolean;
   with function Expression(I: Item_type)return Item_type is Ident;
function Mu_expression(S: Set) return Item_type;
```

Consider the following statement that uses the state model of the filling station.

$\mu\, P : PUMP \cdot S = \text{queues } P \wedge c? \in \text{ran } S$

This statement is translated as follows :-

```
function car_in_Sequence(P:pump) return Boolean is
begin
  range_of (P,s,queues);
  return is_a_member(c,s);
end;

  function car_at_pump is new
                    pump_set_pack.Mu_expression(car_in_Sequence);
```

The function `car_at_pump`, which refines the mu-expression is used in the following manner.

```
procedure ---- is

    p     : pump;
    pumps: pump_set_pack.set;

      ----- function for car_in_sequence
      ----- instantiation of car_at_pump

begin
   whole_domain(pumps,queues);
   P:= car_at_pump(pumps);        -- this delivers the value of P
   ----
```

In this example `P` is used locally to the procedure. However, care may need to be taken to make sure that the delivered item from the mu-expression is in scope with regards to its declaration and use in Ada.

6.6.5 Elimination of Propositions in Precondition

In some cases, some propositions in a precondition can be left out of the Ada code implementing the operation schema. This can be done if the proposition is effectively contained within an Ada operation that raises an exception and is used in the translation of a postcondition. For example, the Ada set operation `insert` models the statement $S' = S \cup \{I\}$. However, it also ensures that the item is not already in the set so in effect `insert` models the situation $I \notin S \wedge S' = S \cup \{I\}$. Consider the following schema from the filling station specification as specified in [Norc91] .

Serviced
ΔFillingStation
p? : Pump

$p? \in \text{dom queues}$
the pump is on the forecourt
$\text{queues } p? \neq \langle\rangle$
the queue at the pump is not empty

$\text{serviced}' = \text{serviced} \cup \{\text{head (queues } p?)\}$
the car at the front of the queue has been serviced
$\text{queues}' = \text{queues} \oplus \{ p? \mapsto \text{tail (queues } p?) \}$
the car has left the queue at p?

In this example, the propositions in the precondition could be implemented in the following way:-

1.) $p? \in \text{dom queues}$ follows the general statement $I \in \text{dom } M$

This is implemented by **Is_a_domain_element(p,queues)**

2.) $\text{queues } p? \neq \langle\rangle$ follows the general statement is_empty(S_q)

This is implemented by **is_empty(sq)**

where S_q = queues p

The general form for this statement is M(d) which is refined by :-

range_of (p,sq,queues)

This would give the following code :-

```
if is_a_domain_element(p,queues)
then
   range_of(p,sq,queues);
   if  not is_empty (sq)
   then

       -- DO POSTCONDITION;

----
```

However, in this case, the precondition is unnecessary for implementation purposes because the two propositions making up the precondition are themselves contained in Ada operations which raise exceptions and are used in the translation of the postcondition :-

$$serviced' = serviced \cup \{head\ (queues\ p?)\}$$
$$queues' = queues \oplus \{p? \mapsto tail\ (queues\ p?)\}$$

The Ada operations `head`, `tail` and `range_of` (which implement *queues p?*) each contain code to raise an exception. In the Ada implementation of `head` and `tail` an exception is raised if the sequence is empty. In the Ada operation `range_of` an exception is raised if an attempt is made to find the corresponding range of an item that does not appear in the domain of the function. These exceptions are effectively equivalent to the propositions $p? \in dom\ queues$ and $queues\ p? \neq \langle\rangle$ in the precondition. The use of exceptions within the reusable software components was discussed in chapter 3.2.1. An alternative version of serviced is possible :-

```
procedure serviced(p:pump) is
    temp_seq,tail_seq:car_sequence;
    c:car;
begin
   range_of(p,temp_seq,queues);--raises exception if p ∉ dom queues
   head(temp_seq,c);           --raises exception if temp_seq = <>
   tail(temp_seq,tail_seq);
   insert(c,serviced);
   function_override(p,tail_seq,queues);

   exception
    when queues_map.domain_not_found     => null;
    when car_seq_pack.sequence_is_empty => null;

end;
```

In the exception handler above, nothing is done when the exceptions are caught. However, if error schemas were specified as in section 6.6.1, the following exception handler could be used to catch the exceptions and export ones relevant to the error schemas :-

```
when queues_map.domain_not_found     => raise not_a_valid_pump;
when car_seq_pack.sequence_is_empty  => raise queue_is_empty;
```

When the program carries out the operations in the refinement of a postcondition an exception is raised if either the domain item does not exist or the sequence is empty. In this example, the only way that a postcondition can alter the state is if no exceptions are raised, or, in the original context, if both propositions in the precondition are true. Care must be taken in order to use this technique. The developer must ensure that all the propositions are effectively contained in Ada operations implementing the postcondition, which raise exceptions. Also, the developer must ensure that no state variables are allowed to change, whilst implementing the postcondition, before the opportunity of raising the relevant exceptions is exhausted.

6.7 Comprehension Terms

Set comprehension offers a powerful means of defining a set by stating properties that distinguish its members from values of the same type. Since functions, relations and sequences can have sets as constituent parts, it is possible to specify and implement comprehension terms over these structures.

In Z, a set comprehension term is written as $\{ \mathbf{D} \mid \mathbf{P} \cdot \mathbf{E} \}$ where D denotes declarations, P is a predicate constraining the values and E is an expression denoting a term. Each software component contains a generic comprehension procedure, and when used will require appropriate functions in order to complete the instantiation process. This allows any number of different set comprehension terms to be defined. The example given below belongs to the set package and defines the standard set comprehension operation. The package also caters for situations where the expression is left out by using an identity function.

```
generic
    with function PREDICATE (I1 : in ITEM_TYPE) return boolean;
    with function EXPRESSION(I1 : in ITEM_TYPE) return
                                        ITEM_TYPE is IDENT;
 procedure SET_COMPREHENSION (S1 : in SET;S2 : in out SET);
```

It is also possible to create a set comprehension procedure in a utility package to allow a set of different types to be created. The set comprehension procedure must be additionally parameterised over the type of the new set. The expression function will be provided by the developer to create the new type from the existing one, ready for insertion into the set of new types (`NEW_SET.SET`, shown below).

```
generic
      type NEW_SET is new Set_bounded_G (<>);
      with function PREDICATE (I : in ITEM_TYPE) return boolean;
      with function EXPRESSION(I : in ITEM_TYPE) return
                                       NEW_SET.ITEM_TYPE is IDENT;

procedure SET_COMPREHENSION(S1:in SET;S2 : out NEW_SET.SET);
```

6.7.1 Lambda Expressions (λ)

A Lambda expression takes the form $\lambda\ \mathbf{D} \mid \mathbf{P} \bullet \mathbf{T}$, where D denotes declarations, and P is a predicate constraining the values which are mapped to a value defined by T. The Lambda expression is merely syntactic sugar for the set comprehension statement given as :-

$$\{x : T \mid P \bullet x \mapsto \text{term}\}$$

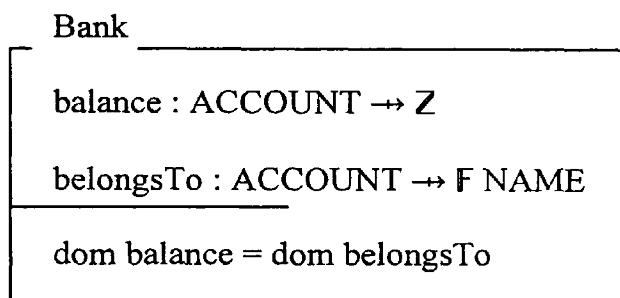
So, the lambda expression $\lambda\ n : \mathbb{N} \mid n < 8 \bullet n^2$ can be written as :-

$$\{ n : \mathbb{N} \mid n < 8 \bullet n \mapsto n^2 \}$$

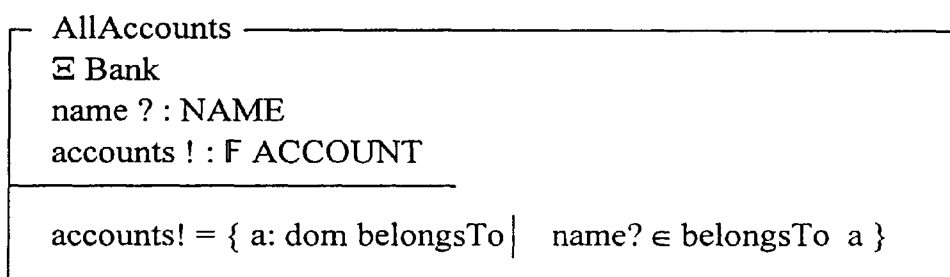
Lambda expressions are therefore translated as set comprehension statements. Some examples of translating set comprehension statements are given below :-

6.7.2 Implementing Set Comprehension

The following state schema models a banking system:-



An operation involving set comprehension could be to output the set of all accounts belonging to a given person :-



The refinement of a set comprehension term is very similar to that of a quantification term.

I. The Z statement defines a set comprehension term which requires an instantiation of a generic function.

II. The type is ACCOUNT. Therefore the set comprehension term must be instantiated from `account_set_pack`. A set over which the comprehension iterates must be enumerated. This is achieved with `whole_domain (account_set,belongsTo)`, where `account_set` is defined locally to the procedure implementing the schema as type `account_set_pack.set`.

III. The proposition 'name? ∈ belongsTo a' follows the general statement I ∈ S where S is obtained from the functional application 'BelongsTo a'.

The translation of these statements results in the following Ada operations :-

range_of (a, name_set, belongsTo)

is_a_member(name,name_set)

The Ada specification for the operation AllAccounts is given as :-

```
generic
    with procedure output_account (a: in account);
procedure AllAccounts (n: in name);
```

The full implementation for AllAccounts is now given, the instantiation for the set comprehension procedure has been highlighted in bold :-

```
procedure AllAccounts (n: in name) is
   account_set :account_set_pack.set;

   procedure output_set_of_accounts is new
                account_set_pack.out_set (output_account);

   function find_accounts_for_name(a:account)return boolean is
      name_set:name_set_pack.set;
   begin
      range_of(a,name_set,belongsTo);
      return (is_a_member(n,name_set));
   end;

   procedure enumerate_set_of_accounts is new
                  account_set_pack.set_comprehension
                                 (find_accounts_for_name);

begin
   whole_domain(account_set,belongsTo);
   enumerate_set_of_accounts(account_set,accounts);
   output_set_of_accounts(accounts);
end;
```

6.7.3 Example of Comprehension Involving Many to One Function

Set comprehension techniques can also be applied to other structures such as functions, relations and sequences. The following schema is taken from the departmental phone database specification.

```
_ Merge_departments ______________________________
Δ Staff_Db
Ξ Phone_Db
d1?,d2? : DEPARTMENT
dnew? : DEPARTMENT
__________________
d1? ∈ departments
d2 ? ∈departments
dnew? ∉ departments
-----
located ′ =
located ⊕ { p: PERSON | located p = d1? ∨ located p = d2? • p ↦ dnew? }
departments ′ = departments ∪ {dnew?} \ {d1?,d2?}
__________________________________________________
```

The statement $located' = located \oplus \{ p: PERSON / located\ p = d1? \vee located\ p = d2? \bullet p \mapsto dnew? \}$ can be implemented by using the generic `map_set_comprehension` procedure found in the many to one map package because the normal function override procedures that exist apply to single pairs or whole functions or relations.

The predicate part can be implemented following the same method as the previous example. It is given by the function :-

```
function is_persons_department_valid(p:person;
                                     dept:department)return boolean is

  temp_department:department;
begin
  range_of(p,temp_department,located);
  return (temp_department = d1 or temp_department = d2);
end;
```

If the predicate above is satisfied then the new department must replace the old one. The procedure below models the expression term in the set comprehension definition. Note that this example makes use of the expression function that is

available whilst in the previous example an expression function was not supplied as the `ident` function was available.

```
procedure join_person_to_new_dept(
                        p:in out person;dept:in out department) is
begin
  dept:=dnew;
end;
```

The package containing the `located` function will then be updated by the comprehension procedure, which will be equivalent to the function override operator. The body of `Merge_departments` can now be completed. Both the formal subprograms listed above must be presented locally to `Merge_departments` so that the `map_set_comprehension` procedure can be instantiated (it is highlighted in bold).

```
procedure Merge_departments(d1,d2,dnew:department) is

  new_located:located_map.map;

  function is_persons_department_valid ( ----
  begin
    -----
  end;

  procedure join_person_to_new_dept ( ----
  begin
    -----
  end;

  procedure new_locations is new located_map.map_set_comprehension
          (is_persons_department_valid,join_person_to_new_dept);

begin
  if is_a_member(d1,departments) then
    if is_a_member(d2,departments)then
       if not is_a_member(dnew,departments)then
          new_locations(located,new_located); --map_set_comp
          copy(new_located,located);          --creates a new map
                                              --so copy back

            -------- Rest of Implementation of postcondition

       end if;
    end if;
  end if;
end merge_departments;
```

6.7.4 Use of Iterator Procedure for Set Comprehension Terms

Some set comprehension terms cannot be implemented as they are stated. In order for the set comprehension procedure to work, the input set must be definite, finite and enumerated. The schema below, is an example of a set comprehension statement that uses a set that has not been enumerated and is therefore not suited to refinement with the set comprehension procedure.

```
 TotalNumber ____________________________________________
| Ξ FillingStation
| n! : ℕ
|________________________
| n! = # { c : CAR; P : dom queues | c ∈ ran (queues p ) • c }
| this uses set comprehension to compute the set of all cars at pumps
|__________________________________________________________
```

The problem here is that the set comprehension term is used to find the set of all cars at the station, which is drawn from the set of all cars which exist. The set of all cars that exists has not been enumerated and hence, it is not possible to draw cars from the set of cars that exist and then test them to see if they are at the service station.

In these cases the generic iterator procedure, found in each component, can also be used to the same effect. The iterator permits non-destructive traversal through a structure over which it is defined. A parameter 'CONTINUE' is included to terminate the iteration if desired. This type of iterator is known as a passive iterator and has been used here because it is safer than the equivalent active iterator [Booc87]. A discussion of iterators and their use in the construction of software components is described in chapter 8 section 8.3. The iterator procedure shown below is taken from the set package. It will visit each item in the set, and carry out operations defined in `process`. This will occur until either then end of the set is reached or when `continue` becomes false.

```
generic
   with procedure PROCESS(I1:in ITEM_TYPE;CONTINUE : out boolean);
procedure ITERATE(S:SET);
```

An alternative to using the iterator procedure would be to rewrite the Z set comprehension statement. One logical statement can be expressed as many different Z predicates, it is possible to rewrite the statement as $\mathbf{n! = \# (ran (\bigcup ran\ queues))}$ in order to achieve the same result. This statement is written in a functional form, but it is still specified in mathematical terms and is as unambiguous as the original statement. However, it is in a form that accesses the information and structures specified in the state schema and is therefore easily implementable. In [MacD89] it is stated that a specification may be changed to make proofs easier. Rewriting the Z statement can be justified on these grounds, in order to make the implementation easier.

The following difficult set comprehension example shows just how versatile the iterator procedure can be.

```
┌─ Departmental_Db ───────────────────────────────────────────
│ Ξ Staff_Db
│ deptartment ? : DEPARTMENT
│ departmental_Db ! : ℙ (PERSON × ROOM_NUMBER × 𝔽 PHONE )
├──────────────────────
│ department? ∈ departments
│ departmental_Db! = { P : PERSON; r : ROOM_NUMBER; t : 𝔽 PHONE |
│   located p = department? ∧ occupies p = r ∧ t = telephones ( {p} )}
└─────────────────────────────────────────────────────────────
```

The output from this schema is a set of tuples. The procedure Departmental_Db which refines this schema can be declared as a generic procedure in the Ada package specification. This allows a client to specify a device or format for the output of the types. The output for the types PERSON and ROOM_NUMBER have been combined in the same procedure to afford the developer more flexibility. The procedure

`out_phone` has been given separately because it is required in order to instantiate an output procedure for the set of phones.

```
generic
      with procedure out_phone(t:a_phone);
      with procedure out_person_room(p:person; r:room_number);
procedure Departmental_Db (dept:department);
```

The operation to create the set of tuples is not suited to the set comprehension procedure. In this case it is better to use the generic iterator procedure as this would avoid the requirement to create a set package of the type (PERSON × ROOM_NUMBER × F PHONE). The terms within the set comprehension statement all involve P. It is possible to iterate through the set of members and for each P in this set extract the relevant information from the database. The body contains the instantiated version of `out_set` to output a set of phones. It is instantiated by the formal subprogram `out_phone`.

```
procedure departmental_db(dept:department) is

   procedure out_phone_set is new phone_set_pack.out_set(out_phone);

     procedure iterate_process(p:in person;continue:out boolean) is
       r             :room_number;
       temp_dept     :department;
       telephone_set:phone_set_pack.set;
     begin
       range_of(p,temp_dept,located);
       if temp_dept = dept then
         range_of(p,r,occupies);
         phone.telephone_map.range_of(p,telephone_set,telephones);
         out_person_room(p,r);
         out_phone_set(telephone_set);
       end if;
       continue:=true;
     end;

     procedure iterate_through_members is new
                          members_set_pack.iterate(iterate_process);

begin
   if is_a_member(dept,departments) then
     iterate_through_members(members);
   end if;
end;
```

6.7.5 Use of Iterators for Counting Sets

A very common use of set comprehension occurs when the number of items in some set of interest must be found. The set comprehension term defines the set containing the items of interest, and the size_of operator (#) is then applied to this set. If the iterator procedure is used, then the need to produce the set first can be avoided. The following schema taken from the hotel specification [Hayw96] serves as an example:-

```
 __ Single_occupant ____________________________
| Ξ Hotel
| numberWithSingle ! : ℕ
|________________________
| numberWithSingle ! = # { r: occupied | # (occupants r ) = 1}
|_______________________________________________
```

where :-

```
| occupied : 𝔽 ROOM
| occupants : ROOM ⇸ 𝔽₁ NAME
| --------
```

This is a very straight forward example of a set comprehension term. However, only the number of items in the set is required so it is wasteful to create the set. A better solution would be to iterate through the set `occupied` and for each room in occupied apply the function `occupants` to get the set containing the names of the people in the room. If the size of this set is equal to one then the running total for the number of single occupant rooms can be incremented. For example :-

```
procedure counting_process (r:room_number; continue: out
                                                boolean) is
   name_set: set_of_names_pack.set;
begin
   continue := true;
   range_of (r, name_set,occupants);
   if size_of(name_set) = 1 then
     numberWithSingle := numberWithSingle + 1;
   end if;
end;
```

6.8 Refining Complex Postconditions and Statements

The refinement of long complex statements within the operation schemas is a difficult process and requires a different technique to aid the process. Originally, the author analysed the statements in applicative order (from the inside out). This made the process difficult as the best place to start was not always obvious. A different tactic has now been employed in order to make the process more mechanical. The statement is written in a general form and refined in normal order (from the outside in) to create the Ada code segments. The code segments are then reversed to establish applicative order for the computer. When the general form is refined with the appropriate Ada operation(s) any parameters in the general statement which cannot be implemented with information already known, will require further refinement. These parameters, will consist of other smaller Z statements which are also written as general statements. The method proceeds until the information for all the parameters in the general statements, and their Ada equivalents are known. The Ada equivalents are then rewritten in reverse order, with the information in their parameter lists being substituted upwards. When the refinement for each general statement ends, any parameters which are unknown after back substitution will be defined locally to the main procedure or function implementing the Z schema. This procedure basically splits the complex operation into a number of operations, available in the reusable components, and enables them to be applied in the correct order.

The following statement serves as an example and is taken from the Switch schema specified in [Norc91], where c? is of type Car and p? is of type pump.

$$\textbf{queues}' = \textbf{queues} \oplus \{p? \mapsto \textbf{queues}\ p? ^\frown \langle c? \rangle, q \mapsto \textbf{squash}(\textbf{queues}\ q \rhd \{c?\})\}$$

As a reminder, the state variables and given types are as follows :-

[Pump, Car] The parachuted types are pump and car.

----

queues : Pump ⤔ seq Car

waitingToLeave : **P** Car

It can be seen that the statement is basically a functional override application where the parameters to be overridden are calculated from other statements.

The statement follows the general form $M' = M \oplus \{d1 \mapsto r1, d2 \mapsto r2\}$

This is implemented by

function_override (d1,r1,M)
function_override (d2,r2,M)

where M = queues
$d1 = p$
$d2 = q$
$r1$ = queues p? ⁀ <c?>
$r2$ = squash (queues q ⩥ {c?})

<u>calculating r1</u> :-

This follows the form S_q ⁀ < I >

and is implemented as **construct (c, S_q)**

where S_q = queues P? and is implemented as **range_of (p, Sq ,queues)**

<u>calculating r2</u> :-

r2 follows the form $S_q \upharpoonright$ {I}

which is implemented by **sequence_range_subtract (c,sq1)**

where sq1 = queues q and is implemented by **range_of (q, sq1,queues)**

Reading upwards and substituting for implemented statements in bold type results in the following full implementation for the statement in applicative order.

```
range_of (q,sq1, queues)
sequence_range_subtract (c,sq1)
range_of (p, sq,queues)
construct (c, sq);
function_override (p,sq1,queues)
function_override (q,sq,queues)
```

6.9 Operations Involving Schema Types and Bindings

Chapter three discussed state schema refinement and introduced the method of refining state schemas that involved schema types and the binding notation (θ). This section implements the operation schemas that were introduced in chapter 5 section 5.6.1. The important schemas are reintroduced, but the reader is redirected to the Ada package specifications for the implemented state schemas.

6.9.1 Class Managers Assistant Operation Schemas

As a reminder, the state for the class system is :-

```
┌─ Class ──────────────────────────
│ enrolled,tested : ℙ STUDENT
├──────────────
│ tested ⊆ enrolled
│ # enrolled ≤ size
└──────────────────────────────────
```

The operation to enrol a student is given as :-

```
┌─ Enrol ──────────────────────────
│ Δ Class
│ s? : STUDENT
├──────────────
│ s? ∉ enrolled
│ # enrolled < size
│ enrolled′ = enrolled ∪ { s?}
└──────────────────────────────────
```

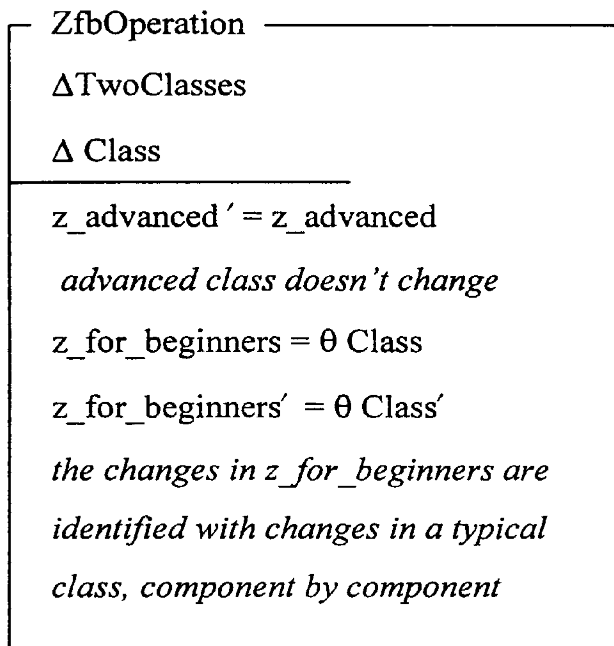
The operation enrol can be implemented in the Ada package body of `a_class`. The `init` procedure would refine the initial state schema to create the sets enrolled and tested.

```
package body a_class is

  procedure init(c:out class)  is
  begin
    set_pack.create_set(c.enrolled);
    set_pack.create_set(c.tested);
  end;

  procedure enrol(s: in student;c: in out class) is
  begin
    if not set_pack.is_a_member(s,c.enrolled) and
                   (set_pack.size_of(c.enrolled) < size)
    then
       set_pack.insert(s,c.enrolled);
    else
       raise enrol_error;
    end if;
  end;

 --------------
```

The state for the two class system is given as :-

```
┌─ TwoClasses ─────────────┐
│ z_for_beginners : Class
│ z_advanced : Class
└──────────────────────────┘
```

An operation can be defined which reuses enrol and operates on the state TwoClasses.



A similar operation would exist to link changes in the class 'two'. An operation to enrol a student into class z_for_beginners would be:-

$$ZfbEnrol \widehat{=} \exists \Delta Class \cdot Enrol \wedge ZfbOperation$$

The operation ZfbEnrol that links enrol with TwoClasses can be implemented in the package body for the two class system :-

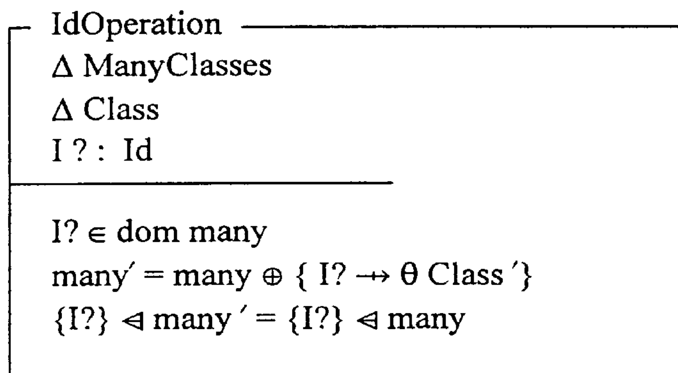
```
package body classTwo is

   ------

  procedure ZfbEnrol(s:in student;b:in out z_for_beginners) is
  begin
     enrol(s,b);
     exception when enrol_error => raise class_two_enrol_error;
  end;
```

The procedure `ZfbEnrol(s,b)` is from the package `a_class`. In this example, the class 'z_for_beginners' is explicitly identified by the theta notation as the current

state upon which the operation must act. For this reason, the type `z_for_beginners` is used as an `in out` parameter to the operation `ZfbEnrol`.

6.9.2 The Extended Class Managers Assistant Operations

To enrol a class the following linking schema is required :-



The operation to enrol a student into a class is expressed in the same way as the OneEnrol operation, namely :-

$$IdEnrol \cong \exists \Delta\ Class \bullet Enrol \wedge Idoperation$$

A problem exists when trying to implement these schemas. In the schema Idoperation, the statement $\theta Class'$ effectively uses a binding that is calculated in the schema Enrol. This is legal in Z because it is in scope, but causes problems when a procedural language like Ada is used. In order to successfully implement these schemas, the statements involving $\theta Class'$ must be implemented in Idenrol and not in Idoperation. In order to achieve this, the class must be output from the procedure `IdOperation`, which is identified by using `range_of` for the input Id. The approach is not structure preserving, but this is the case in general as the structure of Z specifications does not (as they are currently written) map well to Ada implementations [Read92].

```
procedure IdOperation(i:in Id; c:in out the_class ) is

begin
  if is_a_domain_element(i,many)
  then
    range_of(i,c,many);
  else
    raise invalid_id;
  end if;
end;
```

The procedure `Idenrol` can now use the correct binding for Class to enrol the student. The function 'Many' must be updated after `enrol`.

```
procedure Idenrol(s:in student;i: in id) is
    c:the_class;
begin
   IdOperation(i,c);
   enrol(s,c);
   unbind(i,many);
   bind(i,c,many);
   exception
       when enrol_error => raise mclass_enrol_error;
end;
```

6.10 Schema Calculus

Many specifications use existing schemas in the definition of 'new' schemas. This can be achieved because Z specifications can be decomposed into relevant parts and schema calculus can be used to combine the parts to create the full specification. This helps a specifier to concentrate on small parts of a system without getting bogged down by trying to specify a complete system in one go. In Z, there exist a number of operators that can be applied to schemas in order to manipulate them in this way. Collectively the operators form a schema calculus which is analogous to the predicate calculus of typed set theory. This section will take each operation available in the schema calculus and describe how refinement takes place. In many cases, the refinement is trivial, however, this section has been included because schema calculus is an important aspect of Z. Also, when Z specifications are

prototyped using functional languages only a subset of the schema calculus operators are used. In [Morr92, Knot92] the only operators that are available in the method are negation, conjunction and disjunction.

6.10.1 Schema Renaming

Schema renaming is a convenience operation that allows a specifier to create a new schema from an old one by the replacement of selected variables. In some cases of schema renaming, the old schema can be thought of as a template, whilst the new schema can be thought of as an instance of the old schema parameterised over the new variables. For schema renaming, no refinement need take place as it is possible to use just the result of the renaming action.

6.10.2 Schema Piping

Schema piping is another technique to build larger schemas from smaller ones. Basically schema piping takes the output of one schema and 'pipes' it through as the input to the next schema. The problems with implementing the schema piping operator involve the management of the input and output variables of the new schema and the constituent schema.

In general, when implementing R = S >> U

- The procedure R gets all the input parameters of S and all the output parameters of U.
- Additionally R gets the inputs for U if they are unmatched as outputs of S.
- R gets the outputs of S if they are unmatched as inputs of U.
- Where the outputs of S are used as inputs to U they can be declared locally for R and the result used in a call to U.
- Variables that are inputs for S and outputs for U, will have the parameter mode `In Out` in R.

In following the above rules, the parameter list will contain a minimum of information, whilst being sufficient to call S and U from R.

6.10.3 Schema Inclusion

The name of a schema can be included in the declaration part of another schema. The effect is for the included schema to be physically imported in the text of the including schema. Its declarations are merged with those of the including schema and its predicate part is conjoined with the predicate part of the including schema.

NB. The Ada translation of the including and included schemas may appear in different Ada packages. The refinement of schema inclusion amongst schemas in different packages was be discussed along with state extension in the section regarding child packages in chapter 5 section 5.5.

An example of inclusion is as follows :-

```
State ____________
| X : P
| Y : I ⇸ J
|------------
| Dom Y ⊆ X
|____________
```

```
Op ______________
| Δ State
| n? : I
|--------------
| X' = X ∪ {n?}
|______________
```

```
NewOp ________________
| Op
| m? : J
|--------------------
| m? ∉ ran Y
| Y' = Y ∪ {n? ↦ m? }
|____________________
```

The procedure Op can be implemented as :-

```
procedure Op (n:in I) is
begin
  Insert(n,x);
end;
```

The schema NewOp which includes Op is refined as :-

```
procedure NewOp(n: in I; m: in J) is
begin
  if not is_a_range_element (m,Y) then
    Op(n);
    bind(n,m,Y);
  end if;
end;
```

Since the predicates are conjoined, the propositions in the precondition of Op and NewOp must be true before the operations translating the postcondition are implemented. In this example, the propositions in the precondition of NewOP are implemented first. If the precondition of NewOp is valid then Op can be called. It does not matter that the postcondition from Op is implemented before NewOp since no order is forced on the statements in the postcondition.

In general when a schema A includes a schema B, they are implemented as follows:-

```
Procedure B ( parameters of B) is
begin
   ----
end;

procedure A (parameters of A and B) is
begin
   Precondition for A
   B (parameters of B)
   postcondition of A
end;
```

6.10.4 Schema Conjunction

Schema conjunction is an operation that is similar to schema inclusion. This is because the declarative parts of the schema are merged and their predicate parts

conjoined. Schema inclusion imports the definition and predicate parts of the included schema for use in the including schema, whereas schema conjunction creates a new schema from the composite schemas. Since a new schema is created, a slightly different approach is required. In general, suppose $C = A \wedge B$

where A and B were implemented as :-

```
procedure A ( ---- ) is        procedure B is (----)
begin                          begin
   ---                            ---
end;                           end;
```

Then C would be implemented in the following manner (note some code duplication may take place if A and B are not independant) :-

```
procedure C (parameters of A and B) is
begin
  Precondition of A;
  B (parameters of B);
  postcondition of A;
end;
```

In the example above, if the preconditions of A are not true then an exception can be raised and program control will not arrive at the postcondition. Similarly for the precondition of B. The only way that program control moves to the postconditions of A and B, is if the preconditions of A and B are true.

6.10.5 Schema Disjunction

The disjunction of two schemas is a new schema with the signatures of the two schemas merged and their predicate parts disjoined. Once again, the schemas must be compatible. Schema disjunction can be written as $C = A \vee B$. In order to refine C (if C is deterministic) a condition must be found that separates the behaviour of A from that of B. If C is not deterministic then no condition separating the behaviour of A and B can be found. The schema C will be refined as follows :-

```
procedure C  ( parameters of A and B) is
begin
  if determining condition then
       A ( parameters of A);
  else
       B ( parameters of B);
  end if;
end;
```

The following example of disjunction is an extract of the telephone system database.

The state schema is :-

Phone_Db

members : $\mathbb{F}$ PERSON
telephones : PERSON $\leftrightarrow$ PHONE

---

dom telephones $\subseteq$ members
(ran telephones) $\leq$ maxNoLines

Adding an entry into the phone database depends on whether the phone is on a new or existing telephone line.

AddEntry_NewLine

Δ Phone_Db
name ? : PERSON
newNumber ? : PHONE

(ran telephones) < maxNoLines
name ? $\in$ members
newNumber $\notin$ ran telephones
They must be a staff member, the line must be new
------
telephones′ = telephones $\cup$ {name? $\mapsto$ newNumber? }
members′ = members

AddEntry_ExistingLine
Δ Phone_DB
name ? : PERSON
number ? : PHONE

---

name ? $\in$ members
name ? $\in$ ran telephones
(name ? $\mapsto$ number) $\notin$ telephones
They must be a staff member, the line must exist and this person must not already be allocated to it
------
telephones$'$ = telephones $\cup$ {name? $\mapsto$ number? }
members$'$ = members

An operation to add a line is a combination of these two:-

AddEntry = AddEntry_NewLine[number? / newNumber?] $\vee$ AddEntry_ExistingLine

The schema AddEntry is deterministic because there is a condition that separates the behaviour of AddEntry_NewLine over AddEntry_ExistingLine. By examining the precondition, it can be seen that the proposition *name ? $\in$ ran telephones* says that the line must already exist. Whilst the proposition *newNumber $\notin$ ran telephones* says that the new number must not exist. Between them they effectively state which schema should be used. AddEntry can be refined as :-

```
procedure AddEntry( parameters of AddEntry_Newline and
                          AddEntry_ExistingLine) is
begin
   if is_a_range_element(name,telephones)
   then
      AddEntry_ExistingLine(----);
   else
      AddEntry_NewLine (----);
   end if;
end;
```

6.10.6 Schema Override

Schema override is written as $A \oplus B$ and is defined by:-

$$(A \wedge \neg \text{ pre } B) \vee B$$

The schema override operator can be implemented by expanding the schema according to the definition above and using the correct Ada program control structure.

6.10.7 Schema Hiding

The schema hiding operator hides the specified variables so that the variables listed are removed from the declaration and become local variables to an existential quantifier. Schema hiding can be used to remove input variables from the system and allow the system to arrive at a value for the variable itself. This technique may reduce input effort, but it makes the process of refinement with this method more difficult, as it introduces non-determinism to the specification. An example of schema hiding is as follows :-

```
┌─ Add_record ──────────────────────────
│ Δ Database
│ name? : Person
│ number? : Number
├────────────────────
│ number ∉ dom members
│ members′ =  members ⊕ { number? ↦ name ?}
└───────────────────────────────────────
```

The statement **NewAddRecord = Add_record \ [number?]** results in the following schema with number hidden, but existentially quantified:-

```
┌─ NewAddRecord ──────────────────────────────
│ Δ Database
│ name? : Person
├────────────────────
│ ∃ number : code • number ∉ dom members
│               ∧ members′ =  members ⊕ { number ↦ name ?}
└─────────────────────────────────────────────
```

The schema NewAddRecord would cause difficulties for the implementor because the number required is specified as being generated by the system, but no details as to how this is done are given. The set of codes from which the number is drawn is probably an infinite set, which will also cause problems when using a generic iterator or the `there_exists` function as these operations iterate over finite sets. For specification purposes, the way in which the number is arrived at in this case is irrelevant. Burdening the specification with these details would result in over specification. However, for implementation purposes this extra information is crucial. If this schema is to be refined as it is, then some extra code must be included to arrive at the new number. The ideal situation would be one where the new number remained as an input to the schema. In general then, schema hiding results in a loss of determinism, which has to be avoided when refining specifications with this method. Alternatively a global operation could be included in the schema, for example 'get_code_number'. This operation would then allow the implementation details to be left out of the schema. The schema would be implemented with 'get_code_number' as a generic procedure to the procedure NewAddRecord. The developer could then decide upon the form of 'get_code_number' and instantiate the procedure NewAddRecord for 'get_code_number'.

6.10.8 Schema Composition

This is written as $C = A \fatsemi B$. In a programming language this statement can be read as do A then do B. However, in specification terms schema composition is not as straightforward. When schema composition is used there is an intermediate state between A and B. This is the state at which A ends and B starts. Applying A moves the state from S to S′, then applying B moves the state from S′ to S″. Applying C moves the state from S to S″, and the intermediate S′ is eliminated.

When implementing the schema composition some repetition of statements may be incurred as it is possible for pre and postconditions to be repeated in both A and B. This repetition of statements can be avoided if the schema C is expanded, and then

implemented, because after expansion duplicated statements can be identified and removed. This would also introduce the problem of the intermediate state that existed between A and B. It is possible that a precondition of the second schema may involve primed variables of the first schema. This would result in a chicken and egg situation, since all the preconditions must be calculated before the postconditions. In order to implement this, some formal manipulation would be required to arrive at the real precondition.

Care must be taken when using schema composition to write specifications that will be refined with this method. It is best to limit the use of schema composition to those schemas for which the natural programming style of composition can be applied.

6.10.9 Schema Negation

Schema negation creates a new schema with its predicate part negated. However, care must be taken when negation is applied to schemas [Woodc96]. The new schema may have properties that were not expected, because it is the property that changes and not the signature. This occurs when there are constraints upon the components in the declaration part of the schema that are not present in the property that is negated. For schema negation to be used correctly, the schema must be normalised so that the constraints that apply to the components appear in the property of the schema. For the purpose of this project, it is assumed that the schemas have been written in such a way that their negation results in the schema that was intended. It is then a simple matter to produce a new schema with the propositions in the precondition negated.

6.10.10 Schema Implication and Equivalence

Schema implication and equivalence can be formed using negation, conjunction and disjunction. If S and T are schemas then $S \Rightarrow T$ can be written as $\neg S \vee T$ and $S \Leftrightarrow T$ can be written as $(\neg S \wedge \neg T) \vee (S \wedge T)$. These two operators make use of negation, so once again care must be taken when the negation is carried out. The most useful

operators with regards to the construction of specifications are conjunction, disjunction and inclusion whilst operators like negation, implication and equivalence are not very useful.

6.10.11 Schema Projection

If S and T are schemas then $S \upharpoonright T$ is equivalent to $(S \wedge T) \setminus (x_1, ..., x_n)$, where $x_1, ..., x_n$ are the components of S that are not shared by T. The refinement of schema projection can therefore be accomplished along the same lines as for schema hiding discussed above, but by taking into account the extra constraint.

6.10.12 Universal and Existential Quantification of Schemas

The universal and existential quantifiers can once again be used to create a new schema from old ones. If S and T are schemas then $\exists S \bullet T$ and $\forall S \bullet T$ denotes a schema whose signature is that of T with all the names of the signature of S removed. The property of the new schema is then the property of T with an existential or universal quantifier operating over the variables of S constrained by the property of S.

For example, given the schemas S and T as below :-

The schema $\exists S \bullet T$ can be written as :-

$$\exists S \cdot T$$

$$J : \mathbb{Z}$$

$$\exists I : \mathbb{Z} \mid I > 5 \cdot I > J$$

This schema can then be refined using the generic operation for there exists, with the predicate part of S and that of T used together to instantiate the generic operation.

Chapter 7

Evaluation of the Method

7.1 Introduction

In this section, the method will be evaluated by taking into account how the code obtained with reusable components compares with that which could have been produced manually. Also, the code that is produced from a concrete design, obtained from an initial Z specification will be compared with code produced using reusable components from the original Z specification. A comparison of how the reusable components could have been programmed using another imperative language (C++) and a functional language (Haskell) will also be made. Finally, some conclusions of how well the method scales up to industrial sized specifications will be drawn, by looking at the steam boiler [Buss96] and aircraft cabin illumination [Hame95] specifications and their corresponding implementation.

7.2 Comparison of Code Produced Manually

In this section the code that could be produced by a manual translation of a Z specification will be compared with code produced by using reusable components. When the term 'manually' is used for the development of code, it does not imply that the alternative method of using reusable components is automated.

7.2.1 Manual Implementation of Birthday Book Specification

In this section, an Ada program specification for the extended Birthday Book (state schema is in section 5.3.1) is given, along with a possible data model that may be used. The model is based upon linked lists. The data models for NAME, DATE and YEAR are still left as generic parameters, as with the version using reusable components, because their data model is not given in the specification. As a result, generic procedures for outputting those types are also given. It would have been possible to create models for these types in the Ada specification and hence no generic parameters would be required. The linked list model is dynamic, but must incorporate an exception if it is not possible to add more data to the birthday book due to constraints in the machines memory. If this example was used in a real

application, file structures would have been used so that information is not lost when the machine is switched off. However for demonstration purposes lists shall suffice.

7.2.2 Manually Implemented Ada package specification for birthday Book

The Birthday Book could be implemented using linked lists as follows:-

```
------

type birth_day;

type birthday_list is access birth_day;

type birth_day is record
    the_name     : name;
    the_date     : date;
    next         : birthday_list;
  end record;

type card_sent;

type card_sent_list is access card_sent;

 type card_sent is record
   the_year      : year;
   domain        : birthday_list;
   next          : card_sent_list;
end record;

type card_received;

type card_received_list is access card_received;


type card_received is record
  the_name       : name;
  the_year       : year;
  next           : card_received_list;
end record;

birthday        :birthday_list;
cardsent        :card_sent_list;
cardreceived    :card_received_list;

end;
```

7.2.3 Manual Implementation of Add_Birthday Operation

An implementation of the operation `Add_Birthday`, implemented manually with the linked list data model could be :-

```
procedure Add_Birthday(n:name; d:date) is

  index:birthday_list:=birthday;

begin
  while index /=null
  loop
     if index.the_name = n then
        put_line("already known");
        return;               --it may be disorienting to see a
                              --return here but see [Booc87]p264
     else
        index:=index.next;
     end if;
  end loop;

  birthday:=new birth_day' (the_name=>n,
                            the_date=>d,
                            next    =>birthday);
  put_line("ok");

  exception
    when storage_error => raise overflow;
                 --the linked list,although
                 --dynamic, is finite so an
                 --exception is raised if memory is used up
end;
```

In the code above, it is necessary to loop through the list structure for `birthday` to check that a duplicate entry is not made. If the entry is not already there then the persons birthday is added to the list.

7.2.4 Implementation of Add_Birthday using Reusable Components

The same operation can be implemented by direct translation using operations from the reusable components. The state schema for the expanded birthday book specification is on page 66 whilst the state variables are declared on page 68. Add_Birthday is translated as follows:-

```
procedure Add_Birthday(n:a_name; d:date) is
begin
  if not is_a_domain_element(n, birthday)
  then
    bind(n,d,birthday);
    put_line("ok");
  else
    put_line("already known");
  end if;
end;
```

The operations `is_a_domain_element` and `bind` would be contained in a reusable Ada package modelling functions. A data model for a function based upon linked lists and the code for the operations `is_a_domain_element` and `bind` are now given to show typical pieces of code contained within the reusable components.

The data model for a reusable function component based upon linked lists is:-

```
type map;
type map_list is access map
type map is record
   the_domain: dset_pack.item_type;
   the_range : rset_pack.item_type;
   next      : map_list;
end record;
```

An implementation for the operation `is_a_domain_element` from a reusable function component based upon linked lists is as follows :-

```
function is_a_domain_element(d: dset_pack.item_type; m: map)
                                                  return boolean is

    index:map_list :=m;
begin
  while index /=null
  loop
     if index.the_domain = d
     then
       return true;
     else
       index:=index.next;
     end if;
  end loop;
  return false;
end;
```

An implementation of the `bind` operation from a reusable function component based upon linked lists is as follows :-

```
procedure bind( d:        dset_pack.item_type;
                r:        rset_pack.item_type;
                m:in out map ) is

     index:map_list:=m;
begin
   while index/=null
   loop
      if index.the_domain = d
      then
         raise domain_already_exists;
      else
          index:=index.next;
      end if;
   end loop;
   --binding does not exist so add new pair to map

   m:=new map' ( the_domain =>d,
                 the_range  =>r,
                 next       =>m);

   exception
     when storage_error => raise overflow;  --raised when the set
                                    --cannot grow large enough
                                 --to complete the operation

 end;
```

7.2.5 Comparison of reusable code against manually derived code

In the code for Add_Birthday using the reusable components, the calls to `is_a_domain_element` and `bind` effectively duplicate work by traversing the same list structure twice to check for membership of a domain item. The first check is to determine if 'name' is already a member of the domain of the function. The second check occurs because the semantics of `bind` say that a pair can only be entered into the mapping if the domain element is not already there (in the case of the many to one mapping). The `bind` operation, therefore, forces another traversal of the structure in the preservation of its semantics. This is one area where the code produced manually can be more efficient than the code produced using operations from reusable components modelling abstract data types. The problem occurs because, by definition, each operation available in the abstract data type must preserve the semantics of the abstract data type in question, so in the application of

a series of operations when translating Z statements, some duplication of work is inevitable. When code is developed manually, a developer can rationalise code in a manner that cannot be matched when using operations from the reusable components. However, as Boehm [Boeh87] states, a project manager can reduce software costs and improve software quality by using modern software engineering techniques. Boehm specifically mentions Ada with its support for modularity, information hiding and reuse. When using reusable components, errors can only be introduced at the translation stage (provided the operations in the components are error free). A manually constructed version of the software may be more efficient, but it has the opportunity to introduce errors in the actual detailed coding stage. One of the most important objectives of software engineering is to reuse existing pieces of program so that the effort of detailed new coding is kept to a minimum [Barn95 pg15]. As an example of reusable components improving software quality and reducing costs, Raytheons system of reusable software components has achieved figures of 60% reusable code resulting in savings of 10% in the design phase and 50% in the coding and testing phase [Boeh87].

7.2.6 Improving Add_Birthday Performance

A design decision was taken to raise an exception in the code for `bind` (given on page 134), when attempting to add duplicate items, the goal being to avoid the duplication of search effort in the code implementing Add_Birthday. An equivalent implementation for Add_Birthday (which follows), is more efficient. It only requires a call to the `bind` operation because `bind` handles the case where the name is already in the birthday book i.e. that name? $\notin$ birthday. The operation `bind` is an implementation of union but for a single item and with more primitive semantics. An exception is raised for a duplicate item in `bind`, whereas for the `union` operation, nothing is done as duplicate items are simply ignored.

```
procedure Add_Birthday(n:a_name;d:a_date)  is
begin
   bind(d,r,birthday);
   put_line("ok");

   exception
     when domain_already_exists = > put_line("already known");
end;
```

The code for this operation, is now very similar to the version of the code produced manually by direct manipulation of the linked list structure (if the code for `bind` is taken into account).

In the code for Add_Birthday above, `put_line` statements have been used to output messages for ok and already known in a similar manner to many other methods used throughout the literature. Woodcock [Woodc96, page 361] constructs a report type which is passed as a parameter from the operation. However, since the Z specification does not say in what form the message should be output an alternative implementation of Add_Birthday using exceptions is also possible.

```
procedure Add_Birthday(n:a_name;d:a_date)  is
begin
  bind(n,d,birthday);

  exception
     when domain_already_exists = >  raise already_known;
end;
```

If the name is already in the function `birthday`, an exception is raised by the operation `bind`. This can be caught in Add_Birthday and an exception `already_known` (which will be declared in the Birthday Book package) is exported and can be caught in the program that is using the Birthday Book package. A message will have been passed from the Add_Birthday operation without specifying its form or what action should be taken upon its receipt thus satisfying the original intent of the Z specification and following the style of Booch [Booc87]. Of course, catching and then exporting an exception could also be employed in the manually produced version. In both methods, when using reusable components and producing

code manually, the same information is lacking in the Z specification, and hence, similar decisions must be taken to account for this lack of information. When the exception `already_known` is raised the exception handler in the program instantiating and using the birthday book package could, for instance, take the form:-

```
exception
  when already_known => put_line("already known");
```

or it could request that the user try the operation again with a new name, or simply do nothing by using the null statement.

7.2.7 Conclusions

Primitive operations such as searching for items in a structure, updating a structure and deleting items from a structure will match the performance of similar operations constructed manually using the same data structures. However, when translating Z specifications directly, some duplication of effort is encountered because each operation in the reusable components must preserve the semantics of its underlying type. As a result, a complex operation schema may use a series of many operations from the reusable components in its implementation, which may duplicate a lot of work. In the Add_Birthday example it was shown that a duplication of work existed and how the use of exceptions in the construction of the operations in the reusable components could remove it in this case. However, the important point is that a manually produced version of the code has the opportunity to rationalise code in a manner that selecting a series of operations from reusable components cannot. On the other side, the manually constructed version has more opportunity of introducing errors in the detailed coding and also makes less use of reuse.

7.3 Comparison with Code Produced from Concrete Design

A library specification is given in [King89]. The specification is refined into a concrete design from which an algorithm, in Djikstra's guarded command language, is presented for one of the operations. Proof obligations are stated but are not

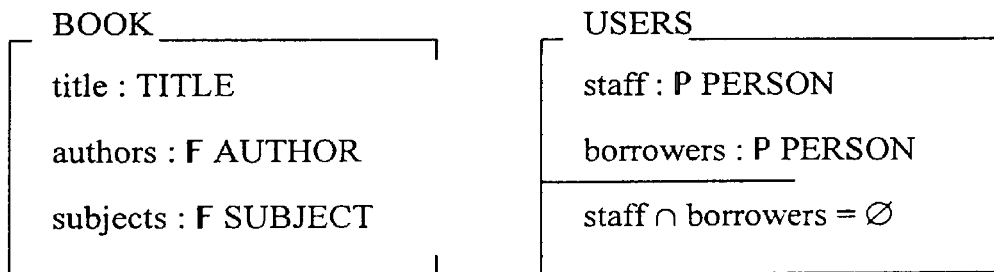
completed. This section seeks to compare the code produced for this algorithm with the code that is produced by using operations from the reusable components using the original abstract operation and state model. As a result some schemas from the specification in [King89] must be introduced.

7.3.1 Library Specification Summary

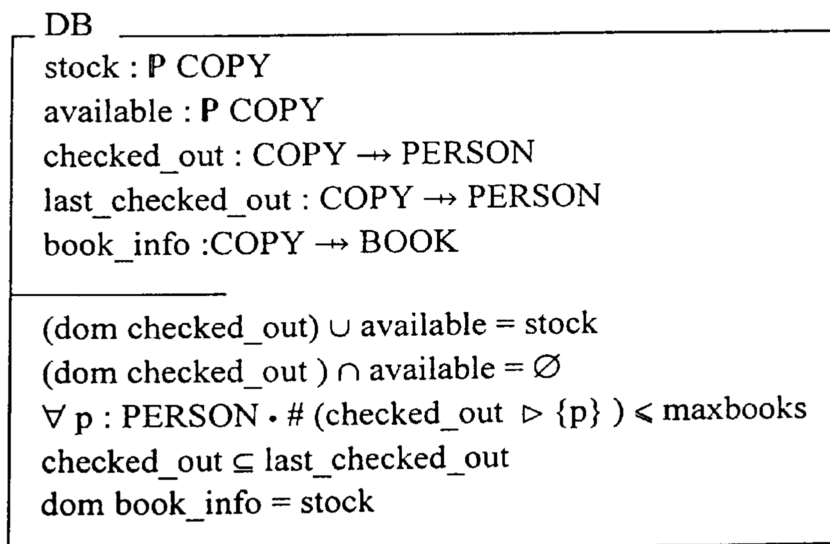
The given types and state schemas are as follows :-

[PERSON, COPY, TITLE, AUTHOR, SUBJECT]

maxbooks : ℕ



The state schema for the database is given as:-



The operation used as the main example in the paper and in this section is specified as:-

```
─ Check_out ──────────────────────────────────
│ Counter_trans
│ borrower ? :PERSON
├──────────────
│ copy? ∈ available
│ borrower? ∈ staff ∪ borrowers
│ #(checked_out ▷ {borrower? }) < maxbooks
│ available ′ = available \ {copy?}
│ checked_out′ = checked_out ∪ {copy? ↦ borrower?}
│ last_checked_out′ = last_checked_out ⊕ {copy? ↦ borrower?}
└──────────────────────────────────────────────
```

Where Counter_trans in the schema Check_out, is a schema used to check that a book is in stock.

King and Sørenson develop the operation Check_out by specifying a concrete version and then providing an algorithm in the guarded command language. A summary of the concrete design is now given for the reader to understand the concrete version of Check_out, named D_Check_out, and the resulting algorithm, named T_D_Check_out which also follows.

```
[TKN]
P_TYPE ::= staff | borrower
C_STATUS ::= in | out
C_BORROWER ::= person ⟪ PERSON ⟫ | none
```

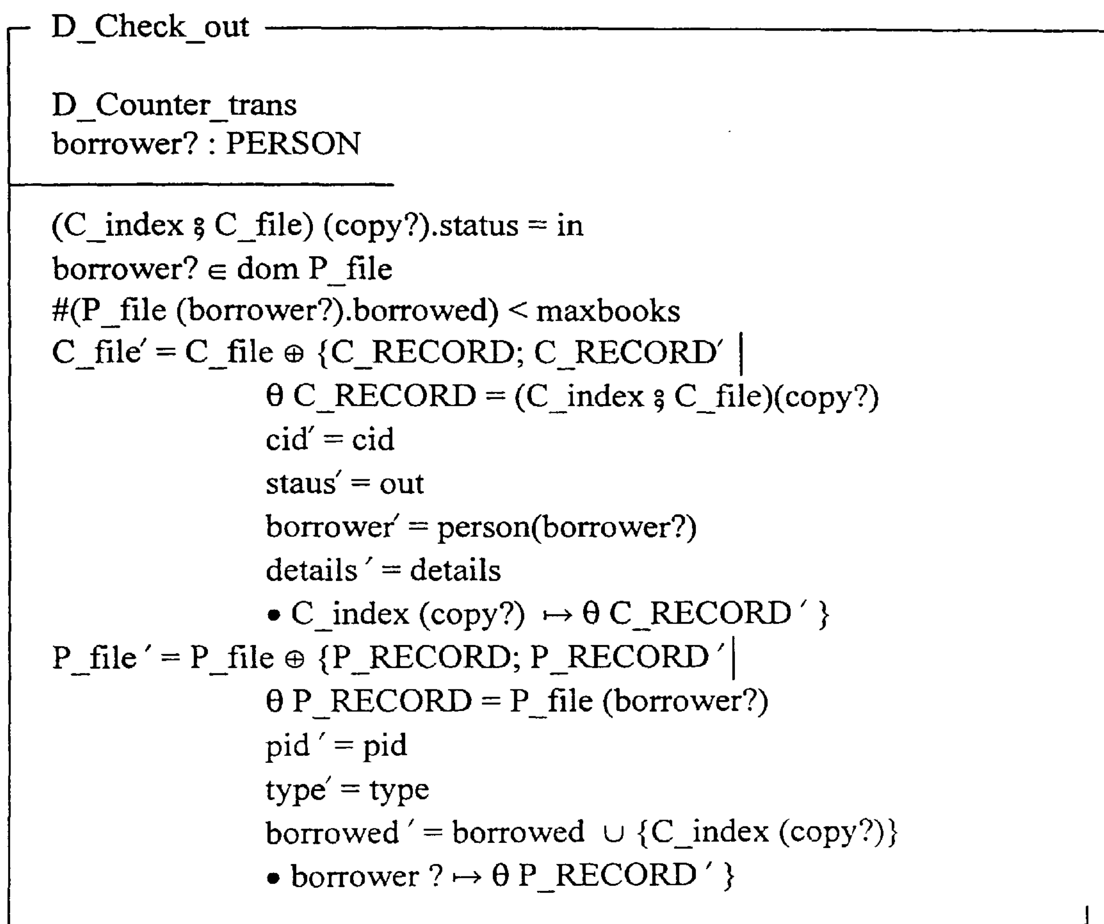
```
─ P_RECORD ──────────────
│ pid : PERSON
│ status : P_TYPE
│ borrowed : FTKN
├──────────────
│ #borrowed < maxbooks
└────────────────────────
```

```
─ C_RECORD ──────────────
│ cid : COPY
│ status : C_STATUS
│ borrower : C_BORROWER
│ details : BOOK
├──────────────
│ borrower = non ⇒ status = in
└────────────────────────
```

```
┌─ D_USERS ──────────────────────────────────────────
│ P_file : PERSON ⤔ P_RECORD
│ staff_list : 𝔽 PERSON
├──────────────────
│ staff_list = dom ( P_file ▷ {P_RECORD | type = staff } )
│ ∀ p : PERSON | p ∈ dom P_file • P_file (p).pid = p
└────────────────────────────────────────────────────
```

```
┌─ D_DB ─────────────────────────────────────────────
│ C_index : COPY ⤔ TKN
│ C_file : TKN ⤔ C_RECORD
│ Auth_list : AUTHOR ↔ TKN
│ Subj_list : SUBJECT ↔ TKN
├──────────────────
│ ran C_index = dom C_file
│ Auth_list = (C_file ⨾ λ C_RECORD • details ⨾
│                {b:BOOK; a:AUTHOR | a ∈ b.authors } ) ⁻¹
│ Subj_list = ( C_file ⨾ λ C_RECORD • details ⨾
│                {b:BOOK; s:SUBJECT | s ∈ b.subjects } ) ⁻¹
│
│ ∀ c : COPY | c ∈ dom C_index • ( C_index ⨾ C_file) (c).cid = c
└────────────────────────────────────────────────────
```

The concrete version for the operation Check_out is specified as:-



```
D_Check_out
  D_Counter_trans
  borrower? : PERSON
  ──────────────
  (C_index ⨾ C_file) (copy?).status = in
  borrower? ∈ dom P_file
  #(P_file (borrower?).borrowed) < maxbooks
  C_file′ = C_file ⊕ {C_RECORD; C_RECORD′ |
                 θ C_RECORD = (C_index ⨾ C_file)(copy?)
                 cid′ = cid
                 staus′ = out
                 borrower′ = person(borrower?)
                 details′ = details
                 • C_index (copy?) ↦ θ C_RECORD′ }
  P_file′ = P_file ⊕ {P_RECORD; P_RECORD′ |
                 θ P_RECORD = P_file (borrower?)
                 pid′ = pid
                 type′ = type
                 borrowed′ = borrowed ∪ {C_index (copy?)}
                 • borrower ? ↦ θ P_RECORD′ }
```

The process of moving to a concrete design has resulted in schemas which are more complicated than their original abstract versions, as design details are included. Indeed, the concrete schema D_Check_out bears little resemblance to the original abstract schema due to the large amount of rewriting work carried out.

7.3.2 Concrete Algorithm Design for T_D_Check_out

The algorithm which follows is an implementation of D_Check_out :-

```
T_D_Check_out
⊑

r1,r2,r3:boolean;
c_tkn : TKN;
member(staff_list, id?, r1);
domlookup(C_index,copy?, c_tkn,r3);
if  ¬ r1              → r! = "unknown librarian"
▯  ¬ r2               → r! = "unknown borrower"
▯  ¬ r3               → r! = "book not in stock"
▯ r1∧ r2 ∧ r3         →
    PR:P_RECORD;
    CR:C_RECORD;
    r4,r5:boolean;
    filelookup(C_file,c_tkn,CR,r4);
    pflookup(P_file,borrower?,PR,r5);
    lengthdll(PR.borrowed,l);
    if CR.status ≠ in     →    r! := "book not available"
    ▯ l ≥ maxbooks        →    r! :=  "too many books"
    ▯ CR.status = in      →    adddll(Pr.borrowed,c_tkn);
        ∧ l < maxbooks         pfinsert(P_file,borrower?,PR);
                               CR.status := out;
                               CR.borrower := borrower?
                               fileinsert(C_file,c_tkn,CR);
                               r! = "ok"
    fi
fi
```

7.3.3 Comparison of Code Obtained from Components and Concrete Design

In order to compare the code produced between the two methods, each of the Z operations from the original abstract version of Check_out and their respective implementations using reusable components shall be examined, in turn, to see if they can match the performance of the concrete design algorithm T_D_Check_out.

The following three statements can be executed extremely efficiently if a reusable component based upon a Btree or an equivalent is used to store the information on a file. The statements in courier font are the implementations for the Z statements.

id? ∈ staff	`is_a_member(id,staff)`
copy? ∈ stock	`is_a_member(copy,stock)`
copy? ∈ available	`is_a_member(copy,available)`

The next statement requires a union operation to be performed.

borrower? $\in$ staff $\cup$ borrowers

```
union(staff,borrowers,temp_set)
is_a_member(borrower,temp_Set)
```

The act of carrying out a union operation will undoubtedly be less efficient than the code produced by the algorithm T_D_Check_out. However, in the concrete design a type is introduced to mark a person as a member of staff or a borrower in the following way :- *P_TYPE ::= staff / borrower*. The relevance of being a staff member or a borrower is then reduced as the concrete operation D_Check_out is only interested in legal users regardless of status as implied by the statement *borrower?* $\in$ *dom P_file*. Here, two sets are effectively combined and there is no need for a union operation.

If A and B are large sets, the difference in performance between a manual version discussed above and the code obtained from reusable components will be large. A discussion on the performance of the union operation and algorithms to improve the performance of the operation are given in chapter 8 of [Weis93]. If the implementation of this operation causes a bottle neck in system performance then using a data structure that is well suited to the `union` operation may improve the performance to a point that is satisfactory. However, a better solution would have been to avoid the union operation by checking for membership of a person in the set of staff <u>or</u> in the set of borrowers. The statement could be rewritten in the form:-

$$(x \in A \lor x \in B)$$

Rewriting the specification statement in this form will have the same result as the original statement, but it may be argued that performance concerns are being introduced into the specification document by making the statement less abstract. However, this refinement may be necessary for some statements if the code is to be used not just for prototyping purposes but as the final system.

Indeed, King and Sørenson use a similar technique when creating the algorithm T_D_Check_out. For example, the statement from D_Check_out

$$(C_index \mathbin{⨾} C_file)(copy?).status = in$$

is implemented in T_D_Check_out by :-

```
dirlookup(C_index,copy?,c_tkn,r3);
filelookup(C_file,c_tkn,CR,r4);
if CR.status ≠ in → r!:="book not available"
```

where as a reminder:- $C_index : COPY \rightarrowtail\!\!\!\!\!\!+ \; TKN$

$C_file : TKN \rightarrowtail\!\!\!\!\!\!+ \; C_RECORD$

Here, King and Sørenson did not attempt to form the relational composition of the two functions in their algorithm design because doing so would have used up much processing time by visiting every copy and every token, TKN, in order to create a new mapping. The objective of the statement is to inquire as to whether a book is in the library. This was easily achieved as King and Sørenson have done by looking through C_index to get the token, C_tkn, for the copy in question and then searching for the token in C_file to access its data and hence its status as in or out of the library.

Returning to Check_out, the statement

$$\#(checked_out \rhd \{borrower?\}) < maxbooks$$

is refined by:-

```
range_restrict(borrower,temp_map,checked_out)
size_of(temp_map) < maxbooks
```

Here, the mapping `checked_out` is restricted to those range elements that match `borrower?` and the number of items in that restricted mapping must be less than `maxbooks`. The operation implementing range restriction traverses the mapping and when a range item in the mapping matches `borrower` it is placed into a new

mapping (temporary in this overloaded version of `range_restrict`). The operation `size_of` simply returns the size of the mapping which is held in a record field of the data structure, so that a traversal of the new mapping is unnecessary. The inefficiency here is that range items may be duplicated through the entire mapping. It is thus necessary to traverse the entire mapping to access all people that may have borrowed a particular book.

Another inefficiency is in the actual creation of the mapping, which must be stored in order to apply `size_of` to the new mapping. It would be more efficient in terms of memory usage to traverse the structure and simply update a count of the number of times the range of the mapping matched the item(s) in question. This can be achieved, if desired, by use of the iterator procedure in a similar manner to that of section 6.7.4. However, the link between the Z statement and the resulting Ada code will then not be as obvious and some complexity will have been introduced by the inclusion of a subprocedure and the instantiation of the iterator operation using that subprocedure. However, in this case creating a temporary mapping will not have much affect because the new mapping will not be large as the number of books a person has checked out is likely to be small.

Even when using the iterator procedure to model the statement, the efficiency of the new concrete design cannot be matched. In the concrete design, each person is modelled as a record which has a field containing the set of books that they have borrowed. In the algorithm for the concrete design, all that is required is to search for the person in question, access the correct record field and then count the number of books that the person has. The efficiency of this statement cannot be matched by using the operations available in the reusable components because the state model in the original abstract Z specification is not a design and was not written with efficiency in mind. The only alternative to improving the efficiency of this operation would be to change the state model, or add information such as adding the function $Borrowed_by : Person \twoheadrightarrow \mathbb{F}\, Copy$. In doing this, the performance of the operation can be matched, but this results in a duplication of data.

The remaining statements from Check_out

$available' = available \setminus \{copy?\}$

$checked_out' = checked_out \cup \{copy? \mapsto borrower?\}$

$last_checked_out' = last_checked_out \oplus \{copy? \mapsto borrower?\}$

are refined respectively by :-

```
delete(copy,available)
bind(copy,borrower,checked_out)
function_override(copy,borrower,last_checked_out)
```

These statements are as efficient as a manual implementation (since they are either inserting, deleting or updating a pair in a mapping) provided the reusable component chosen has the same underlying data structure as was chosen for the manual version.

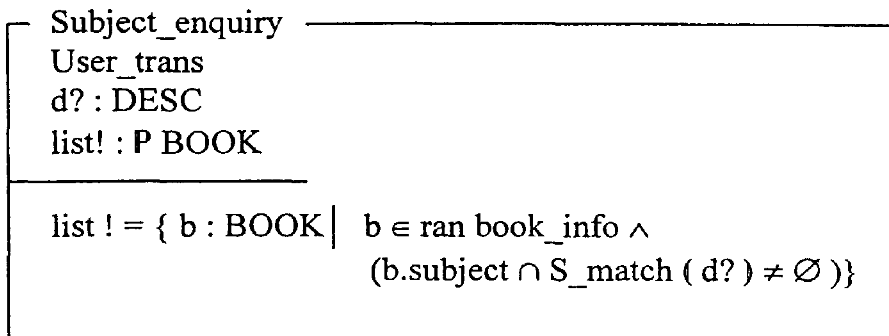
7.3.4 Inserting Extra Data Structures

In the text above, it was claimed that the performance using reusable components could only match the algorithm derived from the concrete design if the state model was refined in some manner or if an extra structure was added, due to the `range_restrict` operation. However, in developing the concrete design this is perfectly allowable. Two operations in the concrete design required traversing the whole of the COPY file to find matching records, so for efficiency reasons, King and Sørenson built two more indexes. The original abstract specification for these two operations, Subject_enquiry and Author_enquiry, are as follows :-

[DESC]

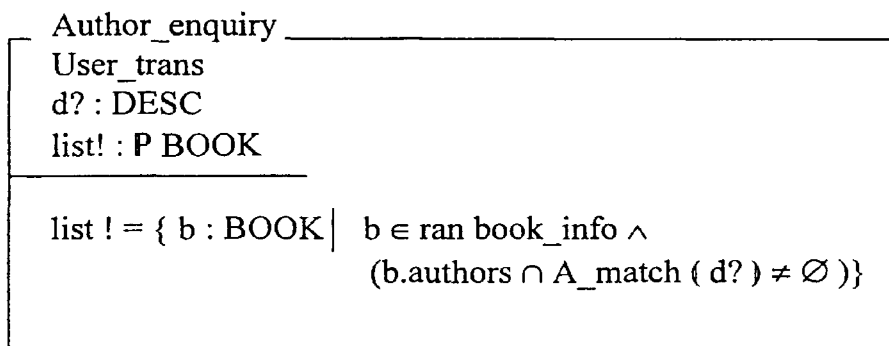
$A_match : DESC \leftrightarrow AUTHOR$

$S_match : DESC \leftrightarrow SUBJECT$



Subject_enquiry
User_trans
d? : DESC
list! : $\mathbb{P}$ BOOK

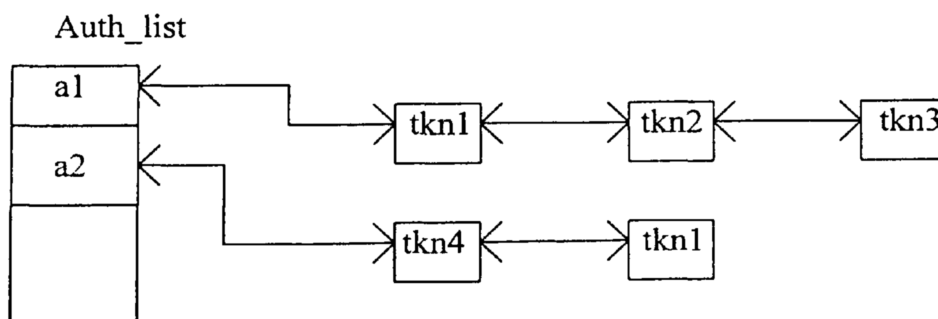
list ! = { b : BOOK | b $\in$ ran book_info $\wedge$
(b.subject $\cap$ S_match (d?) $\neq \varnothing$)}



Author_enquiry
User_trans
d? : DESC
list! : $\mathbb{P}$ BOOK

list ! = { b : BOOK | b $\in$ ran book_info $\wedge$
(b.authors $\cap$ A_match (d?) $\neq \varnothing$)}

The problem is that in order to find all the books whose authors match a given description, the whole of the COPY file (in the concrete design) would have to be searched to see which records matched. The two additional indexes add information to the specification as follows :-



Auth_list : AUTHOR $\leftrightarrow$ TKN

A similar structure would exist for subjects. Each Author is related to some tokens which are keys to the COPY records.

As for implementing the operation Author_enquiry with reusable components, the set comprehension procedure available in the reusable components can be used Implementing the operation $list! = \{ b : BOOK \mid b \in ran\ book_info \wedge (b.authors \cap A_match(d?) \neq \varnothing)\}$ is a straightforward example of using the set comprehension procedure (see section 6.7). However, the resulting code is not as efficient as the code produced by the concrete design because, once again, additional information, in the way of extra data structures have been added to create the new concrete Z specification. If the performance of this operation is not satisfactory using the reusable components, then a similar action of adding a new data structure to the state schema can be performed. In this example, this can be achieved by adding the following statement to the state schema.

Wrote : Author $\rightarrowtail\!\!\!\!\!\rightarrow$ P BOOK

However, BOOK itself contains a power set of authors as a field, so some information has been duplicated but information has also been duplicated in the concrete design since:-

Auth_list : AUTHOR $\leftrightarrow$ TKN

C_file : TKN $\rightarrowtail\!\!\!\!\!\rightarrow$ C_RECORD

where C_RECORD is defined as :-

```
┌─ C_RECORD ──────────┐
│ cid:COPY
│ status : C_STATUS
│ borrower : C_BORROWER
│ details : BOOK
├──────────────
│  - - - - - - -
```

7.3.5 Improving Performance

In writing Z specifications a developer with some experience of using the reusable components to translate the Z specification will know that using statements such as `range_restrict` will result in an inefficient translation for functions and relations. The reusable components modelling functions and relations can use the domain element as a key for efficient searching of domain items, insertion of pairs into the mapping and the retrieval of range items for a given domain item. However, searching for range items in a function or relation, with the domain as the key field is not efficient as the range items may be scattered throughout the data structure.

If performance of the code, translated from an abstract specification, is not satisfactory, then it must be tuned. However, predicting the efficiency of a system or parts of a system is a difficult task. Booch [Booc87] states that characterising the space and time complexity of components within a system is highly dependant on the compiler used. A good compiler that is used on well designed high level code will produce optimised machine code. He also states that it is often too difficult to predict the space and time complexities of a large system and that it is more common to build a prototype of the system and then collect the necessary performance data. It is not practical or necessary to achieve the best possible performance for every part of a large system and it is sufficient to isolate performance bottlenecks and then tune them. The work detailed in this thesis can also be used in this role.

In his summary on the approach to code tuning McConnel [McCo93] lists the following steps as a guideline.

1. Develop the software using a good design, with highly modular code that's easy to understand and modify.

2. If performance is poor, measure the system to find hot spots.

3. Determine whether the weak performance comes from inadequate design, data structures or algorithms and whether code tuning is appropriate. If code tuning isn't appropriate, go back to step 1.

4. Tune the bottleneck identified in step 3. Measure each improvement, and if it does not improve the code take it out.

Repeat from step 2.

Step 3 advises the developer to determine if the poor performance is due to the design, the data structures or the algorithms. The last two can easily be changed by selecting different reusable components based upon different data structures and containing more efficient algorithms. An improvement in the design of operations can be made by respecifying an operation, since in Z there are many ways of expressing the same idea, but they may not be equal in the time taken to evaluate the statements using the operations from the reusable components. These refinements will be easier to do if the specification is written in a modular style since implementing a state schema with different reusable components or improving the design of a schema will not adversely affect the other modules in the system. This is why it is advantageous to program in a modular style. More seriously, if the bad performance is down to the design, it may be necessary to construct a more concrete design using a more efficient state model. However, this more concrete version, whilst taking into account design details, will still be written in Z and as a result, operations from the reusable components can still be used. The concrete design of the library system in section 7.3 can be used for translation using reusable components, since the state schema still defines functions and sets. Functions such as $C_file : TKN \rightarrowtail\!\!\!\!\!\!\rightarrow C_RECORD$ could be implemented using a function component based upon a Btree with *TKN* as the domain field, or the key field of the Btree and *C_record* as the range field containing the data.

7.3.5.1 Implementation of T_D_Check_out Using Reusable Components

The concrete state schema can be implemented in the same manner as the abstract state schema, since it still uses sets and functions, although with a more improved model to access data. The algorithm T_D_Check_out (on page 142) contains procedure calls, such as `member` and `domlookup`, which have direct counterparts to operations within the reusable components. It would also be perfectly possible to translate T_D_Check_out because all the operations stated in the algorithm are available as operations in the reusable components, as is shown below:-

`member(staff_list, id?, r1);`	Equivalent to the `is_a_member` operation from the components.
`domlookup(C_index,copy?,c_tkn,r3);`	Equivalent to the `range_of` operation.
`filelookup(C_file,c_tkn,CR,r4);`	Equivalent to the `range_of` operation.
`pflookup(P_file,borrower?,PR,r5);`	Equivalent to the `range_of` operation.
`lengthdll(PR.borrowed,l);`	Returns the length of dynamic linked list. Equivalent to # operation in Z and `size_of` function from the reusable components.
`adddll(Pr.borrowed,c_tkn);`	Adds an item to a linked list. This is equivalent to the `insert` operation.
`pfinsert(P_file,borrower?,PR);`	Equivalent to `bind.`
`fileinsert(C_file,c_tkn,CR);`	Equivalent to `bind.`

In some cases above, it is stated that the `range_of` or `insert` operation is equivalent to an operation contained in the algorithm T_D_Check_out, even though in the algorithm different names are used, such as `pfinsert` and `fileinsert`. However, versions of `bind` will operate over different types depending on which package instantiation they came from. So for example, `person_set_pack.bind` would

correspond to `pfinsert`, whilst `copy_set_pack.bind` would correspond to `fileinsert`.

7.3.6 Conclusions

The library specification given in [King89] is easily translated into code using reusable components. However, it has been shown that for some operations the performance of the code obtained by using reusable components is worse and may not be satisfactory in all cases. This has not been due to the way in which the code for the operations in the reusable components have been constructed, but in the way in which the state has been modelled and how, subsequently, the Z statements have been written. If it is not possible to add some extra information, due to memory restrictions, or make improvements to Z statements, in order to speed up the implementation, then moving to a more concrete design, such as that given in [King89] will be necessary. The original implementation of the abstract design will not be wasted as it can be used as a rapid prototype to validate the original specification and customer requirements. The operations from the reusable components are still useful because they can be used in the refinement of the concrete design.

7.4 Other languages for Modelling Z

7.4.1 C++

Ada95 and C++ are both general purpose languages that provide features that modern software engineering practice deems indispensable: modularity; information hiding, inheritance and support for object oriented designs and mechanisms for parameterising software components. In writing the reusable software components using Ada95, much use is made of the generic paradigm. In order to construct the reusable components with C++, in a similar fashion, it is necessary to investigate the generic facilities of C++. In C++, templates are used to create generic software components. A typical component for a set would be set out as follows:-

```
template <class Item_type>    //class definition of a set

class set {

   public
      //set operations

private
      //data model
```

Since set is a template, each member function must also be a template e.g.

```
template <class Item_type>    //definition of member function
function
    boolean is_a_member (I Item_type, s Set);
```

Using templates in this manner allows generic packages to be constructed and parameterised over different types. The set class would be used to create an instance of the set e.g.

```
Set<name> S;
```

This statement would construct a set of type name, where the model for name would have to be declared. This is equivalent to the Ada statements as follows :-

```
package name_set is new set(name);
S : name_set.set;
```

In C++, it is also possible to include function parameters that depend on a class parameter, which is equivalent to generic formal subprograms in Ada95. It is then possible to include an ordering or hashing function to improve efficiency of the components in C++.

Some operations in the reusable components need to be parameterised over functions or procedures as required by statements in the Z specification, such as set comprehension terms and quantifiers. In C++ pointers to functions can be passed as

parameters to the operation in question. The universal quantifier defined as $\forall x : S \cdot$ *Predicate(X)* can be implemented as :-

```
boolean predicate (Item_type)
    for_all (S, &predicate)
```

Where `for_all` contains code that traverses the set S and applies the function `predicate` to each member. The equivalent Ada program specification is given on page 95.

One feature available in Ada and lacking in C++ is the ability to instantiate a generic package with another package. In this way the type of the package and its operations, (including its generic operations), are available. In chapter 4 the construction of function, relation and sequence packages is discussed with respect to them relying on a set of domain and a set of ranges types for some operations. Importing set packages as generic parameters to a package implementing a function is shown on page 51. An equivalent using C++ is not possible in this manner. It would be possible to import a non generic set as a property class. However, if this is done, the function cannot be generic because the types will only match in one instance. This facility was also missing in Ada83 and this forced the developer to include the types and operations as generic parameters, which led to a long and complicated instantiation process (see page 199). However, the C++ syntax forces repeated mentions of all the template parameters in every member function body of a class template. So, if a similar method to the one necessary with Ada83 was used <u>every operation</u> available in the software component would have a very long and complex parameter list.

The paper by Minkowitz et al [Mink95] introduces a C++ library for implementing specifications. The only types created are the types set and pair. The type set is constructed as a class and made generic using the template facility. The type pair is

also parameterised and is used to create Cartesian pairs. The following constructor is used to create the pair type :-

```
pair < T1, T2> Pr = pair <T1, T2> (x,y)
```

This creates a pair, `Pr`, with first element of type `x` and second element of type `y`. A function can then be created by declaring a set of pairs in the following manner:-

```
set <pair < domain, range > > s;
```

Minkowitz et al propose the following method when refining operation schemas.

- Copy all state variables into temporary variables.
- Apply the Z operations to the temporary variables.
- Test the types of the temporary variables to ensure that the types defined in the state schema are preserved.
- Test the temporary variables against the state invariants.
- If the temporary variables preserve the state invariants copy them to overwrite the original state variables.

This methodology is very inefficient because of the amount of data copying that must be undertaken. Also, testing the types of the state variables is time consuming because each item in a state variable must be tested to ensure that it preserves the type. For example in testing that a one to one function type is preserved, it is necessary to traverse the entire structure to ensure that each domain and range item only makes one appearance. This test is completely unnecessary if the software components are constructed as abstract data types. This is because every operation available for use within each component has been coded to preserve the underlying type as a matter of course.

The problem encountered by Minkowitz et al is due to the fact that the only type constructed in their library is the set, with functions, relations and sequences being

derived from the set class. However, the semantics of set are not sufficient to use as a basis for implementing functions, relations and sequences in the same class. It is evident that separate classes for implementing the other types are not used as the following example from [Mink95] shows. A function is defined in the state schema for a telephone exchange as follows:-

Calls : PHONE⇸PHONE

This is implemented using the set class as:-

```
Set< Pair< PHONE,PHONE> > Calls
```

An operation schema contains a statement to update the Calls function as follows :-

Calls′ = Calls ∪ { (caller?, receiver?) }

This is implemented using the C++ library as :-

```
Set< Pair < PHONE,PHONE > > New_Calls =
      Un( Calls,  Set < Pair < PHONE,PHONE > >
            (Pair < PHONE, PHONE > (caller, receiver)))
```

Where `New_Calls` is a temporary state variable which is copied to `Call` after the state invariant test function has been satisfied. This union operation from the set package is not sufficient to preserve the type of the function. The following example illustrates this :-

Suppose the following set, models an injective function and is enumerated as :-

$\{1 \mapsto \text{Bob}, 2 \mapsto \text{Fred}\}$

Consider performing a union operation using this function and the function enumerated by $\{3 \mapsto \text{Bob}\}$. A duplicate item is not being added, so the semantics of

set would allow the union. However, the resulting function is not an injective function anymore. The result is :-

$\{1 \mapsto \text{Bob}, 2 \mapsto \text{Fred}, 3 \mapsto \text{Bob}\}$

Performing another union with the function enumerated as $2 \mapsto$ Joe would again be allowable using set semantics, because no single pair (i.e. item in the set) is duplicated. However, now the result is a relation.

It is unclear as to whether operations such as domain restrict, which use a set as an operation parameter, are implemented in the library or indeed if it is possible to implement operations of this type in a set package. A function is modelled by parameterising the set over a pair, of type domain and range, but how would the type set of domain or set of range be made available to the operation in question in a set class? A set would be being used within its own definition. Higher order functions such as the one implemented using Ada95 components in section 5.3.1 would undoubtedly be difficult to implement with the C++ library of Minkowitz et al as it currently stands. The solution using Ada95 is to create separate set and function packages and allow the function package to import two generic set packages for the domain and range sets. In chapter 4.4, the construction of a function package based upon a set package was discussed. This was a different method to that of Minkowitz et al because the function package was a separate concern and used only the types and primitive operations from the set package, as a result the function package had different semantics.

Minkowitz et al justify the creation of a function to test the system invariant and state model as an extra integrity test. In chapter 8, the state invariant is implemented using Ada components. However, it is implemented as a child package, which can be used for testing purposes, but does not have to form part of the final implementation. In C++ the friends concept is similar to child packages in Ada95.

In concluding, the authors state that program efficiency of the C++ library wasn't considered during its construction, but more efficient implementations are possible. They also argue that if using the library on an entire system is impractical, due to critical performance requirements within the system, even using the library on a small part of the system will result in productivity gains.

The target language chosen for this research project was originally Ada83. The reusable components were translated to Ada95 to make use of the new more powerful generic paradigm available in Ada95. The facilities of C++ would enable a developer to follow the methods used in the construction of the reusable components using Ada83 (see chapter 8). However, C++ does not allow a package to be instantiated over another package. This is an important aspect in allowing sets to be used as parameters in operations over other Z types and in the construction of higher order functions. It may be possible for a developer with much C++ experience to achieve a similar result perhaps through the use of pointers. At this stage it remains unclear as to how this could be achieved as the work of Minkowitz appears to be in its early stages.

The following comparison between Ada95 and C++ can be made:-

- Both languages allow modern concepts of information hiding, modularity and reuse in the construction of software components.
- The generic operations in the Ada95 components can be modelled using function templates in C++.
- Packages in Ada95 can be instantiated over other generic packages. However, this feature is not available in C++.
- Both languages are used in real world applications.
- No subsets of Ada are allowed and all Ada compilers must undergo a certification process. There is an ANSI (American National Standards Institute) standard for C, but not for C++. For example templates are not available in

some implementations of C++ [Skan97 pg. 450]. However an ANSI standard for C++ is in draft.

7.4.2 Haskell

Prolog is a functional language commonly used in the animation of Z specifications. However, much work is also being carried out using Haskell, which is a non strict purely functional language which has been written with research, teaching and large scale applications in mind. The non strict (or lazy) evaluation strategy means that no subexpression is evaluated until it is required. This allows the definition of (conceptually) infinite data structures, but, if a computation is to be performed some finite portion of the data structure will be accessed. The pure functional programming paradigm means that programs are referentially transparent. This allows easier reasoning about the behaviour of programs since any function or expression can be replaced by any other function or expression that returns the same value. Functional programs are easier to prove correct and they are also easier to make parallel because of referential transparency.

Sherrel and Carver [Sher94] implement the class managers assistant [Word92] using direct data design, so that the data structures from the target language (in this case Haskell) represent objects in the abstract data space. The work is based upon the fact that the set and some sequence operations can be modelled as list operations in Haskell. However, the work does not consider how other operations in Z which are not suited to implementation using lists are handled. Goodman [Good95a, Good95b] creates a toolkit to model the operations in Z's mathematical toolkit for which there are no equivalent operations defined in Haskell. The toolkit is constructed using Haskell's module facility. The actual creation of the operations modelling the respective Z counterparts is more straightforward than when using Ada95 because the Haskell functions need only state 'what' the operation does and not 'how' it is done. This is because Haskell as a declarative language is much closer in spirit to the Z language. In Ada95 and C++, it is necessary to describe how each operation works by explicitly manipulating variable stores in memory.

The implementation of the more complex operations such as set comprehension, quantifiers and mu expressions are also easier. They do not have to be coded in the same way as the operations were coded as generic procedures in Ada95. In Haskell, set comprehension can be modelled as list comprehension, mu expressions can be implemented in the same way, but with the item in the singleton set being retrieved. Universal quantification is dispensed with because all functions are universally quantified over their argument types [Good95b] and the existential quantifier exists in the language.

Another area that enables Z specifications to be implemented more easily is the fact that Haskell supports higher order functions. A higher order function is one which either produces or uses other functions and as a result leads to a very rich programming style. The reusable components in Ada95 were specifically constructed in order to enable higher order functions to be implemented, (see the extended birthday book example section 5.3.1), whereas higher order functions are available in Haskell as a matter of course.

Modelling input, output and state is not as straightforward when using Haskell to implement Z. Goodman's work (using Haskell 1.2), as discussed on page 24 addresses the problem by using a Monad. In fact the use of Monads for input and output have been included in the release for Haskell version 1.3. This has a major advantage of allowing a programmer to implement input and output whilst retaining referential transparency. It is then possible to program imperative constructs in a functional manner.

One advantage that Ada has over Haskell is that Ada is a more stable language. A committee was set up to develop Ada, which was released for use in 1983. A design review was planned and a team set up to develop Ada9x, which was released for use as Ada95 in February 1995. Another review is scheduled for the next century. These design reviews have been necessary to improve the language and keep up with

developments in software engineering. However, the new versions of the language are backwardly compatible and a large body of experience exists for translation between the language versions. Once again any compiler for the new version must conform to a rigorous standard. The situation for Haskell is somewhat different. A committee was set up in 1987 and the first Haskell version was released in 1990. Version 1.2 appeared in 1992, followed by version 1.3 which added monadic I/O and scrapped the I/O of version 1.2. Here, strict backwards compatibility was abandoned in favour of the more flexible approach to I/O. The next version, 1.4, has been released recently and is generally compatible with version 1.3. However the release of version 1.4 has caused some concern, notably expressed in the Haskell Workshop of the 1997 International Conference on Functional Programming held in Amsterdam, Holland on June 9th-13th. The main concerns expressed were :-

- The language has been changing too quickly, throwing text books out of date before they are even published, and making it hard for serious users to keep up.
- The language has become more complex.
- It now contains traps for the unwary. Simple programs fail in strange and unexpected ways.

The solution, after criticism of release 1.4, was to define a simplified final version called standard Haskell and then freeze the language. Other developments will of course continue, but they will have different names in order to allow Haskell to be taught and used with confidence.

7.4.2.1 Haskell Productivity

A study carried out by the Advanced Research Project Agency and the Office of Naval Research [Huda94] compared Haskell with Ada and C++ (amongst other languages) in an experiment to compare software prototyping productivity. The following results were published which supports claims made by functional programmers that applications in functional languages take less development time and can be completed in less lines of code. Since the study was aimed at

prototyping, no comparison of the performance difference between implementations was made.

language	lines of code	lines of documentation	development time(hours)
(1) Haskell	85	465	10
(2) Ada	767	714	23
(3) Ada9X	800	200	28
(4) C++	1105	130	-
-			
-			
-			
(10) Haskell	156	112	8

Each implementation was developed by an expert programmer in the chosen language. The Ada solution was written by a lead programmer at the Naval Surface Warfare Centre (NSWC). The line count initially reported was 249, but this did not include declarations. The Ada9x version was written by an independent consultant from Intermetrics Inc. The consultant was given the Haskell solution along with the problem specification. The second implementation using Haskell (number 10 above) was programmed by a newly hired graduate given eight days to learn the language. This was done to show that the language is easy to learn. The report concludes that the experiment clearly demonstrates the value of functional programming. However, it was noted that the experiment is lacking and may be criticised in many ways as discussed in [Huda94].

7.4.3 Conclusion

It would seem that the reusable components to model Z could have been constructed with more speed and ease using Haskell. However, in the main, functional languages

as yet do not match the performance of their imperative counterparts. A case study compared 25 different implementations of functional languages executing a single program and found that 24 of them produced code that executed slower than an equivalent C program [Hart96]. This is in part due to the fact that most computers in use today follow the Von Neumann concept with a single processor connected to the machines memory via a data bus. This architecture uses destructive updating where the contents of variables change over time and suits imperative languages which are based upon destructive updating. Functional languages that create referentially transparent programs would allow different parts of a program to be evaluated in parallel on different processors. This would alleviate concerns over efficiency. In the future, functional languages may be used for more and more projects, with improvements in compilers for functional languages, improvements in computer architecture and standardisation of functional implementations. However, at present, functional languages are barely used in the creation of real software applications due to concerns over performance, different implementations of the same languages causing portability problems, and a lack of facilities to interface with other languages. Although it is more time consuming and more difficult to construct the reusable components in Ada95, the advantages that Ada as a widely used, standardised imperative language bring to the method outweigh the advantages of using a language like Haskell, which is simply not used to any extent in real applications due to the problems mentioned above.

7.5 Scaling up to Industrial Sized Problems

7.5.1 Introduction

One of the main barriers to the widespread use of formal methods is the difficulty of scaling up a method to use on an industrial sized application. Methods that define concrete specifications use proofs and have been criticised because the proofs are difficult and time consuming in all but the simplest cases. As a result very few are

completed. Refinement using functional languages provide a means of animating a specification, but, as seen in section 7.4.3, the code is not often usable in a final application.

In this section the Z specification for a steam boiler [Buss96] and a real industrial specification for an aircraft cabin illumination system [Hame95] are implemented. The aim is to shed light on whether or not using reusable components in Ada95 to translate Z specifications is capable of scaling up to an industrial sized specification.

7.5.2 The Steam Boiler

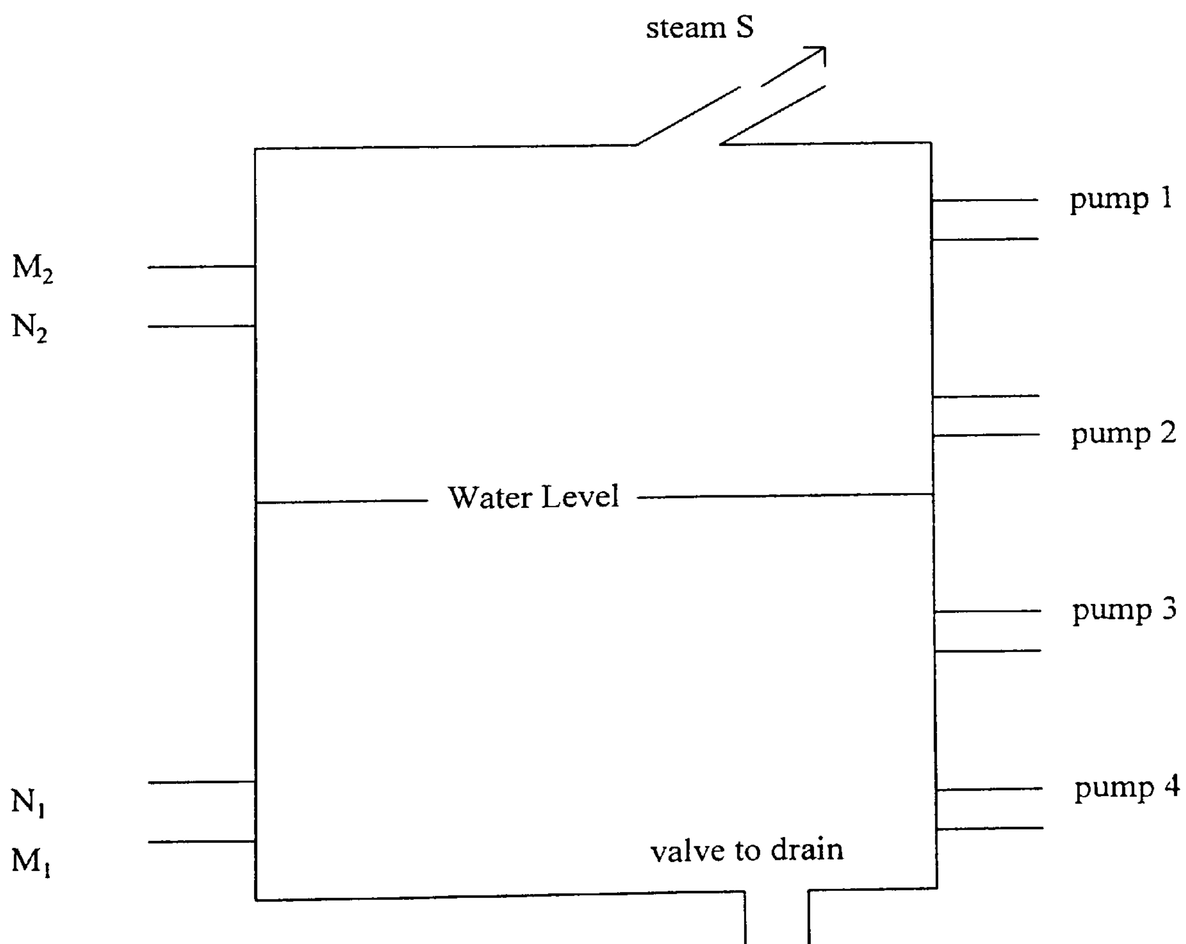
An international seminar on "methods for semantics and specification" was held at Schloβ Dagstuhl, Wadern, Germany on June 5-9, 1995, in order to aid the industrial take-up and scientific progress of formal methods. The seminar took the form of a competition to specify the behaviour of a steam boiler. It was proposed as a common case study in order to assess different formalisms and to compare the strengths and weaknesses of the various methods. In this section, a specification for the steam boiler, that was written in Z and proposed as a solution at the Dagstuhl, will be translated into Ada95 code using the reusable components.

The Z specification for the steam boiler specifies the data definitions, data invariants, the state space and the input/output relations of its operations. An implementation of this specification will result in abstract state machines (since the steam boiler specifies more than one state) that capture the specified behaviour of the steam boiler. However, in order for the steam boiler implementation to work, these abstract state machines must be used in a program that incorporates the reactive behaviour of the steam boiler with respect to timing constraints and requests for operations to be carried out. For this reason, the code for the functional model of the steam boiler has been compiled, but it has not been tested due to the lack of a reactive model and the lack of suitable software to simulate the steam boiler.

Due to the size of the steam boiler specification, only the most relevant information about the specification along with a limited selection of examples for translating schemas will be given. The solution to the steam boiler specification problem was split into three views, in order to make complex systems manageable and to detect inconsistencies at an early stage. The three modelling views of the embedded system were:-

- The Architectural Model - defining class relations and system structure.
- The Functional Model - defining data structures and I/O value transformations.
- The Reactive Model - defining object interaction and time control.

The following diagram of the steam boiler defines the variables used in the specification:-



W	- Maximal steam quantity (litres per second).
C	- Capacity of boiler.
M_1,M_2	- Extreme water levels.
N_1,N_2	- Normal water levels.
qa_1	- Lower water level approximation.
qa_2	- Higher water level approximation.
va_1	- Lower approximation of steam value.
va_2	- Higher approximation of steam value.
u_1	- Maximum gradient of steam increase (litre/second/second).
u_2	- Maximum gradient of steam decrease (litre/second/second).

The steam boiler problem consists of the steam boiler itself along with physical units such as sensors, pumps and monitoring equipment. The various units may each be in a subset of the following states :-

Unitstates ::= working | broken | closed | opening | open | flow | noflow

The states flow and noflow are not equivalent to, or included in, open and closed. Sensor units for example are in the following states:-

SensorStates = = {working, broken}

A non private package is constructed to house the definition of the various sensor states as follows:-

```
package states is
  type sensorStates   is (working,broken);
  type monitorStates  is (flow,noflow,broken);
  type pumpStates     is (closed,opening,open,broken);
  type valveStates    is (open,closed);
  --other types used in specification
end;
```

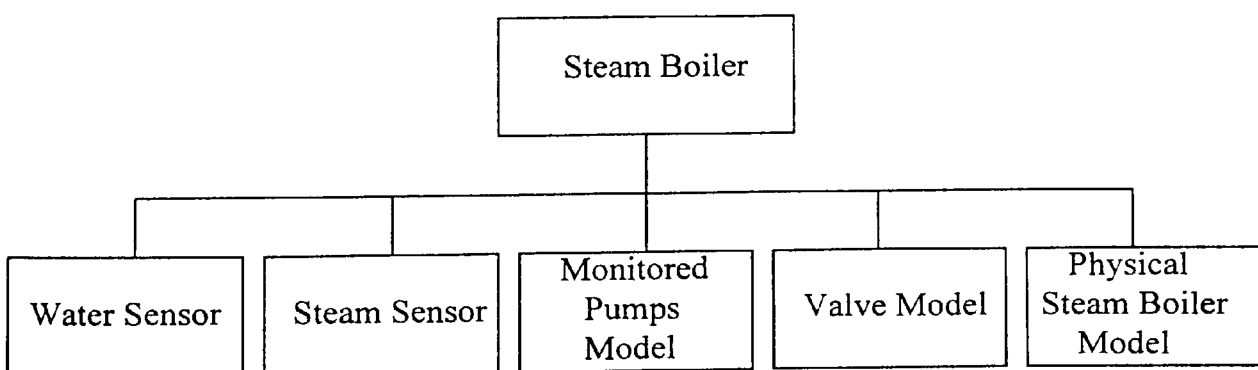
The units are

- The water sensor
- The steam sensor
- The monitored pumps model
- The valve model
- The physical steam boiler model
- The steam boiler state

Each of these units consists of a state, some definitions and in some cases operations to be performed upon the system state. In this section, a package hierarchy will be developed to model each of these physical units. Each state schema and its corresponding implementation shall be given, but, for the sake of brevity, only some of the definitions and operations for the physical units shall be given.

7.5.2.1 Architectural Model for Steam Boiler Software

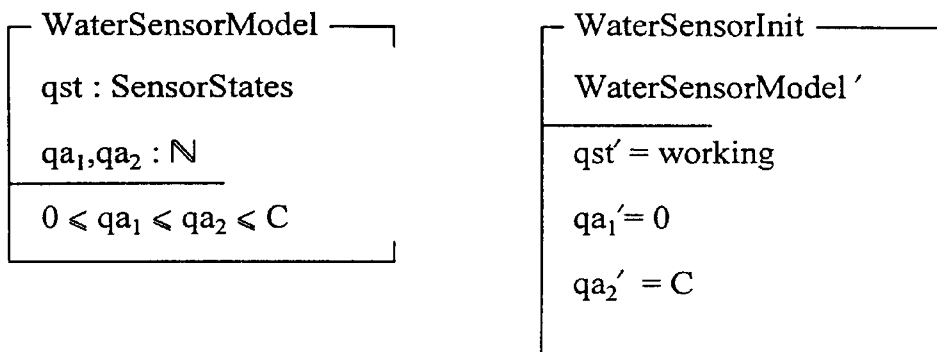
The steam boiler state will be the main package used in the system. The Water sensor, steam sensor, monitored pumps model, valve model and physical steam boiler model will be child packages of the steam boiler state as follows:-



7.5.2.2 The Water Sensor Model

This is modelled by its state and lower and upper approximations of the water level value. These approximations are used in case nothing is known about the sensor

value. During normal operation these two values are equal, representing the unique water level value.



There are no operations defined on the WaterSensorModel, but there are a number of definitions such as :-

$$WaterLow == [WaterSensorModel \mid M_1 \leqslant qa_1 < N_1 \wedge qa_2 \leqslant N_2]$$

$$WaterNormal == [WaterSensorModel \mid (N_1 \leqslant qa_1 \wedge qa_2 \leqslant N_2)]$$

These definitions are used as statements in other schemas such as :-

:

:

$WaterNormal \Rightarrow st' = ready \wedge Ps' = Ps \wedge vlv' = vlv$

For this reason, they are implemented as functions returning a boolean value. This is a very small and intuitive refinement to make. The above statement can then be implemented as :-

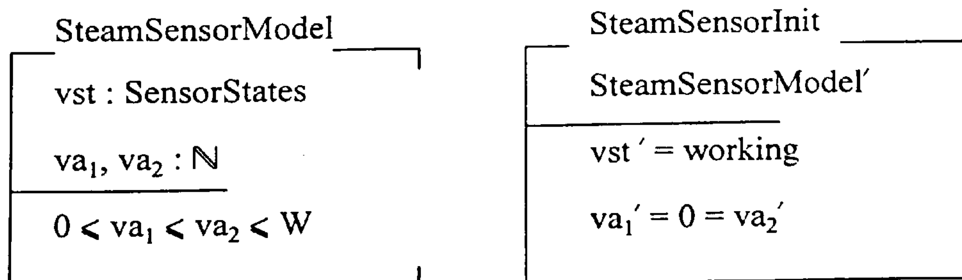
```
if WaterNormal then
   st:=ready;
   - - -
```

The private Ada child package specification for the WaterSensorModel schema is as follows:-

```
with states;
use states;

private

generic

  n1:natural;       --normal higher water level
  n2:natural;       --normal lower water level
  m1:natural;       --extreme higher water level (danger too high)
  m2:natural;       --extreme lower water level  (danger too low)


package steamb.wsenmod is

  function WaterLow              return boolean;
  function WaterHigh             return boolean;
  function WaterNormal           return boolean;
  function WaterTolerable        return boolean;
  function WaterAboveNormal      return boolean;
  function WaterBelowNormal      return boolean;
  function WaterDanger           return boolean;
  function WaterSensorWorking    return boolean;
  function WaterSensorBroken     return boolean;
  function WaterKnown            return boolean;

  procedure waterSensorInit;

  qst :sensorstates;
  qa1 :natural;             --lower water level approximation
  qa2 :natural;             --higher water level approximation

end;
```

7.5.2.3 The Steam Sensor Model

The steam sensor is modelled by its state and lower and upper approximations of the steam value.



Again, a number of definitions are given, such as :-

SteamSensorWorking = = [SteamSensorModel | vst = working]

The private Ada child package specification is given below, note that it must also be generic (even though it has no formal generic parameters) because it is the child of a generic parent.

```
with states;
use states;

private

generic

package steamb.ssenmod is

  procedure SteamSensorInit;
  function  SteamZero return boolean;
  function  SteamSensorWorking return boolean;
  function  SteamSensorBroken return boolean;

  vst : SensorStates;
  Va1 :natural;          --lower  approximation of steam value
  Va2 :natural;          --higher approximation of steam value

end;
```

7.5.2.4 The Monitored Pump Model

Each pump may be in one of the following states defined by :-

PumpStates = = {closed, opening, open, broken}

The possible state 'closing' has not been included by the specifiers. One can assume that it is was not deemed necessary to include this state. An individual pump is modelled by its state and an approximation of its capacity.

PumpModel
pst : PumpStates
$pa_1, pa_2 : \mathbb{N}$

$pa_1 \leq pa_2$
$pst \in \{closed, opening\} \Rightarrow pa_1 = 0 = pa_2$
$pst = open \Rightarrow pa_1 = P = pa_2$

The predicates above define the constraints for the upper and lower pumping capacity approximations for a non-defective pump. If the pump is opening or closed, the pumping rates are set to zero. If open, the upper and lower approximations are set to the actual pumping rate P.

Each pump is controlled by a monitor. The water can be either flowing or not flowing, or the monitor may be broken.

MonitorStates = = {flow, noflow, broken}

A monitored pump is a pump together with a monitor, whose main purpose is to define the pumps capacity approximations in case the pump is broken.

MonitoredPumpModel
PumpModel
mst : MonitorStates

$pst = broken \Rightarrow$
$\quad (mst = flow \Rightarrow pa_1 = P = pa_2) \wedge$
$\quad (mst = noflow \Rightarrow pa_1 = 0 = pa_2) \wedge$
$\quad (mst = broken \Rightarrow pa_1 = 0 \wedge pa_2 = P)$
$pst \in \{closed, opening\} \Rightarrow mst \in \{noflow, broken\}$
$pst = open \Rightarrow mst \in \{flow, broken\}$

In the cases above, the monitor can either show noflow or flow according to the pump state. However, in both cases the monitor can also be broken.

The definition of a monitored pump is lifted to a sequence of monitored pumps. The MonitoredPumpsModel schema sets the upper and lower water pumping approximations for the whole system by multiplying the approximations for individual pumps by the number of pumps in the system. It is defined as follows:-

MonitoredPumpsModel

$Ps : seq_1\ MonitoredPumpModel$

$pa_1, pa_2 : \mathbb{Z}$

$\#Ps = NP$

$pa_1 = P * \# \{ i: 1..NP \mid (Ps\ i).pa_1 = P\}$

$pa_2 = P * \# \{ i: 1..Np \mid (Ps\ i).pa_2 = P\}$

A number of conditions exist for monitored pumps, such as :-

$PumpBroken == [MonitoredPumpsModel; i? : 1..NP \mid (Ps\ i?).pst = broken]$

$PumpsWorking == [MonitoredPumpsModel \mid \forall i : 1..NP \cdot (Ps\ i).pst \neq broken]$

The three Z schemas PumpModel, MonitoredPumpModel and MonitoredPumpsModel could have been implemented as a parent, child and grandchild hierarchy, which as a unit could be a descendant of the steam boiler package specification. However, due to the simplicity of each specification, they were implemented in a single Ada package specification as follows :-

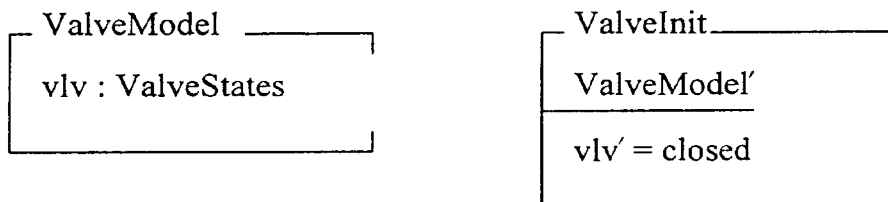
```
with  states,sequence_bounded_G,set_bounded_G;
use states;

private

generic

package steamb.mpsmod is

  procedure monitored_pumps_init;

  function PumpBroken  (i:positive) return boolean;
  function PumpWorking (i:positive) return boolean;
  function PumpsWorking return boolean;

  procedure PumpsClosed;  --the specifiers have not included
  procedure PumpsOpening; --schemas for opening and closing of
  procedure PumpsOpen;    --individual pumps

  function PumpControlBroken  (i:positive) return boolean;
  function PumpControlWorking (i:positive) return boolean;
  function PumpControlsWorking return boolean;

  type monitored_pump_model is record
     pst:pumpStates;
     pa1:natural;
     pa2:natural;
     mst:monitorStates;
  end record;

  package set_of_monitored_pumps is new set_bounded_G
                                     (monitored_pump_model,4);

  package sequence_of_monitored_pumps is new sequence_bounded_G
                                     (4,set_of_monitored_pumps);

   use set_of_monitored_pumps,sequence_of_monitored_pumps;

   pa1: integer;
   pa2: integer;
   ps : sequence_of_monitored_pumps.sequence;

end;
```

These are once again coded as functions in the package implementing MonitoredPumpsModel. As an example, the code implementing PumpsWorking is given as:-

```
function PumpsWorking return boolean is

     function check_pump_working(p:monitored_pump_model) return
                                                          boolean is
     begin
       return p.pst/=broken;
     end;

     function check_all_pumps_working is new
                      sequence_of_monitored_pumps.
                         for_all(check_pump_working);

 begin
   return check_all_pumps_working(ps);
 end;
```

7.5.2.5 The Valve Model

The state and intial state schemas for this model are very simple.



The following definitions are given:-

ValveOpen = = [ValveModel | vlv = open]

ValveClosed = = [ValveModel | vlv = closed]

The Ada package specification implementing ValveModel is given as:-

```
with states;
use  states;

private

generic

package steamb.valmod is

  procedure ValveInit;
  function  ValveOpen return boolean;
  function  ValveClosed return boolean;

  vlv : ValveStates;

end;
```

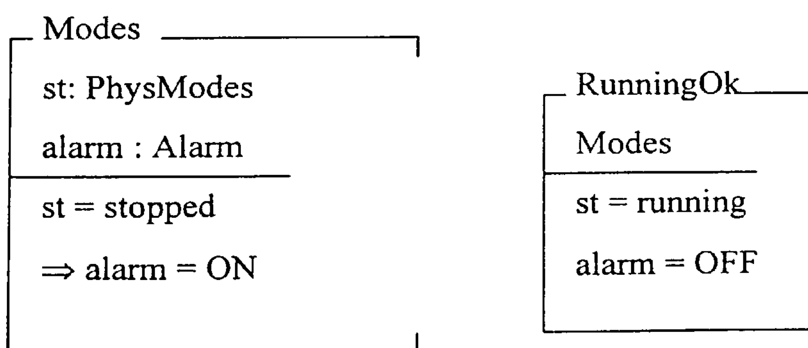
An external piece of software is responsible for controlling the valves. The Z specification does not contain this information and this refinement reflects that.

7.5.2.6 The Physical Steam Boiler

The general modes of the physical steam boiler, based upon all the component models, can be defined as :-

PhysModes ::= waiting | adjusting | ready | running | stopped

alarm ::= ON | OFF



The private Ada child specification for the physical steam boiler is given as :-

```
with states;
use states;

private

generic

package steamb.modes is

  procedure modesInit;
  function runningOk return boolean;

  st  : PhysModes;
  alm : Alarm;

end;
```

Initially, the physical steam boiler is waiting and the alarm is off

```
┌─ ModesInit ──────────────────┐
│ Modes'
├──────────────
│ st' = waiting ∧ alarm' = OFF
└──────────────────────────────┘
```

7.5.2.7 The Steam Boiler State

The state schema of the steam boiler is given as :-

```
┌─ Steam Boiler ─────────────────────────────────┐
│ WaterSensorModel
│ SteamSensorModel
│ MonitoredPumpsModel
│ ValveModel
│ Modes
├──────────────────
│ st ∈ {waiting, adjusting, ready} ⇒
│         (alarm = OFF ⇔ NoDefects ∧ SteamZero)
│ st = running ⇒
│         (alarm = OFF ⇔ WaterTolerable ∧ TolerableDefects)
│ (st = running ∨ PumpsOpen ) ⇒ ValveClosed
└────────────────────────────────────────────────┘
```

The steam boiler is initialised by initialising all its parts.

$$SteamBoilerInit == WaterSensorInit \wedge SteamSensorInit \wedge MonitoredPumpsInit \wedge ValveInit \wedge ModesInit$$

The Ada package specification implementing the state schema, SteamBoiler, is given below.

```
with states;
use  states;

generic
  c  : natural;        --capacity of steam boiler (litres).
  u1 : natural;        --upper and lower steam gradient levels
  u2 : natural;        --in litres/sec/sec.
  T  : natural;        --sampled time interval sec.
  w  : natural;        --maximal steam quantity (litre/sec).

package steamB is
  --the following procedures are all state changing. The states
  --are given in the Ada package body. See paragraph below.

  procedure level(q : natural);
  procedure steam(v : integer);
  procedure pump_control_state(i: natural;st:  MonitorStates);
  procedure pump_state(i: natural; st: pumpStates) ;
  procedure lowmark_reached;
  procedure highmark_reached;
  procedure pressure_balanced;
  procedure level_repaired;
  procedure steam_repaired;
  procedure pump_control_repaired(i:natural);
  procedure pump_repaired(i:natural);
  procedure transmission_error;
  procedure stop;
  procedure waiting_timeout;
  procedure ready_timeout;
end;
```

The steam boiler package depends upon its children. However, in Ada95 the Ada specification of a parent package cannot depend upon the Ada specification of a child package because of circular unit dependancy. As a result, the Ada package body of the steam boiler must 'with' the children and instantiate them for its use. In most cases, the children do not have generic formal parameters. However, the numbers given in the instantiation of the water sensor package (wsenmod), which follows, would depend upon properties not defined in the specification. These would have to be supplied by a person with knowledge of the specific values for these

parameters. In this example, these numbers have been picked arbitrarily. The true values should appear amongst the systems non functional description.

A section of the Ada package body for steam boiler is as follows :-

```
with steamb.wsenmod,steamb.mpsmod,
     steamb.valmod ,steamb.modes,
     steamb.ssenmod,states;

use states;

 package body steamb is

  package WaterSensorModel is new steamb.wsenmod(4,10,20,5,15);
  use WaterSensorModel;

  package new_mpsmod is new steamb.mpsmod;
  use new_mpsmod;

  package new_valmod is new steamb.valmod;
  use new_valmod;

  package new_modes is new steamb.modes;
  use new_modes;

  package new_ssenmod is new steamb.ssenmod;
  use new_ssenmod;


  procedure steamBoilerInit is
  begin
    WaterSensorInit;
    SteamSensorInit;
    Monitored_Pumps_Init;
    valveInit;
    modesInit;
  end;

  -- other steam boiler operations
```

7.5.2.8 Steam Boiler operations

Data is transmitted in four steps, the water level data, the steam data, the pump control data and the pump data. A device failure is recognised by checking whether the fresh sensor level values remain inside the anticipated lower and upper approximations.

CalculatedLevelBounds

SteamBoiler

$qc_1, qc_2 : \mathbb{N}$

$qc_1 = \textbf{if } vlv = open \textbf{ then } 0$

$\qquad \textbf{else } \max\{0, qa_1 - (va_2 + U_1 \text{ div } 2 * T) * T + pa_1\}$

$qc_2 = \min\{C, qa_2 - (va_1 - U_2 \text{ div } 2 * T) * T + pa_2\}$

In the above schema, if the valve is open, the level may drop at an unknown rate so the lower approximation is set to zero. The equations give the upper and lower water approximation based upon the initial water approximation, the level at which steam is created or exhausted and the rate at which water is pumped in.

The Level transmission compares the fresh sensor value with the anticipated bounds and decides whether the sensor's state is considered defective and it updates the sensor value approximations.

LEVEL

$\Delta SteamBoiler$

Ξ WaterSensorModel; Ξ MonitoredPumpsModel; Ξ ValveModel

$q? : \mathbb{N}$

$alarm = OFF; st' = st$

$\exists\, qc_1, qc_2 : \mathbb{N} \mid CalculatedLevelBounds$

$\quad \bullet\ qst' = \textbf{if } qc_1 \leqslant q? \leqslant qc_2 \textbf{ then } qst \textbf{ else } broken$

$\qquad \wedge (qa_1', qa_2') = \textbf{if } qst' = working \textbf{ then } (q?, q?) \textbf{ else } (qc_1, qc_2)$

The code implementing CalculatedLevelBounds is given as a local procedure to LEVEL since it is only used in LEVEL. The schema LEVEL uses two values calculated in CalculatedLevelBounds for the water level.

```
procedure Level(q: natural) is

     procedure Calculated_Level_Bounds(qc1,qc2 : out natural) is
     begin
        if vlv = open then
           qc1:=0;
        else
           qc1 = max (0, qa1 - (va2 + (u1/2) * T) * T + pa1);
        end if;
        qc2 = min (C, qa2 - (va1 - (u2/2) * T) * T + pa2);
     end;


  begin
    if alm = off then
      Calculated_Level_Bounds(qc1,qc2);
      if qc1 <= q and q <=qc2 then
        qst:=qst;
      else
        qst:=broken;
      end if;
      if qst = working then
        qa1:=q;
        qa2:=q;
      else
        qa1:=qc1;
        qa2:=qc2;
      end if;
    end if;
  end;
```

The transmission of the steam value is similar to that of the water level, but the calculations of the steam bounds are different. The more interesting data transmission services are those that update the pump monitor states. The pump control states are updated by the following schema.

```
┌─ PUMP_CONTROL_STATE ───────────────────────────
│ ΔSteamBoiler
│ Ξ WaterSensorModel; Ξ SteamSensorModel; Ξ ValveModel
│ i? : 1..NP; st? : {flow,noflow}
├──────────────────────────
│ alarm = OFF; st′ = st
│ ∀ j : 1..NP • (Ps′ j).pst = (Ps j).pst
│ ∀ j : 1..NP | j ≠ i? • (Ps′ j ).mst = (Ps j).mst
│ (Ps′ i? ).mst = if (Ps i?).mst = broken
│                      ∨ ( (Ps i? ).pst = closed ∧ st? = flow)
│                      ∨ ( (Ps i? ).pst = open ∧ st? = noflow)
│                 then broken else st?
└──────────────────────────────────────────────
```

The first universal quantifier specifies that for each monitored pump, in the sequence of monitored pumps (Ps), the state of each pump does not change. This does not have to be explicitly translated because the code implementing PUMP_CONTROL_STATE will not change the pump states. The second universal quantifier states that the monitored pump states may only change for the value, i?, that is input. Once again, it is not necessary to explicitly implement this quantifier because the code implementing the schema will only change the monitored state for the pump in question. The code is as follows:-

```
procedure pump_control_state(i: natural;st: MonitorStates) is

   procedure update_monitor_sensor(p:in out monitored_pump_model) is
   begin
     if p.mst = broken or(p.pst = closed and st=flow)or(p.pst = open
        and st=noflow)
     then
        p.mst:=broken;
     else
        p.mst:=st;
     end if;
   end;


    procedure update_pump_control is new
              sequence_of_monitored_pumps.
                  update_an_item(update_monitor_sensor);

  begin
    if alm = off   --alm refines alarm
    then
       update_pump_control(i,ps);
    end if;
  end;
```

7.5.2.9 Conclusions

The functional specification for the steam boiler problem was implemented in a quick and straightforward manner. The following conclusions can be drawn about the manner in which the specification for the steam boiler was written.

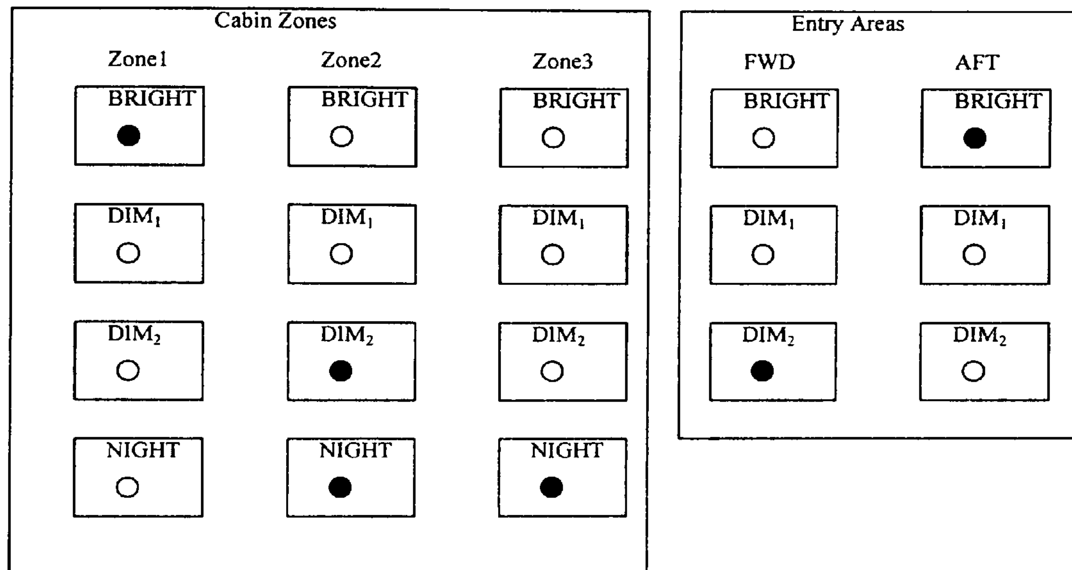
- It separated concerns between the reactive model of the steam boiler and the functional model for the steam boiler. Indeed, it would have been difficult to specify the reactive model for the steam boiler in Z because many models for adding real time to Z have been proposed, but no method has been agreed upon and no method has been shown to be superior [Fidg92].
- Although there is no module construct in Z, the specification was written in a 'modular' style. Each state and its corresponding operation schemas modelled one aspect of the steam boiler functionality. The steam boiler could then use each of these modules. As a result developing the Ada package hierarchy was reasonably straightforward.
- The authors purposefully wrote the specification in a constructive style to aid in its implementation. This enabled the specification to be translated as it was and without carrying out any other specification work in Z.

This case study shows that specifications, when written in a modular and constructive style, can be translated into Ada code using reusable components with considerable speed and ease. One barrier to using reusable components to translate Z specifications is not necessarily the complexity of operation schemas or the number of operations in a specification, but the relationships between the operations and the various states. In general a Z specification will not map to an Ada implementation in a natural manner [Read92]. A Z specification that is written in a modular style will be more easily implemented as each of the modularised states and their respective operations can be housed in Ada packages in a straightforward manner. The next example is a real world specification for an aircraft illumination application which has not been written in a modular style. As a result, the interaction between the Ada packages implementing the specification is more complex.

7.5.3 The Airbus A330/340 CIDS cabin Illumination specification

This section presents a Z specification for the Cabin Illumination function (CIL) which is a sub application of the Airbus A330/340 Cabin Intercommunication Data System (CIDS). The informal requirements and the Z specification for the CIL

application are given in [Hame95]. Once again, for reasons of economy, only some of the details of the specification will be presented here. The CIL application allows for separate control of the illumination of the aircraft cabin zones (e.g. first class, business class etc.) and entry areas (forward and aft). To change the illumination status, the cabin staff uses a panel as shown in the following diagram:-



The intensities for the illumination levels are bright, dim_1 and dim_2. The commands for the cabin zones enable each zone to be illuminated separately according to the required illumination level. For example, if the illumination in zone 1 is off, and the user presses dim_2, the illumination units will be set to this level throughout the zone. This is acknowledged by the indicator unit for button dim_2 lighting up in zone 1. The illumination for entry areas can be controlled in a similar manner. However, in addition to the command from the user, certain sensors can control the illumination levels. For example if the cockpit sensor signals "door open", the illumination level of the forward entry area is lowered to avoid distracting the cockpit crew.

7.5.3.1 Basic Data Types and Sets

[ADDRESS] -- unique identifier for an illumination unit.

Location ::= z_1 | z_2 | z_3 | fwd | aft --z1, z2, z3 are cabin zones

-- fwd, aft are cabin entry areas

Zones = = $\{z_1, z_2, z_3\}$

EA = ={fwd, aft} --entry areas

Dim ::= dim_1 | dim_2 | bright | off | $onNL_2$ --illumination intensities

Dim_0 = = {dim_1, dim_2, bright, off}

Dim_1 = = (dim_1, dim_2, bright}

SWITCH ::= passive | active

LGCIU ::= downcompressed | downlocked | uplocked --Landing Gear Control &

--Interface Unit

PRESSURE ::= low | high

DOOR ::= closed | open

FEATURE ::= disabled | enabled

An ordering relation for illumination levels is defined as:-

$$-<_{dim}- : DIM_0 \leftrightarrow DIM_0$$

$$off <_{dim} dim_2$$
$$dim_2 <_{dim} dim_1$$
$$dim_1 <_{dim} bright$$

$$\forall a,b,c : DIM_0 \mid a <_{dim} b \wedge b <_{dim} c \bullet a <_{dim} c$$

7.5.3.2 Definition of Cabin Assignment Module (CAM) tables.

The Cabin Assignment Module (CAM) tables, which follow, can be viewed as global constants that are accessed anywhere in the specification. The set of ADDRESS is the set of all DEU-A (Decode/Encoder Unit A) addresses combined

with their ports to which any illumination unit is attached. The following axiomatic descriptions are given and are used through the CIL Z specification.

$$
\begin{array}{|l}
CAM_CAB : ADDRESS \twoheadrightarrow ZONES \\
CAM_EA : ADDRESS \twoheadrightarrow EA \\
CAM_NL1 : ADDRESS \twoheadrightarrow ZONES \\
CAM_NL2 : ADDRESS \twoheadrightarrow ZONES \\
CAM_FAP : PLOCATION \\
\hline
CAM_NL1 \subseteq CAM_CAB \\
\mathrm{dom}\ CAM_CAB \cap (\mathrm{dom}\ CAM_EA \cup \mathrm{dom}\ CAM_NL2) = \varnothing \\
\mathrm{dom}\ CAM_EA \cap \mathrm{dom}\ CAM_NL2 = \varnothing
\end{array}
$$

$$
\begin{array}{|l}
CAM_EADIM : ADDRESS \twoheadrightarrow \{off, dim1, dim2\} \\
\hline
\mathrm{dom}\ CAM_EADIM \subseteq \mathrm{dom}\ (CAM_EA \rhd \{fwd\})
\end{array}
$$

$$
\begin{array}{|l}
CAM_DECOMP : FEATURE \\
CAM_BLOCK : FEATURE \\
CAM_NLAUTO : FEATURE
\end{array}
$$

The Cam tables and the basic data types and sets have been implemented in an Ada package, which must be visible to the other Ada packages implementing the state schemas and operation schemas.

```
with Set_bounded_G,Many_to_one_G;

package definitions is

   type location is          (z1,z2,z3,fwd,aft);
   type the_zones is         (z1,z2,z3);
   type the_ea is            (fwd,aft);
   type feature is           (enabled,disabled);

   --other types

   function "<" (d1,d2 : the_dimo) return the_dimo;

   package zone_set is new Set_bounded_G(the_zones,3);
   use zone_set;


   package ea_set is new Set_bounded_G(the_ea,2);
   use ea_set;
```

```
   --- other set package instantitions


   package CAM_CAB_map is new Many_to_one_G
                                    (18,address_set,zone_set);


   package CAM_EA_map is new Many_to_one_G(18,address_set, ea_set);


   --- other CAM table instantiations


    zones    : zone_set.set;
    ea       : ea_set.set;

   --- other set declarations

   CAM_CAB           : CAM_CAB_map.map;
   CAM_EA            : CAM_EA_map.map;

   --- other CAM declarations

   CAM_NLAUTO,CAM_NL1_DIM :feature

   -- others of type feature

end;
```

7.5.3.3 The System State Schemas

The following schema models the state of zone button indicators:-

ZONEINDstate

$zoneInd : ZONES \rightarrow DIM_0$

$nlInd : ZONES \rightarrow SWITCH$

$\forall z : ZONES \bullet$

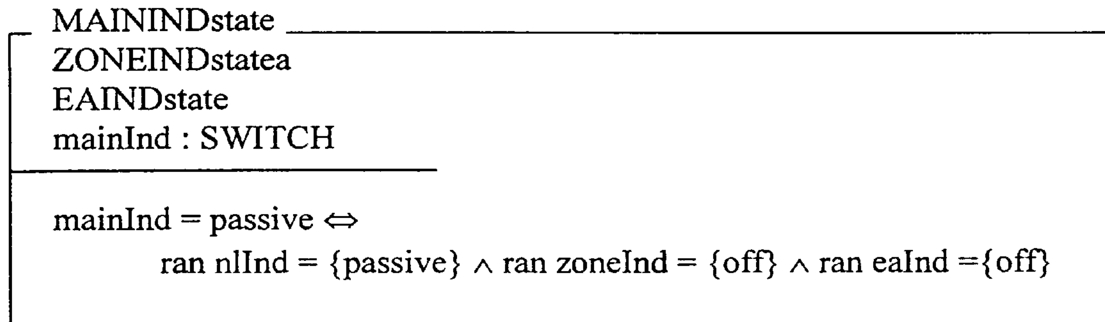
$\quad nlInd(z) = active \Rightarrow (zoneInd(z) = off \vee CAM_NLAUTO = disabled)$

The next schema models the state of entry area button indicators

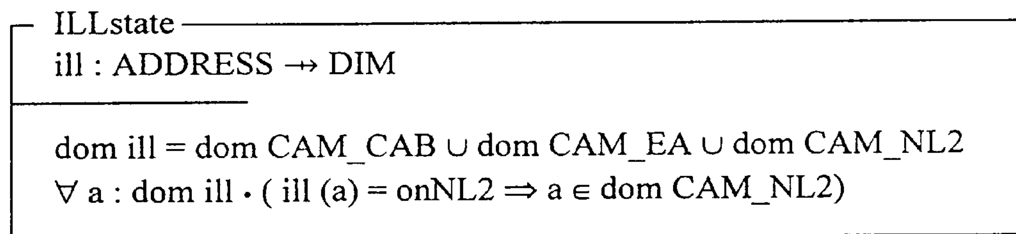
EAINDstate

$eaInd : EA \rightarrow DIM_0$

Schema MAININDstate models the state of the MAIN button indicator.

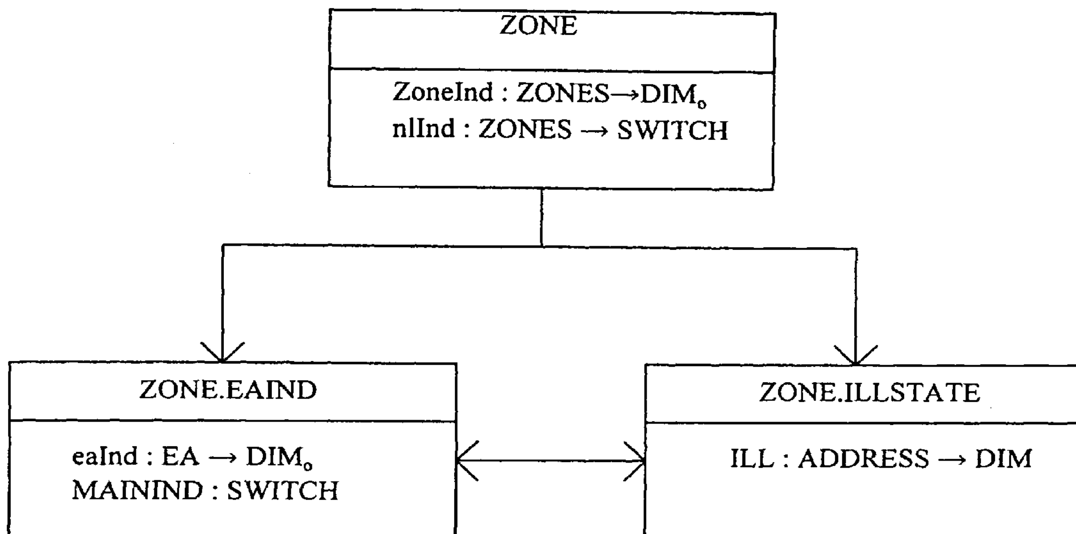


The collection of all illumination units and their corresponding illumination levels is modelled as a mapping between the set ADDRESS and the set DIM, as follows:-



Since the cabin illumination specification is not written in a modular style the operation schemas must be examined to identify which operations must be placed in which package, and how these packages are to be related. Many of the operations throughout the specification make use of more than one state. Operations are also made total by schema calculus, where each of the suboperations making up the total operation also operate over different states. An example, and its implementation is given in section 7.5.3.5.

The following package hierarchy was used to implement the system:-



In the above hierarchy, the packages ZONE.EAIND and ZONE.ILLSTATE are visible to the package ZONE, as they are its children. However, the Ada package body ZONE also depends upon the child packages ZONE.EAIND and ZONE.ILLSTATE for some of its operations. In the Ada package body ZONE, it is therefore necessary to have a `with` clause for the two children. The children must also have visibility of each other, because for example, an operation in ZONE.EAIND changes the state of ZONE.ILLSTATE. The package hierarchy had to be implemented in this way because many operations defined in each package used more than one state. The Z specification was constructed in such a manner that a mapping between the Z specification structure and the Ada implementation was not straightforward.

7.5.3.4 The Ada Package Specifications Implementing the State Schemas

The Ada package specification for ZONE is :-

```
with set_bounded_G,many_to_one_G,definitions;
use definitions;

 package ZONE is

  procedure zoneop(z:in the_zones; d: in the_dimo);

  -----

private
    package zoneInd_map is new many_to_one_G(3,zone_set,dimo_set);
    use zoneInd_map;                --the three zones are z1,z2,z3

    zoneInd:zoneInd_map.map;

    package nlInd_map is new many_to_one_G(3,zone_set,switch_set);
    use nlInd_map;

    nlInd : nlInd_map.map;
end;
```

The private Ada child package specification for ZONE.EAIND is :-

```
with definitions,many_to_one_G;
use definitions;

private

package ZONE.EAIND is

  procedure EAop(ea:the_ea; dim:the_dimo);
  -----


    package EAInd_map is new many_to_one_G(2,ea_set,dimo_set);
    use EAInd_map;                  --there are two entry areas
    EAInd : eaind_map.map;
end;
```

The child package ZONE.ILLSTATE, which follows, must have visiblity of its sibling ZONE.EAIND. The package body of ZONE.ILLSTATE contains the 'with' clause for ZONE.EAIND.

```
with definitions, many_to_one_G;
use  definitions;

private

package ZONE.ILLSTATE is

   procedure NLop (n: in the_zones);
   -----

   package ill_map is new many_to_one_G(18,address_set,dim_set);
   use ill_map;

   ill : ill_map.map;
   mainInd:switch;

end;
```

7.5.3.5 An Operation Schema Example - The Night Light Operation

If the night light button indicator of zone n? is passive, it will be turned on if the night light autofunction is disabled or the zone illumination has previously been turned off. If the night light autofunction is enabled and the illumination is active, pressing a night light button will have no effect, but when switching off the zone illumination the night lights will automatically be turned on. The following schemas describe the operation of the night light buttons:-

```
┌─ NLINDop ──────────────────────────────────
│ Δ ZONEINDstate
│ Ξ EAINDstate
│ n? : ZONES
├──────────────
│ nlInd (n?) = passive ⇒ nlInd ′ =
│         if CAM_NLAUTO = disabled ∨ zoneInd(n?) = off
│         then nlInd ⊕ {n? ↦ active } else nlInd
│ nlInd (n?) = active ⇒ nlInd ′ = nlInd ⊕ {n? ↦ passive}
│ zoneInd ′ = zoneInd
└────────────────────────────────────────────
```

NLILop
ZONEINDstate
Δ ILLstate
n? : ZONES

---

nlInd (n?) = passive $\Rightarrow$ ill $'$ =
  if zoneInd (n?) = off
  then ill $\oplus$ {x:dom (CAM_NL2 $\rhd$ {n?}) • x $\mapsto$ onNL2}
    $\oplus$ {x:dom (CAM_NL1 $\rhd$ {n?}) • x $\mapsto$ CAM_NL1_DIM }
  else ill
nlInd (n?) = active $\Rightarrow$ill $'$
  if zoneInd(n?) = off
  then ill $\oplus$ {x:dom (CAMNL2 $\rhd$ {n?}) • x $\mapsto$ off}
    $\oplus$ {x:dom (CAMNL1 $\rhd$ {n?}) • x $\mapsto$ off}
  else ill

The total operation for the night light is :-

NLop = NLILLop $\wedge$ NLINDop

7.5.3.6 Implementation of Night Light Operation

The schema NLop was expanded in full. In NLILLop there are four functional override statements that override the ILL state according to a set comprehension term that creates a set of address and intensity pairs. It is necessary to create four generic packages instances, one for each of these statements. The procedures used to instantiate the iterator procedure, such as `set_CAM_NL2_to_onNl2`, set each address to the desired illumination intensity. The resulting functions are then used to override the ill state accordingly.

```
procedure NLop (n: the_zones) is

   intensity : the_dimo;
   switch_is : switch;

   temp_CAM_NL2: CAM_NL2_map.map;    -- temporary variables to store
   temp_CAM_NL1: CAM_NL1_map.map;    -- results of range restrict

 temp_ILL_onNl2   :ZONE.ILLstate.Ill_map.map;--temporary ILL mappings
 temp_ILL_NL1_DIM:ZONE.ILLstate.Ill_map.map;--used in the functional
 temp_ILL_NL1     :ZONE.ILLstate.Ill_map.map;--override statements
```

```
temp_ILL_NL2     :ZONE.ILLstate.Ill_map.map;--of NLop

-- the following procedures are used to set the temporary ILL
-- mappings to the correct illumination levels. Each procedure is
-- then used to instantiate the iterator to create a complete
-- temporary map which is then used in the functional override
-- statements of NLop.

  procedure set_CAM_NL2_to_onNl2(x:positive;z: the_zones;
                                 continue:out boolean) is
  begin
    bind(x,onNL2,temp_ill_onNl2);
    continue:=true;
  end;

  procedure set_CAM_NL1_to_CAM_NL1_DIM(x:positive;z: the_zones;
                                       continue:out boolean) is
  begin
    bind(x,CAM_NL1_DIM,temp_ILL_NL1_dim);
    continue:=true;
  end;

  procedure set_CAM_NL2_to_off(x:positive;z: the_zones;
                               continue:out boolean) is
  begin
    bind(x,off,temp_ILL_NL2);
    continue:=true;
  end;

  procedure set_CAM_NL1_to_off(x:positive;z: the_zones;
                               continue:out boolean) is
  begin
    bind(x,off,temp_ILL_NL1);
    continue:=true;
  end;


 -- Four iterator procedures are used because there are four
 -- statements in NLop of the form
 --    ⊕{ x: dom ( CAM_NL1 ▷ {n?} ) • x ↦ onNL2}.
 -- The iterator creates the temporary set of pairs  used in the
 -- override operation. The iterators are instantiated next.

 procedure set_CAM_NL2 is new defin.CAM_NL2_map.
                          iterate(set_CAM_NL2_to_onNl2);

 procedure set_CAM_NL1 is new defin.CAM_NL1_map.
                          iterate(set_CAM_NL1_to_CAM_NL1_DIM);

 procedure set_CAM_NL2_off is new defin.CAM_NL2_map.
                          iterate(set_CAM_NL2_to_off);

 procedure set_CAM_NL1_off is new defin.CAM_NL1_map.
                          iterate(set_CAM_NL1_to_off);


 ---------------------------------------------------------------
```

```
 begin -- main body of  NLop operation

      create_map(temp_ILL_onNl2);
      create_map(temp_ILL_NL1_DIM);
      create_map(temp_ILL_NL1);
      create_map(temp_ILL_NL2);

      defin.CAM_NL2_map.range_restrict  --store results of range
            (n,CAM_NL2,temp_CAM_NL2);   --restrict in temporary map
      defin.CAM_NL1_map.range_restrict
            (n,CAM_NL1,temp_CAM_NL1);

      range_of (n,switch_is,nlInd);

      if switch_is = passive     -- ( nlInd (n?)=passive )
      then
         ZONE.zoneInd_map.range_of (n,intensity,zoneInd);
         if CAM_NLAUTO = disabled or intensity=off
         then
            function_override(n,active,nlInd);
         end if;
         if intensity =off
         then
            set_CAM_NL2(temp_CAM_NL2);    --create temporary ILL maps
            set_CAM_NL1(temp_CAM_NL1);
            function_override(ill,temp_ILL_onNL2 ); -- override ill
            function_override(ill,temp_ILL_NL1_DIM); --state
         end if;
      else  -- ( nlInd(n?) = active )

         function_override(n,passive,nlInd);
         if intensity=off
         then
            set_CAM_NL2_off(temp_CAM_NL2);   --create temporary Ill
            set_CAM_NL1_off(temp_CAM_NL1);   --maps
            function_override(ill,temp_ILL_NL2);  --override ill
            function_override(ill,temp_ILL_NL1);  --state
         end if;
      end if;
   end;
end;
```

The other operations in the aircraft cabin illumination specification are very similar in their construction to that of the night light operation, and can be implemented in the same manner.

7.5.3.7 Conclusions

The following conclusions can be drawn from the implementation of the Cabin Illumination application specification.

- The specification was constructive which enabled the specification to be implemented as it was, with much speed.
- The specification was not written in a modular style. The structure of the Z specification did not map to an implementation is a straightforward manner, although using child packages enabled each operation to have the desired visibility of the correct state schemas.
- The implementation of the specification cannot be used as the final system, because at the level of abstraction for this specification, Hamer and Peleska state that the difference between Flight Attendant Panels (FAP), the Multi Purpose Bus (MPB) and the Additional Attendant Panels (AAP) was irrelevant, since the effect of pressing a button only depends upon the button's function, not on the panel where the button was pressed. This statement leads one to believe that a more concrete specification would take this difference into account and therefore make it relevant. This shows that, since the translation of the specification into Ada code is direct, the specification must be written in such a way as to fully account for the functionality of the system.
- If the distinction between the FAP, MPB and AAP was made relevant in the specification, then the operations implemented in this section, could be used in the final implementation if the performance is satisfactory. Although, the operations may have been programmed by hand in a more efficient manner, due to rationalising code, the small size of the functions means that the performance may be satisfactory. This can be evaluated by checking the operation of the system against the non functional requirements (for example the time delay between pressing the night light illumination button on the control panel and the night lights actually being illuminated).
- The system was a real time system, but these aspects were contained in an operating system layer which was outside the scope of the specification. This

operating system layer is responsible for calling the operations implemented from the CIL Z specification.

7.6 Summary of Conclusions

A Z specification is not concerned with design details, but, when translation is direct, the Z specification is forced to be viewed as a design document. Hence, the performance of the Ada code depends on how well, in terms of efficiency, the Z specification is written. In order to translate the Z specification into Ada code using the reusable components, it is necessary to have a constructive specification. This is true of all methods in the literature that directly translate a specification. Other methods that refine the specification into a more concrete version based upon structures found in a programming language, must also add detail in order to arrive at executable code. It is widely recognised that a hierarchy of specifications is required from an initial abstract specification to one that contains successively more implementation based decisions. In [Grav91], the limits of executing a specification presented in an implicit (or non-construction) manner are pointed out. Gravel and Henderson [Grav91] show methods to improve the executability of specifications and they highlight the advantages of having an implicit specification to capture invariant properties of the system and a constructive one to explain the operation of the system. This work is of value with regards to the method described in this thesis as refinement using resusable software components requires a constructive specification.

The performance of the operations from the reusable components such as inserting, deleting and testing for membership can be constructed to achieve the same performance as equivalent manually constructed code using the same data structure. The difference in efficiency between the code produced manually and the code from the reusable components mostly results from the way in which the Z specification has been constructed. The direct implementation is forced to use the data model of the original abstract state schema, which may well be changed when moving to a more concrete design, as the library example of section 7.3 shows. Obviously,

operations on the new concrete state schema will be different from the abstract versions and will take advantage of a more efficiently represented state space. If the performance of the code obtained from implementing the original specification is not efficient enough, and cannot be tuned sufficiently, then it must classed as a prototype, since it cannot be used as the final implementation. However, as a prototype, it still has the advantage of enabling the customer and the developer to have confidence that the systems requirements have been met. A concrete version utilising a more efficient state model can be constructed, which can also be translated using the reusable components, because the concrete design is still written using Z

In order for this method to scale up to industrial sized specifications, a shift in the way Z specifications are written is required. At present, the method will not scale up to an industrial sized specification easily because specifications are not being written in a modular style. However, it has been stated [McDe89 foreword] that one of the main technical barriers to the widespread use of formal methods in industry is the lack of a adequate techniques for modularising specifications. As a result, methods have been proposed to aid in the reuse of Z specifications [Lano92] and to add modularity to Z [Mitc94]. Read [Read92] has taken a different approach and has proposed a method of writing Z specifications that model the way in which Ada packages are constructed. This enables a Z specification to map to an equivalent Ada package hierarchy and facilitates reuse of both the Ada packages and the Z specification. Writing modular specifications will not only benefit this method, since a concrete design derived from an original and unmodularised specification may also be difficult to implement. The same decisions as to which concrete states and operations will be modelled in which package will still exist if the concrete specification is not written in a modular style.

All three languages that have been compared are Turing complete and hence, each language can be used as the target language. The main disadvantage in C++ is that its generic facilities are not as powerful those found in Ada95. The code may be

All three languages that have been compared are Turing complete and hence, each language can be used as the target language. The main disadvantage in C++ is that its generic facilities are not as powerful those found in Ada95. The code may be developed in Haskell with more ease, but performance, portability, interfacing and its lack of use in industry are major hurdles.

There is a two edged sword between using Z in a very abstract way for specification and the work required to construct an implementation from a specification written in this manner. A very abstract specification will result in much work being carried out in order to arrive at an implementation. A more concrete and constructive Z specification will reduce a specifiers freedom and will introduce design decisions into the specfication document. However, this extra work in the Z domain results in an easier path to executable code.

Chapter 8

Advantages of Ada95 Over Ada83

8.1 Introduction

Initially, Ada83 was used as the target language, but, with the advent of Ada95, the reusable components were converted to use the new version of Ada. This chapter discusses the advantages that Ada95 has over Ada83 with regards to the refinement of Z specifications using reusable components.

8.2 Improvements in the Generic Paradigm

The basic type in Z is the set. Other types such as functions, relations and sequences will depend on the type set when they are implemented as reusable components. This is because some operations, such domain or range restrict can require a set of elements as a parameter to the procedure. In these cases a set and the necessary operations to create a set, insert an item and so forth, must be visible and compatible to the package containing the operation. In Ada83, this can be achieved by importing the set and the necessary operations through the generic part of the client package. The following code segment, from a one to one package specification, highlights the number of generic parameters and formal subprograms required to construct a function package whilst making operations involving sets available. Two sets are required for the domain and range elements, and the necessary operations include those to output a set, test for membership of an item in a set, and ordering functions on the domain and range items in order to aid efficiency. In these examples, the type 'Rainge' has been deliberately mispellt because Range is a reserved word in Ada.

```
with Set_bounded_G;

generic
  Msize:In Positive;
  Type Domain Is Private;
  Type Rainge Is Private;
  Type Domain_Set Is Private;
  Type Range_Set  Is Private;
  With Function  Gt_Domain       (D1,D2:Domain) Return Boolean;
  With Function  Gt_Range        (R1,R2:Rainge) Return Boolean;
  With Function  Lt_Range        (R1,R2:Rainge) Return Boolean;
  With Procedure Insert_Range    (R:In Rainge;Rs:In Out Range_Set);
  With Procedure Insert_Domain   (D:In Domain;Ds:In Out Domain_Set);
  With Procedure Create_Range_Set (Rs:Out Range_Set);
  With Procedure Create_Domain_Set (Ds: Out Domain_Set);
```

```
  With Function  Is_A_Domain_Member(D :Domain;Ds:Domain_Set)Return
                                                   Boolean;
  With Function  Is_A_Range_member (R :Rainge;Rs:Range_Set) Return
                                                   Boolean;

Package One_to_one_G is

      -----------
```

A clients program that uses this package must provide all the types and formal subprograms in order to instantiate the package. The following excerpt is taken from the Ada specification of a library implementation. It involves a one to one function, Written_by, that relates titles of books to the author(s) that wrote them.

Written_by is defined by :- Written_by : Title $\rightarrowtail\!\!\!\!\rightarrow$ $\mathbb{F}$ Author

The domain item for the function is the type Title, whilst the range item for the function is a set of authors (since a number of authors may have written the book) . A set of titles and a set of author sets are required for the domain and range types.

```
package title_set_pack      is new set_bounded_G(Title,  ---);
package author_set_pack     is new set_bounded_G(Author, ---);
Package set_of_author_sets is new set_bounded_G(Author_set_pack.set,
                                                        ---);
```

In order to instantiate the Written_By function all the generic formal parameters for the one to one function package must be matched up with actual parameters as follows:-

```
Package Written_By_Function Is New One_to_one_G (Msize=>10,
      Domain                 => Title,
      Rainge                 => Author_Set_Pack.Set,
      Domain_Set             => Title_Set_Pack.Set,
      Range_Set              => Set_Of_Author_Sets.Set,
      Gt_Domain              => Gt_Title,
      Gt_Range               => Gt_Author_Set,
      Lt_Range               => Lt_Author_Set,
      Insert_Range           => Set_Of_Author_Sets.Insert,
      Insert_Domain          => Title_Set_Pack.Insert,
      Create_Range_Set       => Set_Of_Author_Sets.Create_Set,
      Create_Domain_Set      => Title_Set_Pack.Create_Set,
      Is_A_Domain_Member     => Title_Set_Pack.Is_A_Member,
      Is_A_Range_Member      => Set_Of_Author_Sets.Is_A_Member);

Written_By : Written_By_Function.Map;
```

The introduction of Ada95 greatly simplifies the construction of packages that are themselves based upon other generic packages. Ada95 allows a complete package to be imported through the generic part of a client package. This makes all the operations in the package available to the client package. The code segment below is an extract from the Ada95 version of the one to one function package.

```
with set_bounded_G;

generic
  Msize : in positive;
  with package domain_set_package is new set_bounded_G(<>);
  with package range_set_package is new set_bounded_G(<>);
  with function ">"(d1,d2:domain_set_package.item_type) return
      boolean;
  with function ">"(r1,r2:range_set_package.item_type) return
      boolean;
```

The package `domain_set_pack` is an instance of the set_bounded_G package. The box notation, (<>), is used to say that no parameters are used in the instantiation at this stage. This allows the types such as `domain_set_pack.item_type`, in the ordering function above, to be used before any actual parameters are given. The actual parameters will be given in the private part of the specification, when the state space is created. It has already been stated that reusable components such as the many to one function package use operations that involve sets as parameters. These operations do not have to be explicitly stated because they are contained in the package instances `domain_set_pack` and `range_set_pack`.

The one to one function modelling Written_by can be instantiated as follows.

```
package title_set_pack      is new set_bounded_G(Title,  ---);
package Author_set_pack     is new set_bounded_G(Author, ---);
Package Set_of_author_sets is new set_bounded_G(Author_set_pack.set,
                                                          ---);


Package Written_By_Function Is New One_to_one_G (
                                     msize,Title_Set_Pack,
                                     Set_Of_Author_Sets_Pack,
                                     -----);
```

It can be seen quite clearly, that the rules in Ada95 for including packages as generic parameters are a vast improvement with regards to the length and complexity of generic parameter lists and their respective instantiation. The two examples above show how generic packages can be constructed from other generic packages using Ada83 and Ada95. The improvement of Ada95 is an extra bonus for this method, because each of the reusable components that model the types found in Z, with the exception of the set, use a set of items as a parameter in some of their operations. Some Z specifications contain many functions, relations and sequences, which when implemented in Ada83 made for very long and complex package specifications. The library implementation mentioned earlier involved 9 instantiated set packages and 6 instantiated function packages. The instantiation of the reusable components required by the Z specification for refinement purposes is much easier using Ada95, it leads to less complex Ada specifications and reduces the time taken for the refinement of the state schema.

8.3 Iterators

An iterator is an operation, or a collection of operations, that can be exported from a component and used to traverse the structure of the component. The use of an iterator in an abstract data type allows a user to access the underlying structure of an object. Booch [Booc87] discusses the use of two types of iterator, namely the active iterator and the passive iterator, in detail along with their advantages and disadvantages. For the purposes of this discussion some explanation of the two types will be given.

8.3.1 The Active Iterator

The active iterator exports the iterator as a type along with operations that act as constructors and selectors. An active iterator is very versatile, but it leaves the abstraction open to abuse. The active iterator as modelled by Booch is given as:-

```
type iterator is limited private;
procedure initialise (the_iterator : in out iterator;
                      with_the_structure : in structure);
procedure get_next(the_iterator  : in out iterator);
function value_of (the_iterator  : in iterator) return item;
function is_done  (the_iterator  : in iterator) return
                                                  boolean;
```

8.3.2 The Passive Iterator

The passive iterator has been chosen for use in the software components modelling the types found in Z because it is safer and it is easier for a developer to use. The passive iterator in the software components is modelled as :-

```
generic
  with procedure process(I        : in item_type;
                         continue : out boolean);
procedure iterate (s : in structure_type);
```

This choice of iterator has its drawbacks when using Ada83 to build utility components that iterate over their constituent underlying components. An example is the function package that was built upon a sets package that must iterate through the underlying set for some operations (see section 4.4). Another example is the construction of basic operations in Z such as distributed union (see section 4.5) and distributed intersection which involve nested iterators from different packages. The problem is that the passive iterator is generic and Ada83 does not allow generic units to be used as generic formal parameters. The active iterator could be used by importing the iterator type and its operations but the ease of use and safety would be lost.

The distributed union procedure discussed above is able to use the passive iterator because Ada95 allows a complete package to be used as a generic parameter and that includes the packages own generic units. It is therefore possible to use nested generic iterators from imported generic packages. The ability to use the passive

iterator is retained and the solution is more elegant than using an active iterator and importing all its operations.

8.4 The Use of Child Packages

This project relies on the extensive use of reusable software components to implement Z specifications. However, many specifications themselves rely on reuse and are written in a style where they may reuse other specifications through promotion. The implementation of these specifications could not be done in Ada83 in an efficient or safe manner. In Ada83 the developer had to chose between making the private types non private and allowing client packages visibility which destroyed the abstraction; or implementing the subsystems of the specification in an excessively large package. In the second case, if other operations or subsystems are to be added to the system then it must obviously be recompiled, but, all the other client packages will need recompilation even if the new additions do not affect them.

The use of child packages allows the private part of a parent package to be visible to the child package. In this way the child package can extend the parent package, without a loss of abstraction or safety and without disrupting any other client packages. This aspect of Ada95 is crucial to the refinement of Z specifications which have schema inclusion. It allows specifications which contain subsystems that share private types to be implemented in an efficient and natural way. The use of child packages will also have an impact on the construction of reusable components. Groups of similar operations can be collected in child packages for a particular type and data model. The use of child packages will enable these cases to be implemented with efficiency without providing operations to cover every instance in one huge package. However, there is a disadvantage, and this lies in the management of the extra packages due to the added complexity involved with using a hierarchy of packages. The problem will be exasperated especially when there are many software components using different data models for each of the basic types in Z, which will themselves consist of a hierarchy of packages.

8.4.1 Use of Child Packages to Implement State Invariants

It has been suggested that this method of refinement will only make a good rapid prototyping method because of the amount of copying of information that would go on because of the need to implement the state invariant [Rann94]. To avoid the state invariant being broken when an operation that changes the state is carried out, the state would be copied to a temporary state and the operation would be applied to the temporary state. The temporary state would then be checked against the state invariant and if it obeyed the state invariant then the temporary state would be copied to the real state. If the temporary state did not obey the state invariant, then an error would be raised.

It was correctly stated that this method would make the final implementation much less efficient because of all the copying of information to and from temporary states and the testing of the state invariant. However, if the specification is refined into a correct implementation of an abstract data type or abstract state machine, then all possible inputs to each operation will have a well defined output. Therefore, it is only necessary to implement the state invariant for testing purposes to ensure all usage of the package produces well defined output. Implementing the state invariant may well highlight preconditions that are implicitly stated in the state schema but left out of the operation schema. In order to use reusable components to translate Z specifications preconditions must be explicitly stated.

The method used in this project implements each operation schema in an Ada package without testing the state invariant before or after an operation is carried out. However, the state invariant can be refined in a separate child package so that it can be applied to each operation in the Ada package implementing the specification via a separate test program. The state invariant for the birthday book specification will be used as an example:-

```
┌─ BirthdayBook──────────────────────
│ known : P NAME
│ birthday : NAME ⇸ DATE
│ cardSent : YEAR↔ NAME ⇸ DATE
│ cardReceived : NAME ↔ YEAR
├──────────────────────────
│ known = dom birthday
│ ∀ y : YEAR• dom (cardSent y ) ⊆ known
└──────────────────────────────────
```

The following Ada specification is a child of the package `B_book` (implementing BirthdayBook. It contains the implementation of the state invariant for the birthday book specification (the refinement of which was discussed at section 5.3.1). It contains one function that applies the state invariant and returns true if it is valid. The state is not an input parameter to the function `is_valid_state` because the package birthday book is an implementation of an abstract state machine. All types and operations declared in the Ada specification `B_book` are visible to this child package. The child package must be generic, although no extra parameters are given, because Ada requires that a child package must be made generic if its parent is a generic package.

```
generic
package b_book.test is
  function is_valid_state return boolean;
end;
```

The state invariant says that the set of people for which cards were sent (for every year) must be contained within the set of people known to the birthday book. It can be implemented by providing the necessary generic formal subprograms to instantiate the `for_all` function. The method of refining a universal quantifier has already been discussed, so only the code will be given, without the refinement details. In the code below, the dot notation in Ada has been used to identify which packages the operations are used from. It is not necessary to include tests to model the declarative part of the state schema to ensure that the types remain the same

because each type is derived from an abstract data type and so will always remain the same.

```
package body b_book.test is

   function is_valid_state return boolean is

      years : year_set_pack.set;


       function for_all_predicate(y:year) return boolean is

          name_to_date: birthday_map.map;
          names,known : name_set_pack.set;

       begin
          b_book.birthday_map.whole_domain(known,birthday);
          b_book.cardsent_map.range_of(y,name_to_date,cardsent);
          b_book.birthday_map.whole_domain(names,name_to_date);

          return b_book.name_set_pack.is_subset(known,names);
       end;

       function for_all_years is new
                 b_book.year_set_pack.for_all(for_all_predicate);


   begin
      b_book.cardsent_map.whole_domain(years,cardsent);
      return for_all_years(years);
   end is_valid_state;

end;
```

This child package can now been used in a test program operating on the birthday book implementation. The instantiation of the package `b_book` and the instantiation of the child package must be given.

```
with b_book.test  ----


   package b_book_test_menu  is new

      b_book(size,a_name,date,year,out_name,out_date,out_year,
                 gt_name,gt_date,gt_year,match_date);

   use b_book_test_menu;


   package State_invariant_test is new b_book_test_menu.test;


   use state_invariant_test;
```

A simple procedure can be used within the test program to test the state invariant.

```
procedure test_state is
begin
   if is_valid_state
   then
     Put_line("state invariant ok ");
   else
     put_line("error in state invariant");
   end if;
end test_state;
```

This can then be applied to check that the state is valid before and after operations from the birthday book are applied.

```
-------

      test_state;
      Add_Birthday(P,D)
      test_state;

-------
```

This method of utilising the state invariants of the state schema, in order to test the operation schemas, has the advantages that the code implementing the invariants will not appear in the delivered software. The implementation of the state invariant only appears in a child package which does not have to be used by or supplied to the client. It is simply a tool that can be bolted onto the software as extra ammunition in the search for errors.

Chapter 9

Conclusions And Future Work

9.1 Introduction

Previous attempts at creating executable code from formal specifications have concentrated mainly on two methods. In the first method, the abstract mathematics of the specification are refined into less abstract structures within the formal language that can be mapped to the target language. This involves creating a new specification and proving that it is equivalent to the original abstract specification. For even medium sized specifications this is an extremely difficult task. The second method implements the specification using functional programming languages that share certain structures and operations with Z to create an animation or prototype. It is easier to implement specifications in functional languages as opposed to imperative languages because they are based upon sets and predicate calculus. However, pure functional languages do not have a concept of state and there are aspects of Z that are not easily implemented with a functional language such as input and output [Good95a]. Functional languages are also widely regarded as being slower than imperative languages and they do not interface well with other languages, which would make integration into an existing system difficult.

A method of turning Z specifications into executable code using a series of available reusable components, written in Ada95, that model the operations found in Z has been developed and tested using Z specifications found in the literature and those contained in internal technical reports [Hayw96].

9.2 Quality of the Reusable Components and the Code Obtained.

The software components have been constructed using modern software engineering principles such as information hiding, modularity and genericity, that have been shown to improve quality [Fent91]. Each of the operations in the software components model a single operation in Z which is well understood and documented [Spiv89]. The implementation will be extremely similar to the Z specification and a code reader will not see any underlying details in the body of the implementation and will have no idea what data structures have been utilised. The great similarity between the specification

and the resulting code helps the developer to see where any errors that may have crept into the implementation are located, simply by reading the code. The specifications that have been used throughout this thesis have been implemented as either Abstract Data Types or Abstract State Machines. Therefore, each of the operations that are available to them have been written in such a way that there is a well defined response for all input. This safety is due to the way in which Z models its predicates and is fully captured by the use of Ada.

9.3 Viability of the Method

This research project has shown that the method of refining formal specifications using reusable software is potentially viable. It does produce code in the form of abstract state machines which can be used in other software and may be used in the final system. This method has enriched an existing imperative language with precisely those types and operations that are found in Z. This enables the Z operations to be translated into their corresponding Ada operations.

Many components can be written to model the types and operations in the Z language using different underlying data structures to allow a whole range of different implementations to be created according to the client's requirements for speed and method of data storage etc. The cost involved in using this method is high to begin with in terms of writing many reusable components and in testing them, but this initial cost is continually mitigated as it is spread over all other implementations which follow. This is true with all methods that make use of reusable software. The advantage here is that Z does not grow, so the task of producing components is finite (within the limits of performance etc).

9.4 Advantages in Rapid Prototyping Terms.

Many papers have been written that animate or prototype a subsection of a specification to test that the specification does meet the customer's requirements and to allow the customer and the developer to get a 'feel' for the final product. The work that is put into creating a prototype of the system is deemed worthwhile even when the resulting

implementation is thrown away. The code created using the reusable components presented in this thesis can be used as an animation or a prototype. However, if the performance of the code meets the systems non-functional requirements then it can also be used in the final software system. A prototype of the system can be quickly implemented, if desired, using the simpler reusable software components to give a customer a workable system to test. For example, components based upon internal data structures may be used in the prototype. However, if file structures are to be used for the real system, then the only major changes that must be made are to the Ada specifications that implement the Z specification state schemas and the software that uses the implementation. These changes will be due to the different reusable components that are used and their resulting instantiation. The actual code in the Ada package body that implements the operation schemas, contained within the Z specification, will be almost identical. This is because the same operations are available for each type regardless of the underlying model. The differences in the Ada body if using components based upon file structures, for example, will be due to including operations such as opening and closing files in the Ada bodies of the translated operation schemas. Any changes to the modelling of the main types can be accounted for by simply instantiating the implementation over the new models.

9.5 Efficiency of the Obtained Code.

The reusable components can be constructed using very efficient underlying data models with efficient operations. However, the main barrier to the efficiency of the obtained code is that the Z specification is forced to be used as the design. This was not the original intention of the designers of Z, since Z is unconcerned about implementation issues. An inefficient architectural model of the state space will result in an inefficient implementation using reusable components, whilst translating a well 'designed' state space will result in a more efficient Ada implementation. However, even if the state model is based upon an efficient state representation, the code obtained from the reusable components still cannot match the code obtained from a manually programmed version. This is because a manually developed program has opportunities to rationalise code in a way that using discrete operations from a reusable component cannot. However, if the code obtained does not match that of manually developed

software, it does not mean that it cannot be used in the final system. The code may still meet the non-functional requirements of the system.

In section 7.3.5 it was stated that it is not possible to predict the space and time complexities of a system until it is built and tested. It is then possible to identify system bottlenecks and tune them. It is possible to improve the efficiency of the Ada implementation by using a component with an underlying data structure that is more suited to the application domain or by rewriting an inefficient Z statement in a more efficient manner. Rewriting Z statements has also been carried out in other forms of refinement to make the proofs easier [MacD89]. However, these improvements may not be sufficient to overcome the inherent architectural inefficiencies. If the performance of the system is still unsatisfactory, then the code (as it stands) cannot be used in the final system. However, the exercise is still worthwhile because it is possible to verify that the system meets the customers requirements by allowing the customer to use it.

9.6 Scaling up to Industrial Sized Specifications.

In chapter two, it was seen that one of the main barriers to the widespread use of formal methods is the lack of modularity in Z specifications. Translating modular specifications has advantages because each state and its operations can be housed in a single Ada package. Moreover, if the specification is written in a modular style then it may be possible to use those parts of the translation that meet the systems performance requirements in the final system and improve the design of only those modules that do not. It has been stated that Z is not concerned with design issues, however, a growing number of researchers have argued that the use of Z will be more widespread if some design decisions are taken when constructing the Z specification, which will in turn make the refinement process easier. These include adding modularity to Z and applying the lessons that have been learned in the construction of software to the construction of specifications. Read [Read92] develops a method of specifying components in Z that follow the Ada package model. This allows Z specifications to be reused and will enable a Z specification to easily map into a corresponding Ada implementation. A

specification written using Read's method should make refinement using reusable components an easier prospect.

In chapter seven, a specification that is used as a test case for various formal methods (the steam boiler) and a real world specification (the aircraft cabin illumination system) were implemented. In both cases, a number of abstract state machines were obtained which each contained operations that implemented some part of the systems functionality, as specified in the Z specification. Both systems were real time systems and so in order to use the operations from the abstract state machines, an outer operating system layer is required to act on interrupts and call the necessary operations. In order to assess if the code obtained from the translated Z specification can be used in the final implementation, the performance must be measured. This has not been done because of the difficulty of simulating the steam boiler and the aircraft cabin illumination equipment and due to the lack of non functional data, such as information on the system response times. Therefore, no concrete claims can be made for using the translated software for the steam boiler and aircraft illumination package in the final implementation. However, in both cases, using reusable components succeeded in implementing the Z specification quickly and produced code that could be used, at the very least, as a prototype of the specification.

9.7 Preserving the Style of the Original Specification

The state schema and its operation schemas can be translated very quickly into the appropriate procedure or functions from the relevant Ada component. This method enables Z specifications to be implemented in the spirit in which they were originally written. However, this may not be the best way of writing them in terms of the ease of implementation and the resulting efficiency of the code. Many Z specifications are written in a style where they can be reused in other specifications through state inclusion or operation promotion to enable complicated specifications to be built up in stages. This method allows systems that are specified in this manner to be implemented by using child packages to allow operations in packages that implement different states to share private types. The use of child packages closely follows the Z model of inclusion and is a very important aspect of refinement with this method. This method

can successfully translate all of the operators available in the schema calculus provided that the necessary work has been carried out in Z, so that the final schemas are valid, the operators are well suited and the final schema is correct with regards to the intention of the specifier.

9.8 Non Functional Requirements.

The developer must select which type of Ada component should be used according to the client's requirements for space, speed, bounded or unbounded components and internal or external storage. The developer must then manipulate the reusable Ada components to provide a valid state upon which the operations specified in Z can act. This is necessary because these performance details will not appear in the Z specification. The developer of the generic package that implements the specification must provide the models for any generic types and the code for any generic formal subprograms. The developer must also create an interface for the Ada package, and interpret the input or output devices required. The Z specifications '?' and '!' notation do not state the format for either the input or output data explicitly. A generic unit for output enables many instances of an output procedure for a given type to be instantiated. This allows the type to be output to different devices, such as the screen, a printer, or even as the input to another operation.

9.9 The Use of Ada95

All of the reusable software components have been implemented as ADTs and with generality in mind. Ada has provided the tools to create ADTs through its use of packages and private type declarations. The use of generic parameters has reduced the complexity of the actual software components, but unfortunately, this complexity has to go somewhere and it can be found in the Ada package specifications used to implement the Z specification. The introduction of Ada95, with its ability to use generic formal packages has removed much of the complexity and reduced the number of generic parameters that are required with its ability to import complete packages as generic parameters. This aspect is also important when components are to be built upon other underlying components using an iterator procedure to traverse the underlying data

structure. Previously generic units could not be used as generic formal parameters, but in Ada95 the whole package can be used as a generic parameter, including all of the generic units in each package. It would be possible to use C++ for this method of refinement, but with the absence of the generic feature in C++, the process would be harder.

Another bonus that Ada95 brings is its use of child packages. This aspect is crucial to the implementation of specifications that are based upon multiple states or use inheritance. Child packages also allow a user to implement the state invariant in a separate package that can be used to test each of the operations contained in the implementation. It is not necessary to implement the state invariants and test them before and after each operation if any implicit preconditions are made explicit in the operation schemas, but this technique can be used to test the implementation and to identify any preconditions that are missing from the operation schema.

9.10 Criticisms of the Method

Using reusable components to directly translate Z specifications can be criticised for the following reasons:

- The Z specification must be written in a constructive style. If it is not written in a constructive style, then some work is necessary to make it so. However in order to arrive at executable code from a non-constructive specification some extra work must be done regardless of the refinement method used. This is essentially the point of Hayes i.e that specifications are not necessarily executable [Haye89]. Fuchs recongises the fact that if code is to be generated from the specification then work can be carried out to make it executable [Fuchs92].
- The Z specification must use finite structures. Functional languages allow the definition of (conceptually) infinite data structures, but if a computation is to be performed some finite portion of the structure must be used.
- Operations must be deterministic. However, in some cases generic procedures may be used to defer implementation choices to the software using the Ada translation.
- The Z specification must use explicit preconditions in the operation schemas.

- The method forces Z to be used as a design document which is not the main purpose of Z specifications.
- The performance of the Ada implementation mainly depends upon the way the state and its operations have been modelled. In order for the implementation to have a chance of being used as the final system, a good 'design' is required for the state model and operations must be written in an 'efficient' manner. This reduces the specifiers freedom.
- The code obtained cannot match that of manually developed code in all cases. This is due to the lack of opportunity of rationalising code when using a series of Ada operations.
- The cost of constructing components is high, although this cost is mitigated as it is spread over each implementation that uses the components.
- Large specifications, that are not written in a modular style, are more difficult to implement due to operations having access to many states. As a result, it is not easy to map this type of specification into a series of Ada packages.
- The process is not automated.

9.11 Advantages of Using Reusable Components.

- An implementation of a Z specification using operations from reusable components has less opportunity of introducing errors than a manually implemented version which contains much detailed new coding. Barnes [Barn95 pg15] states that one of the most important objectives of software engineering is to keep detailed new coding to a minimum. Reuse of existing code aids this goal.
- Imperative languages such as Ada95 have performance advantages over their functional counterparts, they have better facilities to interface with other systems and they are used to a much greater extent in industry.
- The implementation of the Z specification can be used as part of the final system provided it meets the performance requirements of the system being built. If not, the implementation of the original Z specification (and some software to use its operations) will at least allow the developer to verify that the customers requirements are being met.

- A more concrete design in Z, perhaps using more efficient data representation, may be necessary, but reusable components may still be used to implement this specification as highlighted in section 7.3.5.1.
- Using reusable components in Ada95 enables specifications with multiple states and schema inclusion to be implemented as seen in the steam boiler and aircraft illumination specifications shown in chapter 7. Specifications of this type have previously caused problems for methods that use functional languages as the target language, as discussed in chapter 2.
- Using an imperative language enables the developer to handle the state and input/output in an easy manner, without having to pass the state around as a parameter, or resorting to the use of a monad.
- Using reusable components in Ada95 allows detailed information that is not stated in the Z specification (but is necessary for its implementation in a programming language) to be deferred to the package that uses the Ada translation of the Z specification.

9.12 Future work

This project was started using Ada83, but in February 1995, the new version of Ada (Ada95) was certified for use. The reusable Ada83 components were then translated into Ada95 to take advantage of its new features. If the project had started in Ada95 then the reusable components may have been written in a different way. The new object oriented features of Ada95, such as tagged types and inheritance could have been used for the components because each of the main types available in Z are dependant on sets. One area of future work could be to investigate these new object oriented features with respect to creating an alternative library of reusable components. This area of research, however, will not affect the refinement of Z specifications because the new object oriented components will contain the same operations as the old components.

So far, all of the implementations of Z specifications have been completed by hand. However, there is a need to investigate automation and tool support if the method is to be used seriously in the creation of software. This could be done by using a typesetting

tool such as CadiZ[1] and instead of using the Troff commands to output the schema text to a screen, the operations from the reusable components can be selected. Obviously, this is an oversimplification because some of the refinement process requires some human intelligence and the process is not a straight forward as "if you find this symbol, then replace it with this code". One bonus is that the code generator can translate the Z specification with no knowledge of the speed or memory requirements required, since they can be tuned by the developer in selecting different reusable components to model the state space.

[1] CadiZ is available from York Software Engineering Limited, University Of York.

References

[Abri91] Abrial J.R, Lee M.K.O, Neilson D.S, Scharbach P.N. "The B-method" VDM 91 : Formal Software Development Methods, vol 551 of Lecture Notes in Computer Science, Springer-Verlag 1991, pp398-405.

[Back88] Back R.J.R. "Calculus of Refinements For Program Derivations." Acta Informatica, vol 25, 1988, pp593-624.

[Bale94] Bale.S, Hayward.J, "A Bounded Set as a Reusable Component for Reifying Z Specifications", Technical Report (M-94-3), Department of Mathematics and Computing, University of Glamorgan.

[Barb94] Barbey.S, Buchs. D "Testing Abstract Data Types Using Formal Specifications", Eurospace Ada Europe 94 Symposiun Proceedings. Lecture Notes In Computer Science, no 887, pp76-89, 1994.

[Bard92] Barden R. "Support For Using Z." 7th Z User Meeting, 1992.

[Bard94] Barden.R, Stepney. S, Cooper.D "Z In Practice", Prentice Hall, 1994, ISBN 0-13-124934-7.

[Barn95] Barnes. J. "Programming in Ada95", Addison-Wesley 1995, ISBN 0-201-87700-7.

[Basi87] Basili.V.R, Selby.R.W " Comparing the Effectiveness of Software Testing Strategies". IEEE Transactions on Software Engineering 1987, Vol SE-13 no 12, pp1278-1295

[Baye72a] Bayer.R "Symmetric Binary B-trees : Data Structure and Maintenance Algorithms". Acta Informatica (1), pp290-306, Springer-Verlag, 1972.

[Baye72b] Bayer.R. and McCreight. E. "Organisation and Maintenance of Large Ordered Indexes." Acta Informatica (1), pp 173-189, Springer-Verlag, 1972.

[Boeh87] Boehm B.W. "Improving Software Productivity." IEEE Computer, Sept 1987, pp43-57.

[Booc87] Booch G. "Software Components With Ada - Structures, Tools and Systems." Benjamin Cummings, ISBN 0-8053-0610-2, 1987.

[Bowe93] Bowen J., Stavridou V. "The Industrial Take-up of Formal Methods in Safety-Critical and Other Areas : A Perspective." Formal Methods Europe 1993 in Lecture notes in Computer Science Vol 670, Springer-Verlag, pp 183-195.

[Bowe95] Bowen J.P, Hinchey, M.G "Ten Commandments of Formal Methods" IEEE Computer 28(4), pp55-63, April 1995.

[Burn77] Burnstall R.M., Darlington J. "A Transformation System for Developing Recursive Programs." Journal of the Association for Computing Machinery (ACM), vol 24, no1, 1977, pp 44-67.

[Carr91] Carrington. D. "ZOOM Workshop Report", Proceedings of the Sixth Annual Z User Meeting 1991.

[Clif95] Clifford A, Zarzycki L. "Combining Formal Notations to Develop Practical Formal Methodologies." Proceedings of 4th Software Quality Conference, pp 413-422, 1995.

[Crai95a] Craigen D, Gerhart. S, Ralston T., "Formal Methods Reality Check : Industrial Usage." IEEE Transaction on Software Engineering, Vol21, no 2, pp90-98, 1995.

[Crai95b] Craigen D., Gerhart S, Ralston T., "Formal Methods Technology Transfer: Impediments and Innovation." In Applications of Formal Methods, M.G. Hinchey, J.P.Bowen (eds), pp 399-419. 1995 ISBN 0-13-366949-1.

[Dijk75] Dijkstra.E.W "Guarded Commands, Nondeterminancy and Formal Derivation of Programs." Communications of the ACM, 18(8) 1975.

[Dill90] Diller A.Z. "An Introduction to Formal Methods." John Wiley and Sons, 1990.

[Drap92] Draper.C "Practical Experiences of Z and SSADM". 7th Z user Meeting Proceedings 1992.

[Fent91] Fenton N.E. "Software Metrics" Chapman & Hall, 1991, ISBN 1-85032-242-2.

[Fidg92] Fidge. C.J "Specification and Verification of Real Time Behaviour using Z and RTL", Second International Symposium Proceedings, Formal Techniques In Real Time And Fault Tolerance Systems, (1992) , Springer-Verlag, pp 393-409.

[Fuch92] Fuchs N.E."Specifications are (Preferably) Executable." Software Engineering Journal, Vol 7, No 5, pp323-333, 1992.

[Gerh94] Gerhart.S, Craigen D, Ralston T. "Experience with Formal Methods in Critical Systems." IEEE Software, January 1994, pp21-28.

[Glas91] Glass and Noiseux, "Software Maintenance Guidebook", Prentice-Hall, 1991.

[Goel91] Goel.A, Sahoo.S, "Formal Specifications and Reliability an Experimental Study", Proceedings : 1991 International Symposium on Software Reliability Engineering (Cat no 91 Th0336-5) pp139-142.

[Good93] Goodman.H.S "Animating Z specifications in Haskell Using a Monad" School of Computer Science, University of Birmingham, Technical Report CSR-93-10, 1993.

[Good95a] Goodman.H.S "The Z-into-Haskell Tool-kit : An Illustrative Case Study." Proceeding of the 9th Z User Meeting, 1995, Springer-Verlag.

[Good95b] Goodman.H.S "The Z-into-Haskell Tool-Kit" School Of Computer Science, University Of Birmingham. Technical Report (CSR-95-1), 1995.

[Grav91] Gravel A.M, Henderson P. "Why Execute Formal Specifications" Proceedings of Mathematical Structures for Software Engineering. Clarendon Press, pp 153-164, 1991.

[Hame95] Hamer U. Peleska J. "Z Applied to the A330/340 CIDS Cabin Communication System". In "Applications of Formal Methods," Eds Hinchey M.G. Bowen J.P. Prentice Hall, 1995, ISBN 0-13-366949-1.

[Hart96] Hartel. P.H. "Benchmarking Implementations of Functional Languages Using 'Pseudoknot' a Float Intensive Benchmark" Journal of Functional Programming, vol 6, part 4, 1996. Cambridge University Press.

[Hayw96] Hayward.J "The Construction Of a Z Specification by Iteration and Enrichment", Technical Report (UG-M-96-3), Department Of Mathematics and Computing, University Of Glamorgan.

[Hayw95] Hayward .J , Bale S. “Refinement of Z Specifications using Reusable Software Components in Ada” Proceedings of the ACM Sig-Ada Tri-Ada’95 conference.

[Hayw94] Hayward.J, Bale.S , "An Algebraic Specification of the Abstract Data Type Set", Technical Report (M-94-2), Department of Mathematics and Computing, University of Glamorgan.

[Haye87] Hayes.I.J “Specification Case Studies”. Prentice Hall International, 1987.

[Haye89] Hayes.I.J , Jones.C.B “Specifications are not (Necessarily) Executable.” Software Engineering Journal (4), pp330-338, 1989.

[Haye96] Hayes. I, “Supporting Module Reuse in Refinement” Science of Computer Programming, 27, pp175-184, 1996.

[Hoar93] Hoare C.A.R “Algebra and Models” Software Engineering Notes, vol 18, No 5, pp1-8, 1993.

[Holl96] Holloway C.M, Butler R.W. “Impediments to Industrial Use of Formal Methods” IEEE Computer, April 1996, pp25-26.

[Horr84] Horrowitz E, Munson J. "An Expansive View of Reusable Software" IEEE Transactions on Software Engineering, vol se-10 (5) 1984.

[Huda94] Hudak P., Jones M. “Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity.” Department of Computer Science, Yale University, New Haven, CT 06518.

[Jack85] Jackson M.I, “Developing Ada Programs Using the Vienna Development Method (VDM)” Software Practice and Experience (1985) ,Vol 15(3), pp305-318.

[Jado89] Jadoul L., Duponcheel L., Puymbroeck W.V. “An Algebraic Data Type Specification Language and its Rapid Prototyping Environment”. Proceedings 11th International Conference of Software Engineering. IEEE Computer Society Press 1989, pp74-84.

[John90] Johnson. M, Sanders P, “From Z Specifications to Functional Implementations”. Proceedings 4th Annual Z User Meeting, Springer-Verlag, 1990, pp86-112.

[King89] King S., SØrensen I.H. “From Specification, Through Design, To Code : A Case Study In Refinement”.
In Theory and Practice of Refinement Approaches to the Formal Development of Large-Scale Software Systems. McDermid.J.A (editor), Butterworths 1989. pp90-121.

[King90] King.S. "Z and the Refinement Calculus ",
VDM' 90. VDM and Z - Formal methods in Software Development. Third International Symposium of VDM Europe Proceedings. Springer-Verlag ,1990, pp 164 - 188.

[Knot92] Knott R.D, Krause P.J “The Implementation of Z Specifications Using Program Transformation Systems- The SuZan Project”. The Unified Computing Laboratory - IMA Series, Vol 35, pp207-220, Clarendon Press 1992.

[Lano92] Lano K., Haughton H. “Reuse and Adaptation of Z Specifications” Seventh Annual Z User Meeting, 1992.

[Lien80] Lientz B.P, Swanson E.B. "Software Maintenance Management", Addison-Wesley, 1980.

[Litt92] Litteck H.J, Wallis P.J.L "Refinement Methods and Refinement Calculi." Software Engineering Journal, vol 7, no 3, May 1992, pp219-229.

[Love92] Love.M "Animating Z Specifications in SQL*forms3.0" Seventh Annual Z User Meeting 1992.

[Maud91] Maude. T, Willis G. "Rapid Prototyping - The Management of Software Risk", Pitman Publishing, 1991, ISBN 0 273 033093.

[Mac D89] MacDonald R., Sennet C. "Refinement of Specification Versus Refinement of Design." In Theory and Practice of Refinement Approaches to the Formal Development of Large-Scale Software Systems. McDermid.J.A (editor), Butterworths 1989. pp122-133.

[Mc Co93] McConnell. S "Code Complete - A Practical Handbook of Software Construction", Microsoft Press, 1993, ISBN 1-55615-484-4.

[McDe89] McDermid J.A "The Theory and Practice of Refinement - Approaches to the Formal Development of Large-Scale Software Systems", Butterworths, 1989, ISBN 0-408-03981-7.

[Mink95] Minkowitz, C, Rann D, Turner J.H, "A C++ Library for Implementing Specifications." Proceedings Workshop on Industrial Strength Formal Specification Techniques. IEEE Computer Society Press, 1995.

[Mitc94] Mitchell R., Loomes M, Howse J, "Structuring Formal Specifications - a Lesson Relearned." Microprocessors and Microsystems, Vol 18, no 10, Butterworth-Heinemann Ltd1994.

[Morri87] Morris, J.M "A Theoretical Basis for Stepwise Refinement and the Programming Calculus." Science of Computer Programming, vol 9, 1987, pp287-306.

[Morre92] Morrey.I, Siddiqi. J, Briggs. J, "Z Animation In LISP" Proceedings 5th International Conference on : Putting into Practice Methods and Tools for Information System Design". 1992. IUT de Nantes.

[Morg94] Morgan C.C "Programming from Specifications" Series in Computer Science. Prentice Hall International ,1994, 2nd Edition. ISBN 0-13-123274-6.

[Nise85] Nise N, Mc kay C, Dillenhunt D, Kim N, Griffin C "A Reusable Software System" Proceedings of the AIAA / ACM / NASA / IEEE Computers in Aerospace V Conference, Long Beach , California 1985.

[Norc91] Norcliffe and Slater "Mathematics of Software Construction" 1991 Ellis Horwood, ISBN 0-13-563388-5

[Rann94] Rann.D, Turner.J, Whitworth.J "Z : A Beginners Guide" Chapman and Hall, 1994, ISBN 0412-55-660-X.

[Read92] Read. T.J "Formal Specification of Reussable Ada Software Packages" In Towards Ada9X, pp98-117, A. Burns (ed), IOS Press, 1992.

[Samp90] Sampaio A., Meira S. "Modular Extensions to Z". Third Symposium of VDM Europe Proceedings, 1990, pp211-232.

[Shen92] Shen. J. and Cormack. G.V "On Generic Formal Parameters In Ada9x" Ada letters Vol xii, number 2, pp110-116, May/June 1992.

[Sher94] Sherrel L.B, Carver.D.L "Experiences in Translating Z Designs to Haskell Implementations." Software Practice and Experience, vol.24 (12), 1994.

[Sher95] Sherrel L.B, Carver D.L "FunZ: An Intermediate Specification Language." The Computer Journal, Vol 38, No 3, 1995.

[Sidd93] Siddiqi J., Morrey I. "Towards CASE Tools for Prototyping Z Specifications", Proceeding of 6th International Workshop on Computer Aided Software Engineering. 1993, pp 166-73.

[Skan88] Skansholm J. "Ada From the Beginning." Addison-Wesley,1988 ISBN 0-201-17522-3.

[Skan97] Skansholm J. "C++ From the Beginning." Addison Wesley Longman, 1997, ISBN 0-201-40377-3.

[Smit91] Smith P., Keighley R., "The Formal Development of A Secure Transaction Mechanism". Proceedings VDM'91 : Formal Software Development Methods pp457-476. Springer-Verlag 1991.

[Spiv89] Spivey J.M. "The Z Notation, a Reference Manual." Prentice-Hall, 1989, ISBN 0-13-983768-X.

[Thom93] Thomas M., " The Industrial Use of Formal Methods" Microprocessors and Microsystems, Vol 17, No 1, 1993.

[Thom96] Thompson S. "Haskell - The Craft of Functional Programming." Addison Wesley Longman, 1996, ISBN 0-201-40357-9

[Utti95] Utting M. "Animating Z : Interactivity, Transparency and Equivalence." Proceedings Asia Pacific Software Engineering Conference. IEEE Computer Society Press 1995, pp295-303.

[Vale91] Valentine S. "Z-- An Executable Subset of Z", 6th Annual Z User Meeting 1991.

[Wadl95] Wadler, P. "How to Declare an Imperative". Proceedings of International Logic Programming Symposium, MIT Press 1995.

[Weis93] Weiss. M, "Data Structures and Algorithm Analysis in Ada" Benjamin/Cummings, 1993, ISBN 0-8053-9055-3.

[West95] West.M.M, "Types and Sets in Godel and Z" 9th International Conference of Z Users 1995, Lecture Notes in Computer Science 967, Springer-Verlag.

[West92] West.M.M, Eaglestone.B.M "Software Development: Two Approaches to the Animation of Z Specifications using Prolog" Software Engineering Journal, vol 7, no 4, July 1992, pp264-276.

[Woodc96] Woodcock J., Davies J. "Using Z Specification, Refinement and Proof", Prentice-Hall, ISBN 0-13-948472-8, 1996.

[WoodK91] Wood.K.R, "The Elusive Software Refinery - A Case Study in Program Development" 4th Refinery workshop, BCS Workshops in Computing, 1991, Springer-Verlag.

[WoodK93] Wood. K.R , "A Practical Approach to Software Engineering using Z and the Refinement Calculus ", Software Engineering Notes 18,5 1993 pp79-88.

[WoodW91] Wood W.G, Place, P.R.H,Luckham D.C, Mann W, Sanker S. "Formal Development of Ada programs using Z and Anna - a case study." Software Engineering Institute, Pittsburgh,CMU/SEI-91-1TR. 1991.

[Word92] Wordsworth J.B. "Software Development With Z." Addison-Wesley, 1992, ISBN 0-201-62757-4.

[Zweb95] Zweben. S.H , Edwards.S.H, Weide,B.W, Hollinsworth.J.E "The Effects of Layering and Encapsulation on Software Development Cost and Quality." IEEE Transactions on Software Engineering vol. 21, no 3, March 1995, pp200- 207.

APPENDICES

APPENDIX 1.

List of software components and specification case studies.

The following reusable components have been constructed throughout the duration of this project.

Set Packages

Internal

Set_bounded_G	- Set package using array model. Set is packed and bounded.
Set_tree_G	- Set package built using a binary tree package, based upon internal memory.

External

Set_file_G	- Set package using direct access file structure.
Set_cache_G	- Set package using direct access file structure and internal memory cache.
Set_btree_G	- Set package built using a btree package, based upon external memory.

Function and Relation Packages

Internal

One_to_one_bounded_G	- One to one function package based upon array structure.
Many_to_one_bounded_G	- Many to one function package based upon array structure.
Many_to_many_bounded_G	- Many to many relation package based upon array structure.

One_to_one_tree_G	- One to one function package built on top of a binary tree package.
Many_to_one_set_G	- Many to one package built on top of the set package Set_bounded_G.

External

One_to_one_file_G	- One to one function package based upon direct access file.
Many_to_one_file_G	- Many to one function package based upon direct access file.
Many_to_many_file_G	- Many to many relation package based upon direct access file.
One_to_one_btree_G	- One to one function package built on top of a btree package.

Sequences

Internal

Sequence_bounded_G	- Sequence package based upon array model.

External

Sequence_file_G	- Sequence package based upon direct access file.

Specification Case studies

The following complete specifications have been translated using this method. All of the translated Ada packages from the specifications have been executed by instantiating them and using them in other programs. However, the steam boiler and aircraft illumination packages have only been compiled due to the difficulty in simulating the steam boiler and aircraft illumination systems, which would use the operations available in the translated Ada packages. In both Z specifications, the real time aspects were not included.

The Steam Boiler	- Implementation of steam boiler specification given in [Buss96].
Aircraft Cabin	- Implementation of real world aircraft cabin illumination specification, as given in [Hame95].
Library	- Package implementing large library specification. Specification contains 6 functions and 6 given types. This specification has been expanded but the original can be found in [Dill90].
Birthday Book	- Implementation of Spiveys birthday book [Spiv89].
Expanded birthday	- An expanded version of birthday book using a higher order book relation.
Filling Station	- Implementation of a specification for a petrol filling station as specified in [Norc91].
Planepac	- Implementation of a seat allocation system for aircraft as specified in [Ligh91].

Class Managers Assistant	- Implementation of Wordsworth's class managers assistant [Woodc96], Assistant showing the refinement of schema types and bindings.
Staff_Db	- Collection of database systems [Dill90]. Implementation Phone_Db highlighted the refinement of multiple state specifications and Departmental_Db those involving extension and inclusion.

APPENDIX 2.

Lists Z statements and operators and gives the required Ada operation(s) to translate them. Guidelines for the translation of specifications are also given.

Logic

Z statement	Refined by
$\neg P$	not P
$\wedge P$	and P
$\vee P$	or P
$P \Rightarrow Q$	If P then Q
$P \Leftrightarrow Q$	(not P and not Q) or (P and Q)
$\exists D \mid P \cdot Q$	Instantiate There_exists function from package according to D (see note below), with functions modelling P and Q.
$\forall D \mid P \cdot Q$	Instantiate For_all function from package according to D, with functions modelling P and Q.

NB. The quantifiers can be used over sets, sequences, functions and relations.

Numbers

Z symbol	Ada
$\mathbb{Z}$	Integer
$\mathbb{N}$	Natural
$\mathbb{N}_1$	Positive
+	+
-	-
÷	÷
*	*
mod	mod
$\leqslant$	<=
$\geqslant$	>=
<	<
>	>
=	=

The following text lists refinements laws for common Z constructs using reusable software components.

I	- An Item	D	- A domain item
S_q	- A Sequence	R	- A range item
S	- A Set	M	- A Map
S_S	- A Set of sets	T	- A type
S_{qSq}	- A Sequence of sequences	P	- A predicate
E	- An expression		

Refinement laws for Set

General statement	Refined by
$S' = S \cup \{I\}$	Insert (I, S)
$S' = S \setminus \{I\}$	Delete(I, S)
$S_1 = S_2$	Is_equal (S_1, S_2)
$S_1 \neq S_2$	not Is_equal (S_1 , S_2)
$S = \varnothing$	Is_empty_set (S)
$\# S$	Size_of (S)
$I \in S$	Is_a_member (I,S)
$I \notin S$	not Is_a_member (I, S)
$S_1 \subseteq S_2$	Is_a_subset (S_1 , S_2)
$S_1 \cup S_2$	Union (S_1, S_2)
$S_3 = S_1 \cup S_2$	Union (S_1 , S_2 , S_3)
$S_1 \setminus S_2$	Difference (S_1, S_2)
$S_3 = S_1 \setminus S_2$	Difference (S_1 , S_2 , S_3)
$S_1 \cap S_2$	Intersection (S_1, S_2)
$S_3 = S_1 \cap S_2$	Intersection (S_1 , S_2 , S_3)
$\bigcup S_S$	Dist_union (S_S, S)
$\bigcup S_S$	Dist_inter (S_S, S)
$\{T \mid P \cdot E\}$	Instantiate generic Set_comprehension procedure from package according to T, with functions modelling P and E (an

identity function can be used if E is not required).

Refinements laws for sequence

General statement	Refined by
$S_q \frown \langle I \rangle$	Construct (I ,S_q)
$S_{q1} \frown S_{q2}$	Concatenate (S_{q1} ,S_{q2})
$S_{q3} = S_{q1} \frown S_{q2}$	Concatenate(S_{q1} ,S_{q2} , S_{q3})
$=$	Is_equal (S_{q1} , S_{q2})
$\neq$	not Is_equal (S_{q1} , S_{q2})
$I \in \text{ran } S_q$	Is_a_member (I , S_q)
$I \notin \text{ran } S_q$	not Is_a_member (I , S_q)
$S_q = \langle \rangle$	Is_empty_sequence(S_q)
$S_q \neq \langle \rangle$	not Is_empty_sequence (S_q)
$\# S_q$	Size_of (S_q)
rev (s_q)	Rev(S_q)
S_{q2} = rev(S_{q1})	rev(S_{q1} , S_{q2})
tail (S_q)	tail (S_q)
S_{q2} = tail (S_{q1})	tail (S_{q1} , S_{q2})
last (S_q)	last(S_q , I)
head (S_q)	head (S_q , I)
front(S_q)	front (S_q)
S_{q2} = front (S_{q1})	front (S_{q1} , S_{q2})
ran S_q	range_of (S_q , S)
$S_q \upharpoonright \{S\}$	Sequence_range_restrict (S_q , S)
$S_q \upharpoonright \{I\}$	Sequence_range_restrict (S_q , I)
$S_{q2} = S_{q1} \upharpoonright \{S\}$	Sequence_range_restrict (S_{q1} , S , S_{q2})
$S_{q2} = S_{q1} \upharpoonright \{S\}$	Sequence_range_restrict (S_{q1} , S , S_{q2})
$\{S\} \upharpoonleft S_q$	Sequence_domain_restrict (S_q , S)
$\{I\} \upharpoonleft S_q$	Sequence_domain_restrict (S_q , I)
$S_{q2} = \{S\} \upharpoonleft S_{q1}$	Sequence_domain_restrict (S_{q1} , S, S_{q2})
$S_{q2} = \{S\} \upharpoonleft S_{q1}$	Sequence_domain_restrict (S_{q1} , S, S_{q2})
$\frown / S_{qSq}$	Dist_concatenate (S_{qSq}.S_q)

Refinement Laws for One to One function

General Statement	Refined by
$M' = M \cup \{ d \mapsto r \}$	Bind (d , r , M)

$M' = M \setminus \{d \mapsto r\}$	Unbind (d , r , M)
$M_1 = M_2$	Is_equal (M_1 , M_2)
$M_1 \neq M_2$	not Is_equal (M_1 , M_2)
$\# M$	Extent_of (M)
$\{d \mapsto r\} \in M$	Is_a_member (d , r , M)
$\{d \mapsto r\} \notin M$	not Is_a_member (d , r , M)
dom (M)	Whole_domain (S, M)
ran (M)	Whole_range (S, M)
$M \rhd \{ S \}$	Range_restrict (S , M)
$M \rhd \{ I \}$	Range_restrict (I , M)
$M1 = M_2 \rhd \{ S \}$	Range_restrict (S ,M_1 , M_2)
$M_1 = M_2 \rhd \{ I \}$	Range_restrict (I , M_1 , M_2)
$M \unrhd \{ S \}$	Range_subtract (S , M)
$M \unrhd \{ I \}$	Range_subtract (I , M)
$M1 = M_2 \unrhd \{ S \}$	Range_subtract (S ,M_1 , M_2)
$M_1 = M_2 \unrhd \{ I \}$	Range_subtract (I , M_1 , M_2)
$\{ S \} \lhd M$	Domain_restrict (S , M)
$\{ I \} \lhd M$	Domain_restrict (I , M)
$M1 = \{ S \} \lhd M_2$	Domain_restrict (S ,M_1 , M_2)
$M_1 = \{ I \} \lhd M_2$	Domain_restrict (I , M_1 , M_2)
$\{ S \} \unlhd M$	Domain_subtract (S , M)
$\{ I \} \unlhd M$	Domain_subtract (I , M)
$M1 = \{ S \} \unlhd M_2$	Domain_subtract (S ,M_1 , M_2)
$M_1 = \{ I \} \unlhd M_2$	Domain_subtract (I , M_1 , M_2)
$I \in \text{dom } M$	Is_a_domain_element (I , M)
$I \notin \text{dom } M$	not Is_a_domain_element (I ,M)
$I \in \text{ran } M$	Is_a_range_element (I , M)
$I \notin \text{ran } M$	not Is_a_range_element (I , M)
$S_2 = M (S_1)$	Relational_image (S_1 , S_2 , M)
$S_2 = M \sim S_1$	Inv_relational_image (S_1 , S_2 ,M)
$D = M \sim R$	Domain_of (D, R, M)
$R = M (d)$	Range_of (d ,r , M)
$M' = M \oplus \{d \mapsto r\}$	Function_override (d , r , M)
$M_2 = M_1 \oplus \{d \mapsto r\}$	Function_override (d , r , M_1 , M_2)
$M_3 = M_1 \oplus M_2$	Function_override (M_1, M_2 , M_3)
$M' = M \oplus M_2$	Function_override (M, M_2)

Refinement Laws for Many to One function

This is as the one to one function except for the following :-

General Statement	Refined by
$S = M \sim I_1$	Domain_of (S , I_1 , M)
$M' = M \oplus \{d \mapsto r\}$	Function_override (d , r , M)
$M_2 = M_1 \oplus \{d \mapsto r\}$	Function_override (d , r , M_1 , M_2)
$M_3 = M_1 \oplus M_2$	Function_override (M_1, M_2 , M_3)
$M' = M \oplus M_2$	Function_override (M, M_2)

Refinement Laws for Many to Many relation

This is as the Many to one function except for the following :-

General Statement	Refined by
$S = M (R)$	Range_of (S, I, M)
$M = M_1 ⨾ M_2$	Rel_comp(M_1,M_2 , M)
$M = M_1 \circ M_2$	Rel_comp(M_2 ,M_1 , M)

Guidelines for Translation

State schema

- A Specification based upon a named state is implemented as an Abstract State Machine.
- In specifications using the binding notation (θ), the schema used as a type should be implemented as an Abstract Data Type to allow multiple instances of itself.
- Each given type will require an instantiation of a sets package.
- A small specification using inclusion can be implemented in a single package, giving access to all types contained within the specification.
- A large specification using inclusion should be implemented using child packages to allow access to all types contained across packages and to avoid a cumbersome implementation with a large interface.

- The order of instantiation is important. Instantiate packages in normal order and remember that function, sequence and relations will require set packages as generic actual parameters.
- The package interface must contain formal generic parameters for each of the actual generic parameters used for the instantiation of the reusable components.
- The state invariant can be safely ignored provided that explicit preconditions appear in each operation.
- The state invariant can be implemented in a child package only as an aid for the testing process.

Operation Schemas

- Most simple statements can be translated using a single one of the rules given above. Complex statements, however, will be made up of many Z operations and will require some analysis in order to translate them. One way to ease the translation of complex statements is to translate the statement in normal order and then reverse the statements to achieve applicative order.
- Preconditions are translated using an 'If' statement and can be negated in order to use the 'elsif' structure to reduce indentation for schemas with many pre-conditions.
- Preconditions can be removed from the refined code if an operation refining a postcondition raises an exception identical to the precondition.
- The operation schemas can be made robust by raising exceptions for the logical complement of each precondition. However, the order for testing and raising exceptions may be of importance and should be investigated (see 6.3.4).
- Be economical in the creation of new types. Don't create a type if it is not required for use elsewhere. For example it is not necessary to create a new set simply to count the number of items in the new set.
- Make sure operation schemas contain explicit preconditions.

- If an operation schema is non-deterministic, attempt to contain the non-determinism as a generic parameter supplied to the operation. If this is not possible then some refinement within Z must take place.