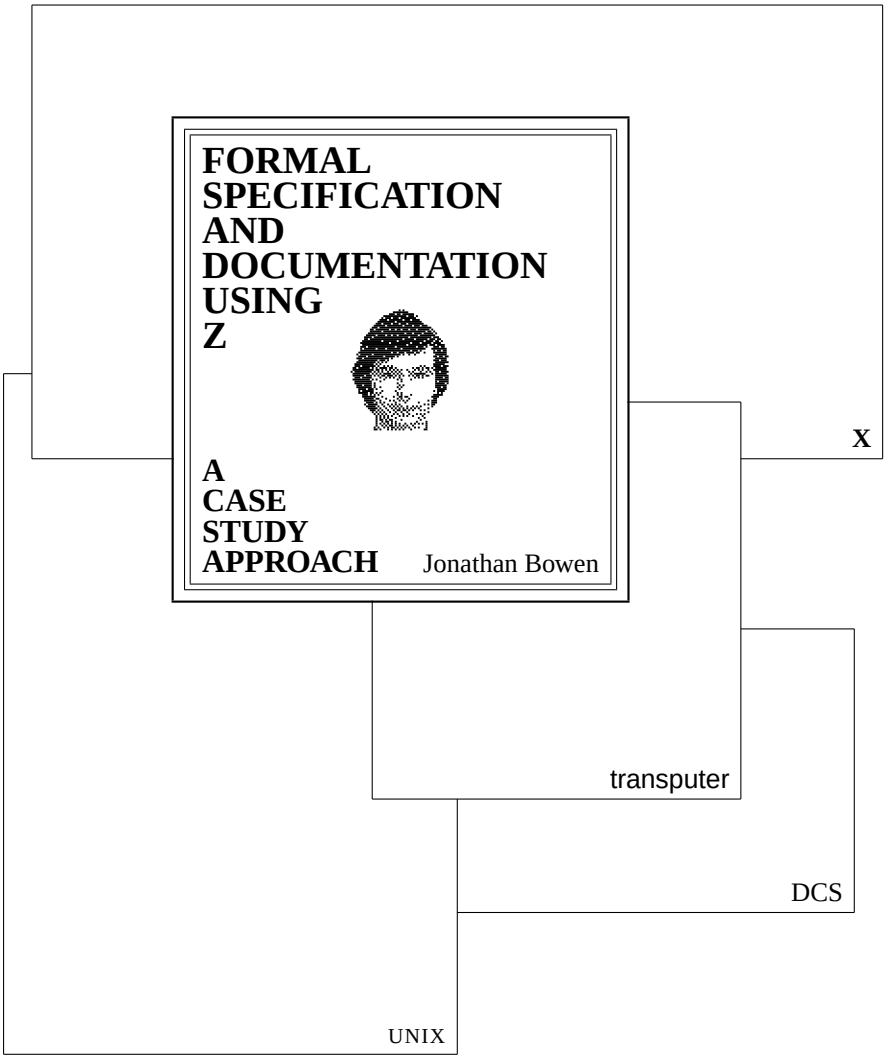
Formal Specification and Documentation using Z:

A Case Study Approach

Jonathan Bowen

Revised 2003


FORMAL
SPECIFICATION
AND
DOCUMENTATION
USING
Z
A
CASE
STUDY
APPROACH
Jonathan Bowen
X
transputer
DCS
UNIX

documentation All material that serves primarily to describe a system and make it more understandable, rather than to contribute in some way to the actual operation of the system. ...

formal specification **1.** A specification written and approved in accordance with established standards.
2. A specification written in a formal notation, such as VDM or Z.

Z A formal notation based on set algebra and predicate calculus for the specification of computing systems. It was developed at the Programming Research Group, Oxford University. Z specifications have a modular structure. ...

Dictionary of Computing [221]

To Jane, Alice and Emma

Contents

Foreword

The formal methods community has, in writing about the use of discrete mathematics for system specification, committed a number of serious errors. The main one is to concentrate on problems which are too small, for example it has elevated the stack to a level of importance not dreamt of by its inventors. While there is a good reason for using small examples at the beginning of a book or a tutorial, the need becomes progressively less important as one progresses towards teaching students and industrial staff topics such as structuring and modelling. Too many books have given up the fight after presenting small examples and have, I believe, contributed greatly to the lack of take-up of this technology. Staff and students who have read introductory materials on formal methods such as Z and VDM have had their hopes raised by small examples which have given the impression that formal specification is merely the writing down of some simple mathematical statements which define the behaviour of a system. What small examples do is to hide one of the most difficult tasks of specification: the process of selecting an adequate model.

Jonathan Bowen is a formal methods researcher who I have a great deal of respect for. Almost all his work has concentrated on the application of this technology to real-life problems – not just stacks and queues. His book teaches through the medium of case studies which are realistic but not too large that they overwhelm the reader. They range from the specification of the Transputer instruction set to that of a tool for formatting free text. All the case studies contain excellent examples of the power of Z: its ability to structure large specifications into chunks which can be read, validated and developed in relative isolation.

The formal methods community still have a long way to go in convincing many industrialists of the power of discrete mathematics; I would regard this book as a major contribution to doing so.

Darrel Ince
The Open University

Preface

Formal methods are becoming more accepted in both academia and industry as one possible way in which to help improve the quality of both software and hardware systems. It should be remembered however that they are not a panacea, but rather one more weapon in the armoury against making design mistakes. To quote from Prof. Tony Hoare:

> *Of course, there is no fool-proof methodology or magic formula that will ensure a good, efficient, or even feasible design. For that, the designer needs experience, insight, flair, judgement, invention. Formal methods can only stimulate, guide, and discipline our human inspiration, clarify design alternatives, assist in exploring their consequences, formalize and communicate design decisions, and help to ensure that they are correctly carried out.*
>
> *C.A.R. Hoare*, 1988

Thus we should not expect too much from formal methods, but rather use them to advantage where appropriate.

Even within the formal methods community, there are many camps: for example, those that believe that a formally correct system must be proved correct mechanically, one small step at a time, and those who use the term *formal* to mean *mathematical*, using high-level pencil-and-paper style proofs to verify a design is 'correct' with respect to its specification. Sometimes the latter method is known as 'rigorous' to differentiate it from the former; and of course there are positions between these two extremes.

Even if a system *is* proved correct, there are still many assumptions which *may* be invalid. The specification must be 'obviously right.' There is no way that this can be formally verified to be what is wanted. It must be simple enough to be understandable and should be acceptable to both the designer and the customer.

This book presents an even more pragmatic view of the use of formal methods than that held by some academics: that is that formal specification alone can still be beneficial (and is much more cost effective in general) than attempting proofs in many cases. While the cost of proving a system correct may be justified in safety-critical systems where lives are at risk, many systems are less critical, but could still benefit from formalization earlier on in the design process than is normally the case in much industrial practice.

Ultimately the computer system will be communicating with the outside world. In a control system, we will probably be dealing with physical laws, continuous math-

ematics (e.g., differential equations), etc. This will have to be converted into digital values and approximations will have to be made. In many cases, a Human-Computer Interface will be involved. Great engineering skill will be needed to ensure that any assumptions made are correct and will not invalidate any formally verified design. It is very important to apportion responsibility between the engineers associated with each design task. Mutually agreed interfaces must be drawn up. Ideally these should be formalized to reduce the risk of ambiguity and misunderstanding on each side of the interfaces.

This book presents the use of one notation in the accumulation of available mathematical techniques to help ensure the correctness of computer-based systems, namely the Z notation (pronounced 'zed'), intended for the specification of such systems. The formal notation Z is based on set theory and predicate calculus, and has been developed at the Oxford University Computing Laboratory since the late 1970's.

The use of a formal notation early on in the design process helps to remove many errors that would not otherwise be discovered until a later stage. The book includes specification of a number of digital systems in a variety of areas to help demonstrate the scope of the notation. Most of the specifications are of real systems that have been built, either commercially or experimentally. It is hoped that the variety of examples in this book will encourage more developers to attempt to specify their systems in a more formal manner before they attempt the development or programming stage.

In Part I, the first two chapters give an introduction to formal specification, using Z in particular, and also to the issues concerning the practical take-up and use of formal methods in industry. Chapter 2 gives an overview of some industrial issues, for those contemplating the use of formal methods as part of the software development process. Some guidelines to help with successful use are given. Finally a brief tutorial is given in Chapter 3, which introduces Z for those who have not seen the notation before, but who wish to tackle the rest of the book. However, it should be noted that this is not a substitute for a fuller treatment, which if required should be sought from one of the numerous Z textbooks now available.

Z has been designed to be read by (suitably trained) humans, rather than by computers, and as such may be included in manuals documenting computer-based systems. Part II gives some examples, using network services designed and built at Oxford University. Two types of manual have been developed, one of the user of a service, giving an idealized external abstract view, and one for potential implementors, giving more details of the suggested internal structure of the service.

In Part III, Chapter 6 details the specification of a text formatting tool designed for using under the UNIX operating system. The structure of UNIX files is discussed in this context. A specification of a mouse-based input system for UNIX workstations is also presented in Chapter 7.

Although Z has mainly been applied to software systems, it is also applicable to hardware. In Part IV, a number of aspects important in the specification of machine instruction sets are discussed. Chapter 8 formally defines some concepts which are useful in the specification of any microprocessor. Building of this, a part of a specific instruction set, namely that of the Transputer, is then presented in Chapter 9.

Part V details some graphical concepts. Chapter 10 introduces general concepts useful for specifying pixel maps and window systems. Chapter 11 defines the raster-op function which is fundamental to many graphics operations.

Window systems are now one of the most popular interfaces for computers. Part VI builds on the ideas presented in Part V and gives details of three window systems, including the highly successful X Window System. Chapter 15 remarks on experience gained by formally specifying the three window systems and other case studies.

Appendix A gives some indications on how to obtain further up-to-date information on Z. A glossary of the Z notation may be found in Appendix B. A literature guide in Appendix C together with a substantial bibliography at the end of the book are included to allow readers to follow up on another aspect of Z and formal methods that are of special interest. Finally an index, particularly of names of definitions in the specifications presented in the book, will aid the reader in navigating the text, especially the formal parts.

It is hoped that the specifications presented here will help students and industrial practitioners alike to produce better specifications of their designs, be they large or small. Even if no proofs or refinement of a system are attempted, mere formalization early on in the design process will help to clarify a designer's thoughts (especially when undertaken as part of a team) and remove many errors before they become implemented, and therefore much more difficult and expensive to rectify.

For further on-line information related to this book, held as part of the distributed World Wide Web (WWW) Virtual Library, the reader is referred to the following URL (Uniform Resource Locator):

`http://www.afm.sbu.ac.uk/zbook/`

J.P.B.
June 1995

I

Introduction

In Chapter 1, the use of the Z notation for the formal specification of computer-based systems is introduced. Chapter 2 considers industrial concerns in the application of formal methods such as the use of Z. Finally a brief tutorial introduction to the formal notation of Z is given in Chapter 3.

Chapter 1

Formal Specification using Z

This chapter provides an introduction to formal methods*, in general, and* formal specification *in particular: what they are, and how and why they should be used, with an emphasis on the Z notation. It provides some motivation for the use of formal specification, a brief introduction to some example applications as presented in more detail in the rest of the book, and some conclusions on the suitability or otherwise for the use of Z for system specification. The chapter is informal in nature and suitable for those who may not wish to read the later more detailed case study chapters.*

1.1 Introduction

Many design and documentation methods make use of informal techniques. For example, natural language and diagrams are often used alone to describe computer systems and software. A more formal approach can result in a simpler design and more thorough documentation. This book presents a general specification language, Z ('zed'), based on set theory and developed at the Oxford University Computing Laboratory (OUCL), as a possible solution to this problem. The notation is useful (once it has been learned) to organize the thoughts and aid the communication of ideas within a design team. It is also readable enough to be used as a documentation tool in a manual. Of course, natural language should also be included to give an informal description of the system and to relate the mathematical description to the real world.

A major advantage of a formal notation is that it is precise and unambiguous and thus the formal notation always provides the definitive description in the case of any misunderstanding. A number of examples are discussed, including network services, software for UNIX, microprocessor instruction sets, computer graphics, and window systems. Full formal descriptions of these in Z are included in the book in later chapters.

This chapter is split into two main parts. The first half deals with the nature of formal specification and why it should be used. Additionally, a brief introduction to Z and how it is used is also presented in general terms, without covering the notation itself. The second half of the chapter deals with the experience gained using Z for the design and documentation of network services and during some case studies of existing systems. Finally some conclusions are drawn about the advantages and disadvantages of using a formal approach.

1.2 Formal Specification

A formal specification is simply a description of a system using a mathematical notation. The advantage of using mathematics is that it is precise, unlike the more ambiguous natural language and diagrams which are often used for specifications. The disadvantage is the barrier of the notation. More people understand natural language than mathematics. The specification language must first be learned, and then experience in its use needs to be gained before its full benefits can be attained.

A specification language may be used as a design tool and, if the notation is readable enough, as a documentation tool. The actual process of designing a system may be undertaken using a formal notation to communicate ideas between members of a design team. Once the design has been finished, it can then form the basis for a manual describing the system.

Note that in the context of this book, the initial 'design' is considered to be the interface specification of the system with the outside world and the 'implementation' is considered to be the *refinement* of this design into a working system.

1.2.1 The Z notation

Z has been developed at Oxford University since the late 1970's by members of the Programming Research Group (PRG) within the Computing Laboratory [203, 336, 376, 381]. It is a typed language based on set theory and first order predicate logic. There is nothing very unusual about the mathematics employed, although a few operators have been added as experience has been gained in its use.

The problem with using mathematics alone is that large specifications very quickly become unmanageable and unreadable. Hence as well as the basic *mathematical notation*, Z includes a *schema notation* to aid the structuring of specifications. This provides the framework for a textual combination of sections of mathematics (known as *schemas*) using schema operators. Many of these match equivalent operators in the mathematical notation.

As well as the formal text, a Z specification should contain English (or some other natural language) to explain the mathematical description. Ideally, the informal description should remain readable even if the formal sections are removed from the document. However, if there is a conflict between the two descriptions, the mathematics is the final arbiter since it provides a more precise specification.

The idea of an abstract Z specification is to describe *what* a system does rather than *how* it does it. Imperative programming languages are specifications, but these concentrate on *how* the result is to be achieved. Functional programming languages are more like specification languages since these describe *what* result is required. However they are designed to be *executable*. Z can be used in a functional style. However it is possible (and sometimes desirable) to write non-deterministic specifications in Z. This means the exact execution of the specification cannot be determined. The Z notation is designed to be *expressive* and *understandable* (by humans) rather than *executable* (by computers).

Some specification languages *are* designed to be executable (although very inefficiently) so that rapid prototyping of the system is possible. However in such specifications, the designers often have to think about making the specification executable in

a feasible amount of time, possibly to the detriment of the design. Even though a Z specification is not in general executable by computer, by passing it round members of a design team it may be mentally executed and checked far more reliably than an equivalent informal specification.

Note that Z is a formal specification *notation* rather than a formal *method* although the term *method* is sometimes used rather loosely in this context.

1.2.2 Why use formal specification?

As previously mentioned, a formal specification is precise. This means that even if such a specification is wrong (i.e., not what the customer wanted), it is easier to tell where it is wrong and correct it. Since an informal specification is often ambiguous, it is more difficult to detect errors and subsequently put them right.

Using a formal notation increases the understanding of the operation of a system, especially early in a design. It helps to organize the thoughts of a designer, making clearer, simpler designs possible. Additionally, it is possible to formally reason about a system by stating and proving theorems about it. These provide a check that the system will behave as expected by the designer.

The use of formal methods can help to explore design choices. Such methods aid the design team in thinking about the operation of the system before its implementation. Missing parts of an incomplete specification become more obvious. The remaining parts of a design can be identified and alternative possibilities considered. In particular, error conditions can be checked by calculating the *precondition* of an operation, and then dealing with the errors to ensure that the precondition of the complete operation is true (i.e., that all possible error conditions have been covered). When using informal methods, it is easy to gloss over such details until the implementation stage.

The likelyhood of errors in a design is reduced. Errors may be pinpointed more easily as a result of the points above. The number of times round the design–implementation–testing cycle should be reduced since more errors will be found and corrected at the design stage.

The quality of documentation of the system can be improved. By using the formal design as a basis for the manual for a system, it is likely that less information will be left out. The final document should include a prose description relating the formal text to the real world.

Finally, and most importantly in industry, the overall cost should be lowered. Errors corrected at the design stage can be up to two orders of magnitude cheaper to correct than if they are found later. The initial barrier to using formal methods is the notation, which may contain unfamiliar symbols, and will require designers to attend training courses. However, in general the notation is no worse than learning a new style of programming language (for example, a functional language if the trainee is used to imperative programming).

1.2.3 How is Z used?

A Z specification may be written in a variety of styles (e.g., a functional style, as mentioned previously). However, it has been found convenient to use a state or model

based approach in many cases. A system may be considered to be modelled as an abstract state and a sequence of operations on this state.

First some basic sets (e.g., file identifiers) may be introduced. At this stage, it is not necessary to elaborate on the description. More precise details which relate to the implementation rather than the abstract specification may be left till later. It may also be useful to introduce some extra operators for a particular specification to make it more readable. These may be defined as the design progresses.

Next an *abstract state* is defined in terms of sets, relations, functions, sequences, etc. This should not be influenced by implementation considerations, but rather should be designed to make the specification as understandable as possible to the reader. This may well be modified during the design to make the specification of operations on the system more clear. Extra components which are redundant in that they are related to other state components may be included if this increases the overall clarity of the specification. The aim is not necessarily to make the formal description as short as possible, but rather to make it as understandable as possible.

An *initial state* (i.e., the state after initialization, at the start of the program, power-up, or whatever) should be specified. This is defined in terms of the abstract state and some extra predicates defining the initial conditions of the system.

Operations on the system will cause a change of state. There will be a *before* state and an *after* state. There may be invariants which relate the before and after states for all operations on the system. These may be included as predicates in a schema defining a general change of state of the system. Sometimes the after state will be the same as the before state (e.g., for status operations). Also a group of operations may only affect a particular part of the state. It is convenient to define schemas which partially specify such cases. This information may then be included in subsequent definitions of operations. This avoids having to cover common details more than once.

For each operation, a number of predicates will specify exactly what it is required to do. Inputs and outputs may also be included. Other temporary state components can be added if this is convenient to aid the clarity of the specification. An operation may be non-deterministic – i.e., there may be more than one possible outcome for the operation. For example, the system could provide a file identifier from a pool of available identifiers. The designer may not care which identifier is chosen. In such cases, it is left to the implementor to select the most convenient choice for a particular implementation.

Operations are considered to be *atomic* (i.e., one operation cannot break in while another is in progress). At the outermost level of the specification, the system is considered to be modelled by the initial state followed by an arbitrary sequence of legal operations. If any of the operations include preconditions, it is up to the implementor to ensure that the operation can only be executed when these are satisfied. If the preconditions for all the operations are true then a completely arbitrary sequence of operations may occur.

Once a design has been formulated, it is useful to state and prove theorems about the system. This helps to verify the design and check for mistakes. (For example, we could check that the creation followed by the deletion of a file leaves the state unchanged.) This process can be tedious, particularly if done completely by hand, but it is very worthwhile to reduce errors and gain understanding about the operation of the system before it is implemented.

Finally it is possible to refine an *abstract* design towards the *concrete* implementation by a series of state and operation refinement steps. For example, a set in an abstract state may not be immediately, or efficiently, implementable in a particular programming language. It could be implemented as an array, hash table, binary tree or other convenient data structure. Each refinement step is related to the previous one by a mathematical relation. There are a number of rules or *proof obligations* governing valid refinement steps. This refinement process will also make any non-determinism in the abstract specification deterministic in the final implementation by making implementation-dependent design choices.

1.3 Case Studies

Besides the work on the theoretical underpinning of Z, many case studies using the notation have also been undertaken to ensure its applicability in a practical environment. This section gives an overview of the case studies presented later in the book.

1.3.1 Network services

The Distributed Computing Software (DCS) project at the Oxford University Computing Laboratory designed a number of network services using the Z language. The results have been documented in several monographs [55, 56, 172]. The designs have formed the basis for manuals for each of the services. Two different types of manual have been produced. *User Manuals* have been designed to describe each service from the point of view of a client program using the service via Remote Procedure Calls (RPCs) over a network. In addition, *Implementor Manuals* have been produced for some services. These describe how the service may be implemented internally. Z is still used for this description, although it is assumed that an imperative sequential programming language will be used for the final implementation. As well as the User and Implementor Manuals, a *Common Service Framework* manual has been produced. This describes common parts of services to avoid repetition in individual manuals. Significant effort was expended in the presentation of the manuals to make them as readable as possible while still employing a formal notation.

A User Manual

A user will normally be interested in how a service reacts with the outside world rather than with the detailed inner workings of the service. Thus the manual can provide an abstract view of the service. This may be based directly on the original abstract design. Indeed, the initial design of the network services which have been produced during the project have consisted of a skeleton version of the User Manual. This has subsequently been tidied up and improved for the final version of the manual, thus greatly reducing the amount of time spent producing documentation.

Each User Manual is split into a number of sections. After a general introduction, the abstract state of the service is presented. Next common parameters shared by a number of service operations are covered (for example, all operations produce an output report). A section details the result of operations when an error occurs and the

reports which a returned. Each error condition is described as a Z schema which may subsequently be included by individual operations as required.

Each operation is normally allocated a page for its description, although occasionally this can spill over onto a second page for more complicated operations. The description is split into three sections. An *Abstract* section describes how the operation may appear as a procedure heading in some programming language. This includes all the explicit input and output parameters of the operation. A *Definition* section provides a formal description of the operation (as a Z schema) when it is performed successfully. Finally, a *Reports* section formally defines the specific errors which may occur when the operation is invoked by combining the schema in the previous section with error schemas. These three sections are accompanied by informal description as required.

The problem of accounting has also been addressed. This is often of secondary interest to a user, so it is included in a separate section. The charge for each operation (which may vary depending on the amount of data transferred, for example) is formally defined in a single 'tariff' schema. Finally all the operations and the tariff schema are combined, together with features from the Common Service Framework (see later) as desired to produce a complete specification of the service.

An Implementor Manual

Unlike a User Manual, an Implementor Manual *does* need to concentrate on the internal operation of a service. Thus a more concrete description of the service must be presented. When an Implementor Manual is produced, a number of design decisions must be taken. In the Implementor Manuals produced by the DCS project, it has been assumed that the service will be implemented using a sequential imperative programming language. (In fact, Modula-2 has been used for the actual implementations which have been produced.) However the manuals have still used Z rather than some pseudo-code to describe the operations. A small number of extra schema operators have been defined to allow descriptions of iteration, etc.

The outline for the Implementor Manuals is similar to that for the User Manuals. However a concrete state and concrete operations are presented, together with sub-operations and sub-states for subsystems as required by a particular service.

Of course, an Implementor Manual *should* be proved correct with respect to the corresponding User Manual. This has only been done for a simple service. Even for a modestly large service, this becomes intractable relately quickly if the process is undertaken by hand. Hopefully this may be alleviated by machine assistance in the future. In any case, the Implementor Manuals have been designed to convey the design to a programmer, rather than to aid a proof of correctness by defining its relationship with the User Manual (although this relationship is included formally in the Implementor Manual).

The Common Service Framework

Some parts of a network service service will inevitably be the same or similar to parts of all or a number of other services. In addition the general outline for the description of an individual service tends to follow a common pattern. Hence it is convenient to

group such aspects of the services in a separate document for use in the description of each specific service.

The Common Service Framework covers such features. First an example of a generalized service is presented, including all common features which may be used by a particular service. Then a number of common subsystems are formally described. These include extra operations to deal with concerns like time, accounting, statistics and access control. Any combination of these subsystems may be included in a given service. This will increase the number of operations which may be performed on that service.

Next, it is shown how all the services in the distributed system may be formally combined to produce a specification of the complete system. Network attributes, including authentication, and client attributes (e.g., identification) are also covered. Finally a summary of the common sets and data types used by the services is given.

Part II provides some actual examples of network service manuals. Chapter 4 presents some more detailed motivation and a very simple service by way of example. Chapter 5 takes the form of a more substantial user manual.

1.3.2 Other case studies

As well as designing and documenting network services, a number of case studies of existing systems in real use have been undertaken. Parts of the systems under investigation were specified in Z to gain a greater understanding of their operation.

UNIX software

The UNIX [37] file system was used as one of the earliest examples of the specification of a real system, demonstrating the structuring feature know as the schema calculus that is provided as part of Z to enable large specifications to be tackled [298]. Part III of this book provides further examples of more detailed software that has been implemented under UNIX. Chapter 6 presents a text formatting tool, useful for justifying ASCII text in a file [44]. A matching UNIX manual page is provided for comparison by the reader. Chapter 7 gives a specification for a library of C routines that implement an event-based input system for UNIX workstations [80].

Instruction sets

Z is not necessarily restricted to the specification of software-based systems. Any system which may be viewed as an abstract state on which a number of operations may be performed can be conveniently specified in Z. For example, Z has proved particularly good for specifying instruction sets. The Motorola 6800 8-bit microprocessor instruction set has been completely specified as an exercise [39, 38]. Additionally, large parts of the Inmos (now SGS-Thomson) Transputer [224] and Motorola 68000 16/32-bit microprocessor instruction sets have also been specified in Z [45, 149, 350]. Z scales up to these larger instruction sets with few problems, mainly because of the schema notation.

Part IV of the book gives an introduction to the formal specification of instruction sets in Z. Chapter 8 presents some general concepts concerning operations on micro-

processor words, consisting of fixed length sequences of bits. Next, Chapter 9 gives a portion of a real microprocessor instruction set, namely that of the Transputer.

Graphics and window systems

As a case study, a number of existing window systems have been studied [43, 47]. Originally it had been intended to compare parts of a number of distributed systems using Z. However, the authors of potential systems for investigation could only supply academic papers (not enough information) or the source code (too much information). What was required was some form of informal documentation for the system. Because window systems are used directly by users, there seems to be more readable documentation for such systems. Hence it was decided to attempt to produce a high-level specification for three window systems. The specifications could be used to contrast the systems and test the documentation for completeness.

The three systems chosen were X (a distributed window system from MIT, and now widely used), WM (part of Andrew, a distributed system developed at Carnegie-Mellon University) and the Blit, including *mux* (developed at Bell Laboratories, Murray Hill). In each case, omissions and ambiguities in the documentation were discovered by attempting to formalize the system. Where necessary, intelligent guesses were made about the actual operation. These were usually correct, but not always. Using such specifications, it would be a simple matter to update existing documentation, or even rewrite it from scratch.

Although Z has been developed as a design tool, it is also well suited for *post hoc* specifications of existing systems, and for detecting errors and anomalies in the documentation of such systems.

Window systems make use of basic graphical concepts such as *pixels* (short for 'picture elements'), and operations on these elements. Part V formalizes some of these ideas in Z. Chapter 10 defines the basic graphical concepts and Chapter 11 uses these to define 'raster-op' functions, useful for manipulating pixel maps. Part VI builds on these to specify parts of three existing window systems mentioned above, namely WM (Chapter 12), the Blit (Chapter 13), and X (Chapter 14).

1.4 Conclusions

Z is one of a number of specification languages which are being developed around the world. It is a general purpose specification language. For example, Z could be specified using itself [376, 79]. It could also be used to specify a more special purpose language such as CSP [215], which is designed to handle concurrency. Z itself is cumbersome for specifying parallel systems. Its use will produce a much longer specification than if CSP is used. Hence it is more convenient to use a language like CSP in such cases. Work has been undertaken to attempt to combine some of the features of CSP with Z [28, 239, 438].

Z has direct competitors. The most mature of these is probably VDM, advocated by Jones [233]. This is also based on set theory and predicates, and is similar to Z in a number of respects. Its differences include explicitly stating which components are read and written by an operation, and explicitly separating the preconditions, involving only the before state, and postconditions, also involving the after state. A more

advanced toolset is available for VDM, although the situation is being rectified for Z. The notation is arguably less readable that Z. It lacks an equivalent to the schema notation of Z which is so useful for aiding the structuring and readability of specifications. Subsequently, a more comprehensive set of notations, with tool support, has been produced in the form of RAISE [343].

Another approach to formal specification is that of algebraic specification (e.g., Larch [183] and OBJ [176]). These uses abstract data types in which the allowed operators on types are specified rather than the types themselves. This approach is theoretically very attractive but problems can occur in scaling up specifications for industrially sized examples.

1.4.1 Z – advantages

Z may be used to produce readable specifications. It has been designed to be read by humans rather than computers. Thus it can form the basis for documentation.

Large specifications are manageable in Z, using the schema notation for structuring. It is possible to produce hierarchical specifications. A part of a system may be specified in isolation, and then this may be put into a global context.

Z is liked by users. Many methods are foisted on designers in industry by managers attempting to improve efficiency. From the feedback which has been obtained, it seems that the use of Z is one of the few specification techniques which has not been received with reluctance by industrial users.

The notation is gradually gaining acceptance in industry, at least in the United Kingdom and is taught in many computer science curricula [314]. A regular Z User Meeting series (see page 229) has been established. Large companies (such as IBM and British Telecom) are particularly interested in investigating the use of Z in an industrial environment. The bigger the company, the more it has to gain by the use of formal methods. The largest project (known to the author) to use Z so far is the IBM Customer Information Control System (CICS) at Hursley Park in the UK (see page 241). This has produced about 2000 pages of Z specifications and designs from which around 37,000 lines of code (14%) have been developed having been fully specified and around 11,000 lines (4%) which were partially specified with an estimated 9% *decrease* in total development cost [247].

Courses are available both from academia and industry. Many introductory books have been published – see page 244 – and still more are likely to follow. An electronic newsgroup and associated Z FORUM [449] mailing list* is also distributed to those interested in Z, including open discussion and information on developments, tools, meetings, publications in a monthly message. An electronic Z archive is also maintained [448]. An international ISO standard is in preparation [79], under the auspices of ISO/IEC JTC1/SC22, which should help acceptance by industry. The ANSI X3J21 committee on Formal Description Techniques (FDTs), such as Z and VDM, is also involved.

* To join the distribution list, contact `zforum-request@comlab.ox.ac.uk` via electronic mail or read the `comp.specification.z` newsgroup.

1.4.2 Z – disadvantages

Z is not ideal for all problems. For example, as mentioned previously, dealing with concurrency is clumsy. However, Z is good for systems which may be modelled as a sequence of operations on an abstract state. This book aims to demonstrate a range of applications where Z *is* useful.

In general formal techniques require a significant amount of training effort and practical experience to be applied successfully. They require the dedication and commitment of *all* those involved, managers and engineers alike. In the past, management and software engineers have not received appropriate training for their use, although the situation is changing with regard to many university computer science courses, especially in the UK [314]. However, once trained, especially if done on the job, engineers can apply for more attractive posts elsewhere, which can be a very real deterrent for industry to train their employees.

The toolset for Z is still not very advanced by industrial standards. Perhaps the best type-checker available is the *f*UZZ system [380] which is intended for use with the widely available LaTeX document preparation system [251] and is compatible with the main Z reference manual in current use [381]. Some theorem proving support is now available (e.g., ProofPower [236] from ICL, based on HOL [178]) but is still not yet widely used. In general Z is still used for specification rather than proof in industry [22]. [326] provides some information on available tools.

1.4.3 General conclusions

Z can be used to succinctly specify real systems. The examples given in this book and other case studies undertaken at Oxford and elsewhere lend support to this assertion. The extensively reported IBM CICS work (see page 241) is probably the largest project to have have used Z. Z has also been used successfully in initial specification for the development of the microcode for the floating-point unit of the Inmos Transputer [24, 281, 282, 367, 368]. A formal notation is useful for the design of systems, allowing better understanding before implementation, and reducing the number of errors. This design can subsequently form the basis of a manual since the notation is readable [224].

Z can also help in refinement towards an implementation by mathematically relating the abstract and concrete states. Reasoning about the system is possible using mathematical logic. Tools are being developed for machine assistance with the checking of Z specifications [326]. Such tools will make the use of formal methods more feasible in an industrial environment. Formal refinement is not normally cost effective (or even tractable) for most software systems of an industrial scale [18, 105, 106]. However basic research in this area could help change this in the future. At some point during the development of an implementation, a change of notation will normally be necessary as a more imperative style is normally required [295, 299]. The related B-Method and its associated B-Tool [3, 4, 5] has proved to be successful in the development of systems on an industrial scale [91, 182] and an experimental tool for Z support has been developed using this [311].

This book only provides a brief introduction to the Z notation itself in Chapter 3; the subject is adequately covered in the references given in the bibliography for those

who wish to learn more about Z (e.g., [336] is recommended). The bibliography, together with the associated literature guide in Appendix C, provide a comprehensive and categorized list of references on Z, including other examples of significant systems specified in the Z notation which help to demonstrate that it can be advantageously applied to industrially sized problems.

Formal techniques such as Z are now sufficiently well established and supported for the software industry to gain significant benefits from their use. In practice this has only happened to a very limited extent so far despite a number of well publicized successful examples. In the future, the advantages are likely to be even greater and those that do not keep up with developments are likely to be left behind. Other more mature engineering disciplines make use of mathematics as a matter of course to describe, verify and test their products. It is time for all practising software engineers to learn to do likewise if computing is to come of age.

For further on-line information about the Z notation, held as part of the distributed World Wide Web (WWW) Virtual Library, the reader is referred to the following URL (Uniform Resource Locator):

`http://www.zuser.org/z/`

The next chapter addresses industrial concerns in particular when using formal methods for development of computer-based system, with some general guidelines on the application of formal methods in practice.

Chapter 2

Industrial Use of Formal Methods

Formal methods are propounded by many academics but eschewed by many industrial practitioners. Despite some successes, formal methods are still little used in industry at large, and are seen as esoteric by many managers. In order for the techniques to become widely used, the gap between theorists and practitioners must be bridged effectively. In particular, safety-critical systems, where there is a potential risk of injury or death if the system operates incorrectly, offer an application area where formal methods may be engaged usefully to the benefit of all. This chapter discusses some of the issues concerned with the general acceptance of formal methods and gives some guidance for their practical use. The chapter is informal and suitable for those without a mathematical knowledge of the formal methods involved.

2.1 Introduction

The software used in computers has become progressively more complex as the size of computers has increased and their price has decreased [335]. Unfortunately software development techniques have not kept pace with the rate of software production and improvements in hardware. Errors in software are renowned and software manufacturers have in general issued their products with outrageous disclaimers that would not be acceptable in any other more established industrial engineering sector [170].

It has been suggested that formal methods are a possible solution to help reduce errors in software. Sceptics claim that the methods are infeasible for any realistically sized problem. Sensible proponents recommend that they should be applied selectively where they can be used to advantage. More controversially, it has been claimed that formal methods, despite their apparent added complexity in the design process, can actually *reduce* the overall cost of software. The reasoning is that while the cost of the specification and design of the software is increased, this is a small part of the total cost, and time spent in testing and maintenance may be considerably reduced. If formal methods are used, many more errors should be eliminated earlier in the design process and subsequent changes should be easier because the software is better documented and understood.

2.2 Technology Transfer Problems

The following extract from the BBC television programme *Arena* broadcast in the UK during October 1990 graphically illustrates the publicly demonstrated gap between much of the computing and electronics industry, and the formal methods community, in the context of safety-critical systems where human lives may be at stake; these, arguably, have the most potential benefit to gain from the use of formal methods [26].

> Narrator: [On Formal Methods] '*... this concentration on a relatively immature science has been criticized as impractical.*' Phil Bennett, IEE: '*Well we do face the problem today that we are putting in ever increasing numbers of these systems which we need to assess. The engineers have to use what tools are available to them today and tools which they understand. Unfortunately the mathematical base of formal methods is such that most engineers that are in safety-critical systems do not have the familiarity to make full benefit of them.*'
>
> Martyn Thomas, Chairman, Praxis plc: *'If you can't write down a mathematical description of the behaviour of the system you are designing then you don't understand it. If the mathematics is not advanced enough to support your ability to write it down, what it actually means is that there is no mechanism whereby you can write down precisely that behaviour. If that is the case, what are you doing entrusting people's lives to that system because by definition you don't understand how it's going to behave under all circumstances? ... The fact that we can build over-complex safety-critical systems is no excuse for doing so.*'

This repartee is typical not only of the substantial technology transfer problems, but also of the debate between the 'reformist' (pro 'real world') and the 'radical' (pro formal methods) camps in software engineering [404].

Formal methods have a reputation for being oversold by their proponents. To quote Prof. C.A.R. Hoare, as reported in *Computing* [327]:

> *Advocates of formal methods must preserve, refine and teach the valuable knowledge we have gained for assisting some key areas of software engineering. But we should be more modest in our aims and very much more modest in our claims than we have sometimes been in the past.*

This book aims to impart some of that knowledge, but readers should bear the above quotation in mind at all times, despite the sometimes enthusiastic nature of the material in this volume. Formal methods are *not* a panacea, but another technique available in the battle against the introduction of errors in computer systems.

2.2.1 Misconceptions and barriers

Unfortunately formal methods is sometimes misunderstood and relevant terms are even misused in industry (at least, in the eyes of the formal methods community). For example, the following two alternative definitions for *formal specification* are taken from a glossary issued by the IEEE [220]:

1. *A specification written and approved in accordance with established standards.*
2. *A specification written in a formal notation, often for use in proof of correctness.*

The meaning of 'formal notation' is not elaborated further in the glossary, although 'proof of correctness' is defined in general terms.

Some confuse formal methods with 'structured methods'. While research is underway to link the two and provide a formal basis to structured methods (e.g., see [241]), the two communities have, at least until now, been sharply divided apart from a few notable exceptions. Many so-called formal 'methods' have concentrated on notations and/or tools and have not addressed how they should be slotted into existing industrial best practice. On the other hand, structured methods provide techniques for developing software from requirements to code, normally using diagrams to document the design. While the data structures are often well defined (and easily formalized), the relationships between these structures are often left more hazy and are only defined using informal text (natural language).

Industry has been understandably reluctant to use formal methods while they have been largely untried in practice. There are many methods being touted around the market place and formal methods are just one form of them. When trying out any of these new techniques for the first time, the cost of failure could be prohibitive and the initial cost of training is likely to be very high. For formal methods in particular, few engineers, programmers and managers currently have the skills to apply the techniques beneficially (although many have the ability).

Unfortunately, software adds so much complexity to a system that with today's formal techniques and mechanical tools, it is intractable to analyze all but the simplest systems exhaustively. In addition, the normal concept of tolerance in engineering cannot be applied to software. Merely changing one bit in the object code of a program may have a catastrophic and unpredictable effect. However, software provides such versatility that it is the only viable means of developing many products.

Formal methods have been a topic of research for many years in the theoretical computer science community. However they are still a relatively novel concept for most people in the computing industry. While industrial research laboratories are investigating formal methods, there are not many examples of the use of formal methods in real commercial projects. Even in companies where formal methods are used, it is normally only to a limited extent and is often resisted (at least initially) by engineers, programmers and managers. [184] is an excellent article that helps to dispel some of the unfounded notions and beliefs about formal methods (see Section 2.5).

Up until quite recently it has widely been considered infeasible to use formal techniques to verify software in an industrial setting. Now that a number of case studies and examples of real use are available, formal methods are becoming more acceptable in some industrial circles [182, 212, 218]. Some of the most notable of these are mentioned in [73], particularly those where a quantitative indication of the benefits gained have been published.

2.2.2 Modes of use

Formal methods may be characterized at a number of levels of usage and these provide different levels of assurance for the resulting software that is developed. This is sometimes misunderstood by antagonists (and even enthusiasts) who assume that using formal methods means that *everything* has to be proved correct. In fact much current industrial use of formal methods involves no, or minimal, proofs [22].

At a basic level, formal methods may simply be used for a high-level specification of the system to be designed (e.g., using the Z notation). The next level of usage is

to apply formal methods to the development process (e.g., VDM [233]), using a set of rules or a design calculus that allows stepwise refinement of the operations and data structures in the specification to an efficiently executable program. At the most rigorous level, the whole process of proof may be mechanized. Hand proofs or design inevitably lead to human errors occurring for all but the simplest systems.

Mechanical theorem provers such as HOL [178] and the Boyer-Moore system have been used to verify significant implementations, but need to be operated by people with skills that very few engineers possess today. Such tools are difficult to use, even for experts, and great improvements will need to be made in the usability of these tools before they can be widely accepted in the computing industry. Tools are now becoming commercially available (e.g., the B-Tool and Lambda) but there is still little interest in industry at large. Eventually commercial pressures should improve these and other similar tools which up until now have mainly been used in research environments. In particular, the user interface and the control of proofs using strategies or 'tactics', while improving, are areas that require considerable further research and development effort.

2.2.3 Cost considerations

The prerequisite for industrial uptake of formal techniques is a formalism which can adequately deal with the pertinent aspects of computer-based systems. However, the existence of such a formalism is not sufficient; the relevant technology must also be able to address the problems of the industry by integrating with currently used techniques [422], and must do so in a way that is commercially advantageous.

It should be noted that despite the mathematical basis of formal methods, errors are still possible because of the fallibility of humans and, for mechanical verification, computers. However formal methods have been demonstrated to reduce errors (and even costs and time to market) if used appropriately [218, 281]. In general though, formal *development* does increase costs [71, 72].

Even if the use of formal methods incurs higher development costs, this is unlikely to be the predominant factor. The critical considerations to a greater or lesser extent (depending on market growth rates) are development speed and final product cost. Is it, therefore, evident that formal methods can deliver cheaper products rapidly? Given the current technology, the over-zealous use of formal methods can easily slow down rather than speed up the process, although the reverse is also possible if formal methods are used selectively. It is however the case that in specialized markets such as the high integrity sector, where the correctness of the software and overall system safety are very important, other factors such as product quality may be the overriding concern. A further consideration must be whether formal methods can enhance product quality, and even company prestige.

2.3 Industrial-scale Usage

As has previously been mentioned, the take up of formal methods is not yet great in industry, but their use has normally been successful when they have been applied appropriately [403]. Some companies have managed to specialize in providing formal methods expertise (e.g., CLInc in the US, ORA in Canada and Praxis in the UK),

although such examples are exceptional. A recent international investigation of the use of formal methods in industry [106, 105] provides a view of the current situation by comparing some significant projects which have made serious use of such techniques. [18] is another survey worthy of mention, which suggests that Z is one of the leading formal method in use within industry.

[73] provides a survey of selected projects and companies that have used formal methods in the design of safety-critical systems and [102] gives an overall view of this industrial sector in the UK. In critical systems, reliability and safety are paramount to reduce the risk of loss of life or injury. Extra cost involved in the use of formal methods is acceptable because of the potential savings later, and the use of mechanization for formal proofs may be worthwhile for critical sections of the software. In other cases, the total cost and time to market is of highest importance. For such projects, formal methods should be used more selectively, perhaps only using informal proofs or just specification alone. *Formal documentation* (i.e., formal specification with adequate accompanying informal explanation) of key components may provide significant benefits to the development of many industrial software-based systems without excessive and sometimes demonstrably decreased overall cost (e.g., see [212, 218]).

2.3.1 Application areas and techniques

Formal methods are applicable in a wide variety of contexts to both software and hardware [213]. They are useful at a number of levels of abstraction in the development process from requirements capture, through to specification, design, coding, compilation and the underlying digital hardware itself. Some research projects have been investigating the formal relationships between these different levels [54, 51], which are all important to avoid errors.

The *Cleanroom* approach is a technique that could easily incorporate the use of existing formal notations to produce highly reliable software by means of non execution-based program development [145]. This technique has been applied very successfully using rigorous software development techniques with a proven track record of reducing errors by a significant factor, in both safety-critical and non-critical applications. The programs are developed separately using informal (often just mental) proofs before they are certified (rather than tested). If too many errors are found, the process rather than the program must be changed. The pragmatic view is that real programs are too large to be formally proved correct, so they must be written correctly in the first place! The possibility of combining Cleanroom techniques and formal methods have been investigated [323], although with inconclusive results. Further attempts could be worthwhile.

There is considerable research into object-oriented extensions of existing formal notations such as Z and VDM [387, 388] and the subject is under active discussion in both communities. Object-oriented techniques have had considerable success in their take-up by industry, and such research may eventually lead to a practical method combining the two techniques. However there are currently a large number of different dialects and some rationalization needs to occur before industry is likely to embrace any of the notations to a large degree.

An important but often neglected part of a designed system is its *documentation*, particularly if subsequent changes are made. Formalizing the documentation leads to

less ambiguity and thus less likelyhood of errors [41]. Formal specification alone has proved beneficial in practice in many cases [22]. Such use allows the possibility of formal development subsequently as experience is gained.

The *Human-Computer Interface* (HCI) is an increasingly important component of most software-based systems. Errors often occur due to misunderstandings caused by poorly constructed interfaces [261]. Formalizing an HCI in a realistic and useful manner is a difficult task, but progress is being made in categorizing features of interfaces that may help to ensure their reliability in the future. There seems to be considerable scope for further research in this area, which also spans many other disparate disciplines, particularly with application to safety-critical systems where human errors can easily cause death and injury [192].

Security is an area related to safety-critical systems. Security applications have in some cases been very heavy users of formal methods. However, it is normally extremely difficult to obtain hard information on such projects because of the nature of the work. Thus there is comparatively little widely published literature on the practical application and experience of formal methods in this field, with a few exceptions (e.g., see [35]).

2.4 Motivation for Use

2.4.1 Standards

Up until relatively recently there have been few standards concerned specifically with formal notations and methods. Formal notations are eschewed in many software-related standards for describing *semantics*, although BNF-style descriptions are universally accepted for describing *syntax*. The case for the use of formal notations in standards is now mounting as formalisms become increasingly understood and accepted by the relevant readership [34]. Hopefully this will produce more precise and less ambiguous standards in the future, although there is still considerable debate on the subject and widely differing views across different countries [122]. Formal notations themselves have now reached the level of maturity that some of them are being standardized (e.g., LOTOS, VDM and Z) [48].

An important trigger for the exploitation of research into formal methods could be the interest of regulatory bodies or standardization committees (e.g., the *International Electrotechnical Commission*). Many emerging safety-related standards are at the discussion stage [414]. A major impetus has already been provided in the UK by promulgation of the Ministry of Defence (MoD) Interim Defence Standard 00-55 [291], which mandates the use of formal methods and languages with sound formal semantics.

It is important that standards should not be prescriptive, or that parts that are should be clearly separated and marked as such. Goals should be set and the onus should be on the software supplier that their methods achieve the required level of confidence. If particular methods are recommended or mandated, it is possible for the supplier to assume that the method will produce the desired results and blame the standards body if it does not. This reduces the responsibility and accountability of the supplier. Some guidance is worthwhile, but is likely to date quickly. As a result, it may be best to include it as a separate document or appendix so that it can be updated more

frequently to reflect the latest available techniques and best practice. For example, 00-55 includes a separate guidance section.

2.4.2 Legislation

Governmental legislation is likely to provide increasing motivation to apply appropriate techniques in the development of safety-critical systems. For example, a new piece of European Commission (EC) legislation, the Machine Safety Directive, came into effect on 1st January 1993 [121]. This encompasses software and if there is an error in the machine's logic that results in injury then a claim can be made under civil law against the supplier. If negligence can be proved during the product's design or manufacture then criminal proceedings may be taken against the director or manager in charge. There is a maximum penalty of three months in jail or a large fine [306]. Suppliers have to demonstrate that they are using best working practice. This could include, for example, the use of formal methods. However the explicit mention of software in [121] is very scant. Subsection 1.2.8 on *Software* in Annex B on p. 21 is so short that it can be quoted in full here: '*Interactive software between the operator and the command or control system of a machine must be user-friendly.*' Software correctness, reliability and risk are not covered as separate issues.

Care should be taken in not overstating the effectiveness of formal methods. In particular, the term *formal proof* has been used quite loosely sometimes, and this has even led to litigation in the law courts over the Viper microprocessor, although the case was ended before a court ruling was pronounced [270]. If extravagant claims are made, it is quite possible that a similar case could occur again. 00-55 differentiates between *formal proof* and *rigorous argument* (informal proof), preferring the former, but sometimes accepting the latter with a correspondingly lower level of design assurance. Definitions in such standards could affect court rulings in the future.

2.4.3 Education and certification

Most modern comprehensive standard textbooks on software engineering now include a section on formal methods. Many computing science courses, especially in Europe, are now including a significant portion of basic relevant mathematical training (e.g., discrete mathematics such as set theory and predicate logic). In this respect, education in the US seems to be lagging behind, although there are some notable exceptions (e.g., see [162]). It is particularly important that the techniques, once assimilated, are used in practice as part of an integrated course, but this has not always been the case in the past.

[340] discusses the accreditation of software engineers by professional institutions. It is suggested that training is as important as experience in that *both* are necessary. In addition, software engineers should be responsible for their mistakes if they occur through negligence rather than genuine error. Safety-critical software is identified as an area of utmost importance where such ideas should be applied first because of the possible gravity of errors if they do occur.

A major barrier to the acceptance of formal methods is that many engineers and programmers do not have the appropriate training to make use of them and many managers do not know when and how they can be applied. This is gradually be-

ing alleviated as the necessary mathematics is being taught increasingly in computing science curricula. In the past it has been necessary for companies to provide their own training or seek specialist help, although formal methods courses are now quite widely available from both industry and academia in some countries (e.g., for the UK, see [314]). It appears that Europe is leading the US and the rest of the world in this particular battle, and in the use of formal methods in general, so this may be a good sign for the long term development and reliability of software emanating from within Europe.

Some standards and draft standards are now recognizing the problems and recommending that appropriate personnel should be used, especially on safety-critical projects. There are suggestions that some sort of certification of developers should be introduced. This is still an active topic of discussion, but there are possible drawbacks as well as benefits by introducing such a 'closed shop' since suitably able and qualified engineers may be inappropriately excluded (and vice versa).

2.4.4 Bridging the gap

Technology transfer is often fraught with difficulties and is inevitably – and rightly – a lengthy process. Problems at any stage can lead to overall failure [85]. A technology such as formal methods should be well established before it is applied, especially in critical applications where safety is paramount. Awareness of the benefits of formal methods must be publicized to a wide selection of both technical and non-technical people, especially outside the formal methods community (e.g., as in [384]), and the possibilities and limitations of the techniques available must be well understood by the relevant personnel to avoid costly mistakes.

Unfortunately, the rapid advances and reduction in cost of computers in recent years has meant that time is not on our side. However, formal techniques are now sufficiently advanced that they should be considered for selective use in software development, provided the problems of education can be overcome. It is likely that there will be a skills shortage in this area for the foreseeable future and significant difficulties remain to be overcome [93].

Software standards, especially those concerning safety, are likely to provide a motivating force for the use of formal methods, and it is vital that sensible and realistic approaches are suggested in emerging and future standards. 00-55 [291] seems to provide such an example and is recommended as guidance for other forward-looking proposed standards in this area [48, 63].

2.5 Guidelines for Use

2.5.1 Some myths

In a classic paper, Anthony Hall presented *Seven Myths of Formal methods* [184]. These are briefly presented here:

Myth 1: *Formal Methods can guarantee that software is perfect.*

One should remember that *any* technique is fallible. Even if a correct mathematical proof *is* achieved, the assumption that the mathematics models reality correctly is still prone to error.

Myth 2: *They work by proving that programs are correct.*
It is not *necessary* to undertake proofs to gain benefit from the use of formal methods; indeed much if not most industrial use of formal methods does not involve proofs [22]. Major gains can be achieved just be formally *specifying* the system being designed since this process alone can expose flaws, and in a much more cost-effective manner. Proofs may be worthwhile in highly critical systems where the extra cost can be justified.

Myth 3: *Only highly critical systems benefit from their use.*
A range of formal methods have been applied to many types of system, some of greater, others of lesser criticality. The extent of and type of application will depend on the level of criticality, which is ultimately a case of engineering and financial judgement.

Myth 4: *They involve complex mathematics.*
The mathematic required (and desired!) for formal specification is of a level that could be taught at school. After all, a major goal of a specification is to be easily understandable, so using esoteric terminology is in nobody's interest. Unfortunately, although relatively simple, it is a fact that many software engineers have not received the requisite training in the past.

Myth 5: *They increase the cost of development.*
Proofs *do* increase the cost of development in general, but formal *specifications* do not if used appropriately. This is because they allow many errors to be discovered earlier on in the design process when they are still relatively cheap to correct.

Myth 6: *They are incomprehensible to clients.*
The mathematics may not be readable by an untrained client, but a formal specification helps produce a much clearer natural language description of the system as well. This should be presented to the client, giving a much less ambiguous description of the system than is often the case.

Myth 7: *Nobody uses them for real projects.*
There are now a number of examples of actual use of formal methods, with demonstrably beneficial results [213]. Two recommended examples which used Z, and both of which won UK Queen's Awards for Technological Achievement in 1990 and 1992, are the Inmos Transputer Floating Point Unit microcode design [281] and the IBM CICS Transaction Processing System [247].

Seven more myths are presented in [64, 68]. These may be summarized as follows:

Myth 8: *Formal methods delay the development process.*
Some projects using formal methods *have* been seriously delayed in the past, but this has been as much to do with the problem of introducing any new technique into the design process as to do with formal methods *per se*. The *over*-use of formal methods *does* delay the development process. Certainly full proofs are a time-consuming activity which may not be (indeed, normally *will* not be) worthwhile. However the use of formal *specification* (e.g., on the IBM CICS project) and even formal development with appropriate personnel and tools (e.g., for the Inmos Transputer Floating Point Unit microcode) – see Myth 7 – as part of the development process have been demonstrated to be worthwhile, giving measurable improvements in cost and time.

Myth 9: *They do not have tools.*

There are now some significant tools supporting formal methods, many of which have been put to serious industrial use. Large toolsets worthy of mention include the B-Tool [3], and the associated B-Toolkit from B-Core (UK) Limited, for the B-Method [4, 5]; the RAISE (Rigorous Approach to Industrial Software Engineering) development method, a more comprehensive successor to VDM, and its associated toolset available from CRI (Computer Resources International) in Denmark [343]; the VDM Toolbox from IFAD, Denmark. Some theorem provers, such as EVES (based on ZF – Zermelo-Fraenkel – set theory) [110], HOL (based on higher order logic) [178], LP (the Larch Prover, for algebraic specifications [183]), Nqthm (a successor to the Boyer-Moore theorem prover, from CLInc in Austin, Texas), OBJ [176] and PVS (a more recent Prototype Verification System from SRI in California, based on higher order logic), have been used for significant proofs. Some can provide support for Z (e.g., see [58, 352]). A number of tools for Z are listed in page 225, together with relevant contact information.

Myth 10: *Their use means forsaking traditional engineering design methods.*

Formal methods should *not* be used to replace the existing development process. Rather they should be slotted into the process in an appropriate and thoughtful manner. This can be a tricky issue which needs serious consideration by the project manager and all concerned. Considerable goodwill is required for this to be done in a smooth way. Method integration is an important issue, e.g., for structured and formal techniques [364]. One example is the combination of SSADM and Z to produce SAZ [274, 332, 334]. Formal methods can also be used effectly to augment an existing design process by providing extra feedback to correct errors early in the design process [17]. *Cleanroom* [145] is another approach which could be combined with formal methods. See also further information on method integration on page 240.

Myth 11: *They only apply to software.*

Formal methods are used for hardware development as well as software. The Inmos Transputer work mentioned in Myth 7 is one example [367]. Z has also been applied to microprocessor instruction sets (see [38, 39, 45] and Chapter 9), oscilloscopes [120], etc. For further examples, see page 241. Hardware/software co-design is a rapidly developing area and formal methods could be useful in clarifying this difficult area [217].

Myth 12: *They are not required.*

Increasingly, standards *will* mandate, or at least highly recommend, the use of formal methods for systems of the highest integrity, such as those that are safety-critical [48, 63]. See Section 2.4.1 (on page 20) for further information on standards.

Myth 13: *They are not supported.*

There are now many books on formal methods (including this one!), conferences, courses, etc. For pointers specifically concerned with Z, see Appendix A. A number of companies now specialize in formal methods (e.g., B-Core (UK) Limited, Computational Logic Inc., CRI, DST, Formal Systems (Europe) Limited, IFAD, Praxis). A range of formal methods tools are commercially marketed (e.g., the FDR model checker from Formal Systems, the LAMBDA toolset from Abstract

Hardware Limited, ProofPower by ICL, etc.). The literature guide in Appendix C may also be helpful.

Myth 14: *Formal methods people always use formal methods.*
While formal methods *can* be useful, they are not always appropriate. Even those well versed in the use of formal methods do not always used them. This can be particularly useful to communicate design ideas within a team. Thus in a design team of one for a small project they may not be worthwhile.

2.5.2 Some suggestions

This section provides some guidance for the use of formal methods. It summarizes an article entitled *Ten commandments of Formal Methods* [68], but should not be taken as 'gospel' despite the title! Rather it should be used to augment the reader's own experience.

1st commandment: *Thou shalt choose an appropriate notation.*
Z is appropriate if you wish to undertake formal specification as part of a design team. Other formal notations and methods have different strengths and weaknesses depending on what is required of the method in the development process. For example, the B-Method [5] is more suitable than Z if formal development with tool support is to be undertaken. Many factors will affect the selection of a notation, not least of which is the background and expertise of the team involved. Learning a new notation is a time-consuming process which should be avoided if an appropriate (enough) notation is already know by the team.

2nd commandment: *Thou shalt formalize but not over-formalize.*
This relates to Myth 14. In addition, getting the right level of abstraction is very important in a specification. This will affect its readability, the ease with which proofs can be undertaken, and the number of design choices left open for the developer. The level of abstraction should be as high as possible, but no higher; otherwise important information may be omitted.

3rd commandment: *Thou shalt estimate costs.*
This is perhaps one of the most difficult things to do for any software product, whatever the development approach. Unfortunately the estimation technique used in many other engineering disciplines break down for software. The complexity of the solution is very difficult to estimate before it has been undertaken. Long experience can help to give some insight, but estimates are still difficult. Formal methods may help to quantify the estimation, since a formal description of the problem is obtained early in the design process. However, the effort to produce the final implementation may be difficult to determine from the formal specification.

4th commandment: *Thou shalt have a formal methods guru on call.*
Formal methods *do* require significant mathematical ability and training. These are not beyond the level obtainable by the average engineer, but it is still worthwhile having an expert with several year's experience of formal methods available, at least on an easily accessible consultancy basis, especially if many of the design team are relatively new to formal methods. This could well avoid a great deal of unnecessarily wasted time and cost.

5th commandment: *Thou shalt not abandon thy traditional development methods.*
This relates to Myth 10. Formal methods should be used as an extra technique in the armoury available for the elimination of errors. Certainly they will not catch all the errors, so other techniques should also be used (e.g., see the 9th commandment below).

6th commandment: *Thou shalt document sufficiently.*
Much of this book is dedicated to this subject, hopefully demonstrating that a formal notation like Z can be used in a beneficial way for system documentation. Even if the formal specification is omitted from the final documentation, its production is likely to make the informal documentation clearer and less ambiguous.

7th commandment: *Thou shalt not compromise thy quality standards.*
Software quality standards such as ISO 9000 need to be met, whatever the development techniques that are used. Formal methods can help in this respect if applied sensibly, but the project manager should ensure that they do help rather than hinder in practice.

8th commandment: *Thou shalt not be dogmatic.*
Absolute correctness in the real world can never be achieved. Mathematical models can be verified with a good level of certainty, but these models might not correspond with reality correctly. When applying formal methods, the level of use should always be determined beforehand and monitored while in progress. A project manager should always be prepared to adjust the level of use if required.

9th commandment: *Thou shalt test, test, and test again.*
Formal methods will never replace testing; rather they will reduce the number of errors found through testing. Formal development and testing tend to avoid and discover different types of error, so the two are complementary to some extent.

10th commandment: *Thou shalt reuse.*
Formal specifications can be written in a reusable manner, with some thought. As an example, Z includes a 'toolkit' of definitions, defined in Z itself, which have proved to be useful for many different specifications. The core of the toolkit is accepted as standard by most people who use Z for specification. In this book, the Common Service Framework mentioned in Part II, the machine word definitions in Chapter 8, and the graphics definitions in Part V, could all be reused – indeed, *have* been reused – for other specifications.

The above 'commandments' will hopeful provide basic guidance in the use of formal methods in practice. For further details, see [68]. For a number of examples of the realistic application of formal methods, see [213].

2.6 Future Developments

To secure the successful future of formal methods, a number of developments are desirable. These include:

- *Taking an engineering approach to formal specification and verification.* Formal methods must be integrated smoothly into existing best industrial practice in a manner which causes as little disruption as possible [268].

- *Better tools.* Most formal methods tools so far have resulted from formal methods research projects, and associated spin-off companies, rather than mainstream tools developers. As a result, their usability, and sometimes robustness, can often leave a lot to be desired. Unfortunately the formal methods tools market is still fairly small and raising capital to invest in serious production quality tools may be difficult. Raising commercial venture capital is likely to be difficult because the banks will be more interested in the size of the market rather than the potential improvement in software quality!
- *Investment in technology transfer.* The transfer of technology like formal methods is a time consuming and costly business. The effects and benefits of formal methods are less palpable than some of the other more popular techniques that come and go with fashion. The investment in learning and using formal methods is large, but the returns in the long term can be commensurate with this. Most people who have made the investment have not regretted it afterwards, and would not go back to their old ways.
- *Unification and harmonization of engineering practices involved in building high integrity systems.* While the use of formal methods may seem to run perpendicular and even counter to some other concerns on software engineering projects, such friction should be minimized. It is important that all those involved, be it managers or engineers, and whether the personnel involved fully understand the techniques or not, at least understand the way the techniques slot into the overall framework. It can be galling to some managers that the use of formal methods considerably delays the start of production of code in the life-cycle. However it considerably speeds up and improves its production when it is generated.
- *More practical experience of industrial use of the methods.* A number of significant projects *have* now been undertaken using formal methods [213], but more are needed to gain a better insight into the general applicability of such techniques. Most successful formal methods project have had the help of an expert on call in case of difficultly. It remains to be seen if formal methods can be successfully applied when less expert help is at hand. Fortunately computer science undergraduate courses (in Europe at least) do now provide some suitable grounding for many software engineers who are now entering the profession. However, the effects will take some time to filter through in practice.
- *Assessment and measurement of the effectiveness of formal methods.* Metrics are a problematic area. It would obviously be helpful and commercially advantageous to know the effect of the use of formal methods on the productivity, error rates, etc., in software (and hardware) development. However these can be hard and expensive to obtain, and even if hard numbers are available, these may not actually measure the aspect that is of real interest. It is also difficult to extract such commercially sensitive into the public domain, which hampers academic study of and potential solutions to the problems. Metrics should be treated with caution, but improvements in such techniques would be worthwhile.

The actual formal methods, etc., available at any given time, can and will of course vary, and hopefully improve. For further up-to-date on-line information on formal methods, notations, and tools, held as part of the distributed World Wide Web (WWW)

Virtual Library, the reader is referred to the following URL (Uniform Resource Locator):

`http://www.afm.sbu.ac.uk/`

This chapter has considered the practical use of formal methods in general. The rest of the book concentrates on the use of Z in particular, providing a brief introduction in the next chapter, followed by a number of case studies, and appendices of related information.

Chapter 3

A Brief Introduction to Z

This chapter provides a brief guide to the main features of Z. In the space available here, it is only possible to present a flavour of the notation. Rather than give a sketchy presentation of all of Z, some parts are presented in considerably greater detail than others. Some lesser used features are omitted altogether. This eclectic description is no substitute for a full tutorial introduction to Z. There are many textbooks which already provide such introductions.

Some basic knowledge of predicate logic and set theory is highly desirable before attempting this chapter, and most of the rest of the main part of the book. If required, the reader is referred to the comprehensive list of Z textbooks collected together on page 244 in Appendix C. Many of these include a grounding in the mathematics involved before tackling the specific notation of Z. One that is widely used for Z courses is [336]. Readers already familiar with Z may skip this chapter.

3.1 Introduction

In summary, Z [381] is a typed formal specification notation based on first order predicate logic and Zermelo-Fraenkel (ZF) set theory [376]. It is a typed language which allows a certain amount of static machine checking of specifications to avoid 'obvious' errors (e.g., using the *f*UZZ [380] or ZTC [444] type-checking tools). The notation was originated and inspired by Jean-Raymond Abrial while visiting the Oxford University Computing Laboratory, and was subsequently further developed by Hayes, Morgan, Sørensen, Spivey, Sufrin and others [78]. Z is popular with governments, academics and parts of industry [18], especially those developing critical systems where the reduction of errors and quality of software is extremely important [73]. It is undergoing international standardization under ISO/IEC JTC1/SC22 [79]. A thriving Z User Group organizes regular meetings (e.g., see the ZUM'95 proceedings [69]).

3.2 Predicate Logic

The formal basis for Z is first order predicate logic extended with type set theory. Here we introduce logic only very briefly, since in practice many Z specifications actually use very few logic symbols. These tend to be hidden away by various conventions which mean that the reader can concentrate on the specification rather than the logic.

There are two logical constants in predicate logic, namely *true* and *false*. In Z, un-

like VDM [233] for example, predicates have one or other of these values. There is no third 'undefined' value, which helps considerably in minimizing the complexity of the interpretation of the language. If the result of a predicate cannot be established (for example, because of an undefined expression within the predicate), then the predicate may be interpreted as being either true or false, but the value is impossible to determine. This contrasts with some other logics which sometimes add a third 'undefined' value to handle such cases. This extra value can add considerable complexity, and the Z approach is a compromise to try to keep things as simple as possible.

3.2.1 Propositional Logic

Propositional logic is a subset of full predicate logic. It has a number of connectives which act on existing predicates. The simplest is the unary logical negation operator '$\neg\ p$' that takes a predicate value (*true*/*false*) and returns the opposite value (*false*/*true*).

There are a number of standard infix binary connectives:

- Logical conjunction '$p \land q$' returns *true* if both p and q are *true*, otherwise *false* is returned.
- Logical disjunction '$p \lor q$' returns *true* if either of p or q are *true*, otherwise *false*.
- Logical implication '$p \Rightarrow q$' is defined to be the same as '$\neg\ p \lor q$'. Intuitively, if p is *true* then q must also be *true*, otherwise q may take any value.
- Logical equivalence '$p \Leftrightarrow q$' is the same as $p \Rightarrow q \ \land \ q \Rightarrow p$. Intuitively, p and q must both have the same value for the resulting predicate to be true, otherwise *false* is returned.

3.2.2 Quantification

Full predicate logic augments propositional logic with quantification over a list of variables X, including type information in the case of Z:

- Universal quantification $\forall X \bullet q$ is only true when the predicate q is true for all possible values of X.
- Existential quantification $\exists X \bullet q$ is true if there is any (at least one) set of values for X possible which make q true. There may be more than one value; indeed all possible values, as required for universal quantification, would still be valid.
- Unique existential quantification $\exists_1 X \bullet q$ is a special but useful case of the more general existential quantification which only allows X to take a single value, not an arbitrary (non-zero) number of values.

X may be one or more variables, with type declarations in each of the above cases. The scope of the variables listed in X is bounded by the clause q in the examples above. Thus the same names may be reused outside the clause to stand for different variables if desired.

3.2.3 Laws

There is a rich set of algebraic laws for transforming predicates when performing proofs about specifications. A good selection of these is presented in [297]. Only a

very cursory introduction to predicate logic has been given here. For more information, see almost any of the Z textbooks listed on page 244.

3.3 Sets and Relations

First order predicate logic forms the logical foundations of Z. Another important aspect of Z is set theory. In combination these areas of discrete mathematics are helpful in formally describing computer-based systems. This is presented in more detail here.

Z includes the concept of *types*. In this section we first discuss why the use of types is important in specifications. We then introduce the notion of *sets* and the *operations* which may be performed on sets. Finally we consider *relations* between elements of sets.

3.3.1 Types

Z is a typed language; that is to say, every variable in Z has a particular type (i.e., set from which it is drawn) associated with it which must match appropriately when it is combined with other variables.

The question arises, why fuss about types? They introduce a lot of extra complexity to a specification. However this proves to be well worthwhile for the following reasons:

1. It helps to structure specifications.

   We can specify the set of possible values from which a variable can be drawn, and we can then further constrain the variable using a predicate if required. For example in the (English) statement:

   x is a path of least cost from A to B

   we could give x a type:

   $x : Path$

   and constrain it further with a predicate:

   $least\_cost\, x$

2. We wish to refine the specification into code.

   Eventually, each variable in a specification will need to be implemented using some data type in a programming language. It is worth thinking about this earlier than at the coding stage. By adding the discipline of types, more thought must be put into the specification. This means more errors are likely to be ironed out at this early stage rather than later on when mistakes are more expensive to correct.

3. It helps avoid nonsense specifications.

   The use of types means the specification can more easily be checked for consistency, either manually by human inspection, or automatically using the machine assistance of a type-checker. For example, given $Z : Methods$ and $Jonathan : People$, it would be meaningless to say

   $Z = Jonathan$

Consider the introduction of predicates in specifications (using a *schema* box):

A, B : *People* x : *Likes*
Predicate(x)

The first part of this is a *signature* (of variables including types) and the second part is a *predicate*. The parts of the signature after the colons specify the *types* of the variables before the colons, much like definitions in a programming language such as Pascal.

How can we build up *types* and what are the variables in *predicates*? For this, we need to explore the world of *set* theory.

3.3.2 Sets

Mathematicians early in the 20th century discovered how to construct a world of sets, powerful enough to describe anything we meet in practice.

A set is a collection of objects or *elements* of some type. Here are some examples of the notation used in Z to describe sets:

$\emptyset$ This denotes the *empty set* – i.e., the set with no elements in it. This can also be denoted as $\{\}$.

$\{Jane, Alice, Emma\}$ This is a set of people containing three elements. Note that the *types* of *Jane, Alice* and *Emma* must be compatible!

$\{0, 1, 2\} = \{0, 0, 1, 0, 2, 1\} = \{1, 2, 0\}$ All three of these represent a three element set of numbers. An important property of sets is that there is no inherent ordering in its elements (unlike in a list for example). Thus the numbers above may be specified in any order, and repeated elements simply map on top of one another.

$\{0, 1, 2, 3, \ldots\}$ This is the set of *natural numbers*, or integers ranging from zero upwards. Note that this set contains an infinite number of elements. The '$\ldots$' in this definition is *informal* (i.e., not part of the Z language). We cannot write out all the elements of this set using this notation. Later we shall introduce a method for overcoming this. Since the set of natural numbers is a very important set in many Z specifications, it is normally denoted $\mathbb{N}$ for brevity.

$\{\emptyset\} \neq \emptyset$ It is important to understand that the set containing the empty set is *not* the same as the empty set itself. It is a set containing one element (namely, the empty set). Thus it is possible to have sets containing other sets (which may themselves contain sets), and so on. We will look at this again later.

Note that every set must be drawn from some *basic type* (or *given set*) in Z. This even applies to the empty set, $\emptyset$. I.e., there is a different empty set for each type. When the empty set is used in Z, its type should be obvious from the context. If not, there is probably something wrong with the specification. To avoid confusion, the notation $\emptyset[T]$ may be used to indicate the empty set drawn from type T.

It is often important to be able to say that an element belongs to ('*is a member of*') a particular set. In set theory, we write 'x is an element of S' as:

$$x \in S$$

This is a predicate (extra constraint) on x and S. Note that x and S must be type-compatible for this to be meaningful. Here are some examples of this notation:

$0 \in \{0, 1, 2\}$ This is patently true since 0 is one of the elements in the set containing 0, 1 and 2.

$\emptyset \in \{\emptyset\}$ The empty set (of a particular type) is a member of the set containing just the empty set of that type. In fact it is the only member.

$0 \notin \emptyset$ This is another way of writing '$\neg\ (0 \in \emptyset)$' – i.e., 0 is *not* a member of the empty set (of numbers). This is true because no element can be a member of the empty set; there are no elements in it by definition. In fact, for any x, we can say $x \notin \emptyset$. A special notation is used for '*is not a member of*' since this occurs quite often in specifications.

$\mathit{William} \notin \{\mathit{Jonathan}, \mathit{Jane}, \mathit{Alice}, \mathit{Emma}\}$

The set being checked by $\notin$ need not be empty for a predicate using it to be true. For example, the set of people given here does not include *William*, so the predicate is true.

Question: Is $\{0\} \in \{\{0, 1, 2\}\}$ true?

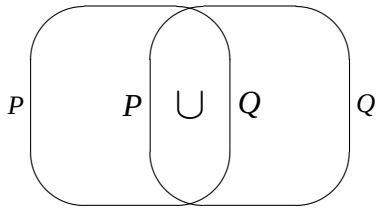
Operations on sets

So far, we have discovered how to denote (finite) sets as a number of elements and how to specify membership and non-membership of sets. However, to manipulate sets usefully, we need a richer collection of operators on sets. We shall now look at a number of such operators.

The following operators two sets and return a new one:

$P \cup Q$ '*P union Q*' – union.

This returns a set containing all elements which are either a member of P or a member of Q (or both). Pictorially, we may view this as a Venn diagram:

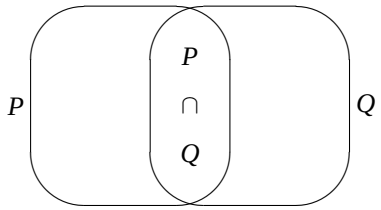


Note that the notation $\{a, b, c, \ldots\}$ is effectively shorthand for

$$\{a\} \cup \{b\} \cup \{c\} \cup \ldots$$

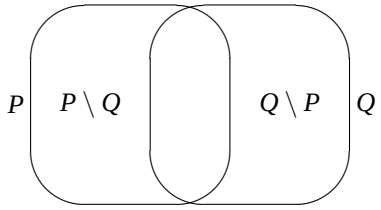
$P \cap Q$ '*P intersect Q*' – intersection.

This gives a set containing elements which are both a member of P and a member of Q.



$P \setminus Q$ '*P minus Q*' – difference.

This gives the set with elements which are contained in P, but are not members of Q.



The following are *predicates* on sets (or *expressions* representing sets):

$P = Q$ '*P equals Q*' – identity of sets.

The sets P and Q each contain exactly the same elements. As we have seen before,

$$\{Alice\} = \{Alice, Alice, Alice\}$$
$$\{Alice, Emma\} = \{Emma, Alice\}$$

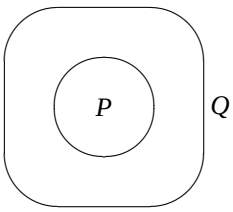
You can see from the previous three Venn diagrams that

$$(P \setminus Q) \cup (P \cap Q) \cup (Q \setminus P) = P \cup Q$$

Often we wish the say the negation of $P = Q$ – i.e., P and Q do *not* contain the same elements. This may be written as '$P \neq Q$' (which is equivalent to '$\neg (P = Q)$').

$P \subseteq Q$ '*P contained in Q*' – subset.

All the elements of P are in Q. Note that it may be the case that $P = Q$. To specify a *strict subset*, we use the notation '$P \subset Q$'. This is shorthand for '$P \subseteq Q \land P \neq Q$'.

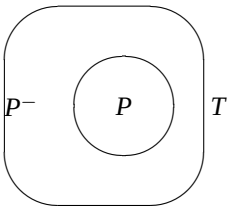


The following operator takes a set and returns another set:

P^- '*complement P*' – complementation.

Set theory includes the idea of the *complement* of a set – i.e., all the elements *not* in the set. Because Z is typed, this means all the elements of the same *type* not in the set. This is not a part of Spivey's 'toolkit' of operators [381], but can easily be defined in terms of the '$\setminus$' operator. For a type T where $P \subseteq T$,

$$P^- = T \setminus P$$



Note that $T^- = \emptyset[T]$ and $\emptyset[T]^- = T$.

For completeness, we include our concept of set membership here:

$x \in P$ '*x belongs to P*' – membership.

x is one of the elements in P; and conversely, '$x \notin P$' means x is not one of the elements in P (i.e., '$\neg\ (x \in P)$').

Question: Simplify:

1. $(P \setminus Q) \setminus P$
2. $(P \setminus Q) \cup (P \cap Q)$

Generalized set operations

Set union and intersectiongeneralized intersection my be generalized to act on a set of sets rather than just a pair of sets. Informally:

$$\bigcup\{A, B, C, \ldots\} = A \cup B \cup C \cup \ldots$$
$$\bigcap\{A, B, C, \ldots\} = A \cap B \cap C \cap \ldots$$

Set comprehension

So far we have defined sets in terms of individual members of that set. This is a rather restrictive (although often useful) method of defining sets. For any set containing more than a few elements, it is going to be a very cumbersome way of specifying sets. For infinite sets, it fails completely since we would require an infinitely long document to describe such sets.

What we need is a more general way of specifying sets. For this, we use a construction known as *set comprehension*. This 'comprehensive form' of set definition takes the following form:

$$\{x : \text{Type} \mid \text{Predicate}(x)\}$$

or more generally $\{\text{Signature} \mid \text{Predicate}\}$, where the signature may include may than one variable.

Here are some examples of the use of set comprehension:

$\{x : \mathbb{N} \mid x\ \mathit{is\_prime}\}$ $(= \{2, 3, 5, 7, 11, 13, \ldots\})$.

This defines the set of all prime numbers (provided *is_prime* is defined appropriately). This is, of course, an infinite set.

$\{x : \mathit{Path} \mid \mathit{least\_cost}(x)\}$

This is the set of paths between A and B such that the cost is minimized. There could be more than one such path if two or more paths have equally low cost; or the set could be empty if there are *no* paths between A and B.

$\{x : Methods \mid Jonathan \underline{uses}\ x\}$

This will define the set of all methods which *Jonathan* uses. This could be the empty set, or it could be one or more methods depending on '*uses*' and *Jonathan*. (It is unlikely to be an infinite set, but may be large if, for instance, the number of formal methods continues to proliferate!) Note that underlining is sometimes used in Z to clarify infix operators like *uses* here.

Question: How could we write a definition for $\emptyset$ using set comprehension?

Subsets

We have introduced the idea of a *subset* – i.e., a set contained within another set. Let us look at a few examples of this in use:

$A \subseteq B$ Each element of A is also an element of B.

$\emptyset \subseteq A$ This is always true. Every set contains at least the empty set (of compatible type).

$A \subseteq A$ Every set 'contains' itself.

$\{0, 1\} \subseteq \mathbb{N}$

The numbers 0 and 1 make up a subset of the natural numbers.

$\{x : \mathbb{N} \mid x\ is\_prime \land x \neq 2\} \subseteq \{x : \mathbb{N} \mid x\ is\_odd\}$

The set of prime numbers (not including the number 2) is a subset of the odd numbers.

Sets and predicates

Sets and predicates are similar in some respects. Each set operator has a corresponding logical connective which may be associated with it. For example, set intersection ($\cap$) and logical conjunction ($\land$) are connected as follows, where p and q are predicates, typically involving x:

$$\{x : T \mid p\} \cap \{x : T \mid q\} = \{x : T \mid p \land q\}$$

A similar equivalence holds for union ($\cup$) and disjunction ($\lor$):

$$\{x : T \mid p\} \cup \{x : T \mid q\} = \{x : T \mid p \lor q\}$$

Complementing a set is like negating a predicate:

$$\{x : T \mid p\}^- = \{x : T \mid \neg\, p\}$$

Note that in the above case, it is important that T is a basic type.

The notion of a subset ($\subseteq$) matches that of implication ($\Rightarrow$):

$$\{x : T \mid p\} \subseteq \{x : T \mid q\} \text{ iff } p \Rightarrow q$$

Set equality ($=$) matches logical equivalence ($\Leftrightarrow$):

$$\{x : T \mid p\} = \{x : T \mid q\} \text{ iff } p \Leftrightarrow q$$

The empty set (of a particular type T) is equivalent to the *false* predicate:

$$\emptyset[T] = \{x : T \mid false\}$$

The entire set of a given type is equivalent to the *true* predicate:

$$T = \{x : T \mid true\}$$

Set operations

Consider the subset operator $\subseteq$. This has a number of mathematical properties. For sets P, Q and R, drawn from the type T (i.e., $P \subseteq T$, $Q \subseteq T$ and $R \subseteq T$):

$\subseteq$ is *reflexive*:	$P \subseteq P$
$\subseteq$ is *transitive*:	$(P \subseteq Q \land Q \subseteq R) \Rightarrow P \subseteq R$
$\subseteq$ is *antisymmetric*:	$(P \subseteq Q \land Q \subseteq P) \Rightarrow P = Q$
$\emptyset[T]$ is the *minimum* of T:	$\emptyset[T] \subseteq P$

Similarly, set intersection ($\cap$) also has a number of properties:

- $\cap$ is the *greatest lower bound* of $\subseteq$:

  $$R \subseteq P \land R \subseteq Q \Leftrightarrow R \subseteq (P \cap Q)$$

  $P \cap Q$ is the largest subset of both P and Q.

- $\cap$ is

idempotent	$P \cap P = P$
symmetric	$P \cap Q = Q \cap P$
associative	$(P \cap Q) \cap R = P \cap (Q \cap R)$
monotonic	$P \subseteq Q \Rightarrow (R \cap P) \subseteq (R \cap Q)$

We can write P^- for the complement of P (with respect to its type!). Complementing a set is *involutive*:

$$(P^-)^- = P$$

Compare the following with *contrapositive* in predicate logic:

$$P \subseteq Q \Leftrightarrow Q^- \subseteq P^-$$

($\neg$). Note that

$$P \subseteq Q \Leftrightarrow P \cap Q^- = \emptyset$$

We normally write $P \cup Q$ for $(P^- \cap Q^-)^-$ (one of De Morgan's laws). We could list further properties of $\cup$, etc. See [381] for a more comprehensive collection of laws.

Cartesian product

Sometimes it is useful to associate two or more sets together in order to build up more complex types. If T and U are types then the *Cartesian product*

$$T \times U$$

denotes the type of ordered pairs

$$(t, u)$$

with $t : T$ and $u : U$. If P and Q are subsets of types T and U respectively then

$$P \times Q = \{p : T;\ q : U \mid p \in P \land q \in Q\}$$

The notation may be generalized to an ordered *n-tuple* and any valid expression may be used for the types:

$$E_1 \times E_2 \times \ldots \times E_n$$

Here are some examples of the notation of tuples $(\ldots)$ and Cartesian products in use:

$(Z, Jonathan) \in Methods \times People$

A method has been associated with a person as an ordered pair or 2-tuple, perhaps because they approve of it.

$(Oxford, Cambridge) \in Places \times Places$

Two places are associated with each other (in some sense!). Here the types are the same. This could be useful when specifying some relationship between places, for example.

$(Jonathan, Oxford, 1956) \in People \times Places \times \mathbb{N}$

This could be specifying the place and year of birth, as a 3-tuple.

Sometimes it is useful to extract the *first* or *second* element from an ordered pair. We use the following notation for this:

$$first(t, u) = t$$

$$second(t, u) = u$$

For an ordered pair t, the following law applies:

$$(first\ t, second\ t) = t$$

Question: Is $X \times Y \times Z$ equivalent to $(X \times Y) \times Z$ or $X \times (Y \times Z)$?

Set comprehension revisited

Recall that set comprehension takes the form:

$$\{ \text{Signature} \mid \text{Predicate} \}$$

For example:

$$\{x : Methods;\ y : People \mid Jonathan\ uses\ x \land x\ approved\_by\ y\}$$

The signature forms an *ordered* tuple of type $Methods \times People$. *Note that the order of the signature is important.* I.e., here:

$x : Methods;\ y : People$ is not the same as $y : People;\ x : Methods$

The latter would give a tuple in the opposite order from the first.

Sometimes it is desirable to extract only parts of the signature to define a set using set comprehension. More generally:

$$\{x_1 : T_1;\ \ldots;\ x_n : T_n \mid \text{Predicate}\}$$

is the same as

$$\{x_1 : T_1;\ \ldots;\ x_n : T_n \mid \text{Predicate} \bullet (x_1, \ldots, x_n)\}$$

This notation permits us to write other, more complex, sets. For example, the set of squares of primes may be defined as:

$$\{x : \mathbb{N} \mid x\ is\_prime \bullet x * x\}$$

Here the expression $x * x$ acts as the defining term for the set.

In general the last term after the '•' may be any valid expression:

$$\{\ \text{Signature} \mid \text{Predicate} \bullet \text{Expression}\ \}$$

The signature declares any variables required for the set comprehension definition, the predicate constrains them as required, and the expression returns the elements in the desired set. We may omit this last expression when it is simply a tuple formed from the components of the signature. We can omit the predicate if it is simply *true*.

Power set

When defining a variable which is itself a set, its type will be a set of sets. Since many variables in Z specifications are sets, we use a special notation for this. If S is a set, $\mathbb{P}\,S$ denotes the set of all subsets of S, or the *power set* of S. Note that

$$X \in \mathbb{P}\,S \Leftrightarrow X \subseteq S$$

Also $\emptyset \in \mathbb{P}\,S$, so $\mathbb{P}\,S \neq \emptyset$. Here are some examples of power sets:

$\mathbb{P}\,\emptyset$ This set just contains the empty set of a given type (i.e., $\mathbb{P}\,\emptyset = \{\emptyset\}$).

$\mathbb{P}\{a\} = \{\emptyset, \{a\}\}$

The power set of a singleton set is a set consisting of the empty set and the singleton set.

$\mathbb{P}\,\mathbb{N}$ This is the set of all possible sets of natural numbers. This is infinite of course.

$\mathbb{P}\,Path$ Set of all sets of paths from A to B.

$\mathbb{P}(\mathbb{N} \times \mathbb{N})$ Sets of pairs of numbers. The Cartesian product may be used as required in the definition of power sets.

$S == \mathbb{P}((Places \times Places) \times Path)$

This is an *abbreviation definition*. $X == Y$ means 'replace X by Y' where X is used subsequently.

Here brackets have been used to group and nest Cartesian products. This can be done to an arbitrary depth, but it is best to limit such usage to aid readability.

Given the definition above, we can say $\_ \rightarrowtail \_ \in S$. The under scores are placeholders for parameters; i.e., $\rightarrowtail$ is an infix operator here of type $(Places \times Places)$ to the left and *Path* to the right.

A power set can include infinite subsets. For example, $\mathbb{N} \in \mathbb{P}\,\mathbb{N}$. If we are specifically interested in finite subsets, then Z has a special notation for this. If S is a set, $\mathbb{F}\,S$ denotes the set of all *finite* subsets of S. We will explore exactly what 'finite' means later.

Sometimes we are interested in non-empty subsets. For these, we use the notation $\mathbb{P}_1$ (non-empty power set) or $\mathbb{F}_1$ (non-empty set of finite subsets):

$$\mathbb{P}_1 S == \mathbb{P} S \setminus \{\emptyset\}$$

$$\mathbb{F}_1 S == \mathbb{F} S \setminus \{\emptyset\}$$

3.3.3 Relations

Sometimes individual elements in one set are related to particular elements in another set. For example,

R_1**:** Places 'A' and 'B' are adjacent.

R_2**:** Path x costs y to traverse.

R_3**:** $(A, B) \looparrowright P$ (P is a path from A to B of least cost.)

These are all examples of relationships (two-place predicates). A relation R between sets P and Q is a subset of $P \times Q$:

$$R \subseteq P \times Q$$

Taking the three examples above, we can be slightly more formal:

$(A, B) \in R_1$ iff A,B adjacent.

$(x, y) \in R_2$ iff path x costs y.

$((A, B), P) \in R_3$ iff P of least cost A to B.

If R is a relation from P to Q, we often write

$p \; R \; q$ instead of $(p, q) \in R$

(For example, '$s \looparrowright t$'.) We can write $\_R\_$ for relation R to indicate that it is an infix relation (as in $\_ \looparrowright \_$). We can also use the *maplet* notation

$$p \mapsto q$$

instead of (p, q) as a more graphical indication that p is related to q (although the mathematical meaning is identical). Here are some examples of relations:

$$\_ \leq \_ = \{x, y : \mathbb{N} \mid (\exists z : \mathbb{N} \bullet x + z = y)\}$$

$$uses = \{Jonathan \mapsto Z, Peter \mapsto VDM, Peter \mapsto RAISE\}$$

If T, U are types,

$$T \leftrightarrow U == \mathbb{P}(T \times U)$$

denotes the set of all relations from T to U.

$$\_ \leq \_ \in \mathbb{N} \leftrightarrow \mathbb{N}$$

$$uses \in \{Jonathan, Mike, Peter\} \leftrightarrow Methods$$

Every relation has a *domain* and *range*. The domain of a relation $R : T \leftrightarrow U$ is the set of all elements in T which are related to at least one element in U by R. The range is all elements in U related to at least one in T. Formally:

$$\begin{aligned} \mathrm{dom}\, R &= \{x : T \mid (\exists y : U \bullet (x, y) \in R)\} \\ \mathrm{ran}\, R &= \{y : U \mid (\exists x : T \bullet (x, y) \in R)\} \end{aligned}$$

Every relation also has an *inverse*. The inverse of a relation is another relation with all its tuples (x, y) reversed (i.e., (y, x)). Again, formally we have:

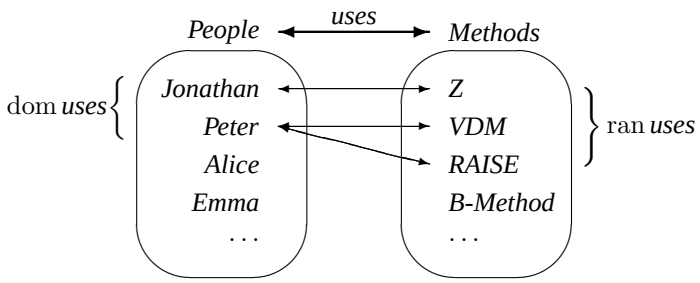
$$\begin{aligned} R^{\sim} &= \{x : T;\ y : U \mid (x, y) \in R \bullet (y, x)\} \\ &= \{x : T;\ y : U \mid (x, y) \in R \bullet y \mapsto x\} \\ &= \{y : U;\ x : T \mid (x, y) \in R\} \end{aligned}$$

If R is of type $T \leftrightarrow U$, then its inverse $R^{\sim}$ is of type $U \leftrightarrow T$. Note that $R^{\sim}$ may also be written as R^{-1}.

As an example, for the '*uses*' relation we have:

$$\begin{aligned} \mathrm{dom}\, \mathit{uses} &= \{\mathit{Jonathan}, \mathit{Peter}\} \\ \mathrm{ran}\, \mathit{uses} &= \{Z, \mathit{VDM}, \mathit{RAISE}\} \\ \mathit{uses}^{\sim} &= \{Z \mapsto \mathit{Jonathan}, \mathit{VDM} \mapsto \mathit{Peter}, \mathit{RAISE} \mapsto \mathit{Peter}\} \end{aligned}$$

Pictorially, we may view the '*uses*' relation as follows:



Here is another example:

$$\begin{aligned} \mathrm{ran}\, \_ \leq \_ &= \mathrm{dom}\, \_ \leq \_ \\ &= \mathbb{N} \\ (\_ \leq \_)^{\sim} &= \_ \geq \_ \end{aligned}$$

The following laws concerning dom, ran and $^{\sim}$ are relevant when reasoning about relations:

$$\begin{aligned} \mathrm{ran}(R^{\sim}) &= \mathrm{dom}\, R \\ \mathrm{dom}(R^{\sim}) &= \mathrm{ran}\, R \\ (R^{\sim})^{\sim} &= R \end{aligned}$$

Otherwise, reasoning about relations is as for sets.

3.4 Functions and Toolkit Operators

We have explored the world of sets and relations, particularly with respect to the Z notation. Next we consider an important class of relations known as *functions*, and some operators which are useful for manipulating both relations in general, and functions in particular.

3.4.1 Functions

In Z, functions are a special case of relations in which each element in the domain has at most one value associated with it. For example, a partial function '$\pfun$' is defined using set comprehension in terms of a more general relation '$\rel$' as follows, using an 'abbreviation definition':

$$X \pfun Y == \\ \quad \{ f : X \rel Y \mid (\forall x : X;\ y_1, y_2 : Y \bullet \\ \qquad\qquad (x \mapsto y_1) \in f \land (x \mapsto y_2) \in f \Rightarrow y_1 = y_2) \}$$

I.e., any element x in the domain can only map to a single value in the range of the function, not to two (or more) different ones. X and Y are identifiers which represent arbitrary formal *generic parameters*. These are local to the right hand side of the definition. $\pfun$ is an infix symbol, and here '$X \pfun Y$' is actually a short form for '$(\_ \pfun \_)[X, Y]$' where the underlines indicate positions of the parameters. The generic parameters may be explicitly given in square brackets when $\pfun$ and other similar constructs are used, but they are normally omitted since in most cases they can be inferred from the context and clutter the specification unnecessarily.

There are a number of special types of function in Z, each given there own unique type of arrow. For example, *total* functions, as indicated by '$\fun$' have a domain consisting of all the possible allowed values.

Here is a list of the other standard types of function available in Z:

- $X \pinj Y$ – Partial injections, in which there is a one-to-one mapping between elements in the domain and the range. Different values in the domain map to different values in the range. Thus the inverse is also a function.
- $X \inj Y$ – Total injections, where the function is a partial injective in which the domain completely populates the set X. I.e., if a function is defined as $f : X \inj Y$, then $\mathrm{dom}\, f = X$.
- $X \psurj Y$ – Partial surjections, where the range completely populates the set Y. I.e., if a function is defined as $f : X \psurj Y$, then $\mathrm{ran}\, f = Y$.
- $X \surj Y$ – Total surjections, where both the domain and the range completely populate X and Y respectively. Here, if a function is defined as $f : X \surj Y$, then $\mathrm{dom}\, f = X$ and $\mathrm{ran}\, f = Y$.
- $X \bij Y$ – Bijections, one-to-one total injective and surjective functions. Again, with a function defined as $f : X \surj Y$, $\mathrm{dom}\, f = X$ and $\mathrm{ran}\, f = Y$, but in addition, the inverse is also a total function (i.e., $f^{\sim} \in Y \fun X$).
- $X \ffun Y$ – Finite partial functions, where the domain of the function (and hence the also range which can never be larger than the domain for a function) must be finite. This is a subset of all partial functions on X and Y, and also of $\mathbb{F}(X \times Y)$.
- $X \finj Y$ – Finite partial injections, which as well as being finite are also one-to-one. This is the same as the intersection of finite functions and partial injections on X and Y.

All these different types of function are formally defined on pages 105 and 112 of [381].

Function application is written in the standard mathematical manner in Z; for example, $f(x)$ applies the element x to the function f. It is permissible to omit the brackets if desired; e.g., '$f\,x$' is the same as $f(x)$. Brackets should be used in Z when they help with clarity, and are often omited in practice when they are not necessary.

3.4.2 Toolkit operators

There are many operators on relations and functions which help make up a mathematical 'toolkit' of Z. A widely accepted set of such definitions are presented in full in [381], together with many relevant laws. These are also summarized as part of the *Z Glossary* in Appendix B; in particular, see page 234. However, note that the proposed Z standard [79] may affect which operators are considered 'standard' in future. Some of the generally accepted main operators are briefly presented informally here:

- $\mathrm{id}\,A$ – *Identity relation*, in which each element in the domain is mapped onto itself (and only itself) in the range, and vice versa.
- $Q \fatsemi R$ – *Forward relational composition*, where elements that match in the range of the first relation and the domain of the second relation are joined together to form a new binary relation. In the resulting relation, the domain is a subset of the domain of the first original relation and the range is a subset of the range of the second original relation. In Z, this is often just known as 'relational composition'.
- $Q \circ R$ – *Backward relational composition*. This is the same as forward relational composition with the parameters reversed: $R \fatsemi Q$. Forward relational composition is more widely used in Z than backward relational composition for stylistic reasons.
- $A \lhd R$ – *Domain restriction*. The set on the left hand side is used to restrict the resulting relation to be just the mappings in the original relation which have a member of that set as the first element in each tuple.
- $A \lhd\!\!\!\!- R$ – *Domain anti-restriction*. Here the complement of the set is used to restrict the relation.
- $R \rhd A$ – *Range restriction*. This is similar to domain restriction, but the range of the relation is restricted by the set instead.
- $R \rhd\!\!\!\!- A$ – *Range anti-restriction*. Hopefully you can work out what this does from the three previous operators!
- $R(\!|\ A\ |\!)$ – *Relational image*. Here the relation R is restricted in a similar manner to domain restriction
- *iter* $n\,R$ – *Relational iteration*. The relation R is composed n times with itself. This is often written as R^n in Z for brevity. The relation must have a type-compatible domain and range (say A). Then *iter* $0\,R$ (R^0) is the same as $\mathrm{id}\,A$, *iter* $1\,R$ is the same as R, *iter* $2\,R$ is the same as $R \fatsemi R$, *iter* $3\,R$ is the same as $R \fatsemi R \fatsemi R$, and so on.
- $R^\sim$ – *Inverse of relation*. All tuples in the relation are reversed. Thus the domain becomes the range and vice versa. In the case where the domain and range have the same type (i.e., the relation is *homogeneous*), this is the same as R^{-1}, defined using relational iteration as presented above. More precisely, $^\sim$ may be applied to any relation $R : X \leftrightarrow Y$, but R^{-1} may only be used if the relation R has a type of the form $X \leftrightarrow X$.

- R^* – *Reflexive-transitive closure* consists of the union of all possible non-negative iterations of R – i.e., $R^0 \cup R^1 \cup R^2 \cup \ldots$ In Z, this is often just known as 'transitive closure'.

- R^+ – *Irreflexive-transitive closure*. This is similar to reflexive-transitive closure, but for strictly positive iterations only. I.e., it does not include the identity relation R^0. Thus $R^+ = R^* \setminus R^0$.

- $Q \oplus R$ – *Overriding*. For each tuple in the right hand relation R, any tuples with a matching first element in the left hand relation Q are omitted and the tuple is included in the resulting relation. If no such tuple exists for a given tuple in Q, then it is included in the resulting relation. More formally, $Q \oplus R$ is the same as $(\mathrm{dom}\, R \lhd Q) \cup R$. Note that $\oplus$ is often applied to functions where part of the left hand function needs to be modified by the right hand function (often typically a single tuple), but in the general case in may be applied to a pair of relations.

3.5 Numbers and Sequences

Two important concepts in many specifications are *numbers* and *sequences*, and we will give more details of these here. Z provides integers and natural numbers (integers from zero up) as sets in its standard toolkit, together with many of the standard operators on these. Numbers are also useful for defining sequences.

Sets provide a method of describing a collection of unindexed objects. However, sometimes we do wish to index the objects in some way. Since this is often important in specifications, and particularly as a first step in data refinement (e.g. implementing a set as an array), Z provides the notion of a sequence, in which the objects are 'labelled' with a natural number. A number of operators and functions for the manipulation of sequences are also included in the standard Z toolkit. These, and other operators are covered in this section.

3.5.1 Numbers

Let us consider sets of numbers which could be useful in defining sequences. The set of integers

$$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$$

and particularly the set of natural numbers

$$\mathbb{N} = \{n : \mathbb{Z} \mid n \geq 0\} = \{0, 1, 2, \ldots\}$$

are often used in Z specifications.

Arithmetic

The following operators on integers are assumed and may be used in expressions as required:

addition	$\_ + \_$	$2 + 2 = 4$
subtraction	$\_ - \_$	$4 - 2 = 2$
multiplication	$\_ * \_$	$2 * 2 = 4$
division	$\_ \,\mathsf{div}\, \_$	$5 \,\mathsf{div}\, 2 = 2$
modulo arithmetic	$\_ \,\mathsf{mod}\, \_$	$5 \,\mathsf{mod}\, 2 = 1$
negation	$-$	$-(-2) = 2$

Brackets may be used for grouping (as they may in any Z expression). For example, $2 * (4 + 5) = 18$. The standard comparison relations are available:

less than	$\_ < \_$	$2 < 3, \neg\, 3 < 2$
less than or equal	$\_ \leq \_$	$2 \leq 2$
greater than	$\_ > \_$	$a > b \Leftrightarrow b < a$
greater than or equal	$\_ \geq \_$	$a \geq b \Leftrightarrow b \leq a$

The maximum and minimum values of a (non-empty) set of numbers can be determined.

maximum of a set	$max\{1, 2, 3\} = 3$
minimum of a set	$min\{1, 2, 3\} = 1$

Care should be taken to ensure that the set supplied to *max* or *min* is non-empty. Otherwise the result will be undefined.

Extra operators may easily be added if required. For example, the function which returns the absolute value of an integer may be defined using an *axiomatic description*:

$$
\begin{array}{|l}
abs : \mathbb{Z} \to \mathbb{Z} \\
\hline
\forall n : \mathbb{Z} \bullet \\
\quad n \leq 0 \Rightarrow abs\ n = -n \land \\
\quad n \geq 0 \Rightarrow abs\ n = n
\end{array}
$$

Note that $\mathrm{ran}\, abs = \mathbb{N}$.

Question: Can you suggest an alternative definition for *abs*?

An axiomatic description is available for use globally in the rest of a specification. Such a description may be a *loose* specification in that there may be more than one possible model for the specification. E.g., for the description

$$
\begin{array}{|l}
n : \mathbb{N} \\
\hline
n \leq 10
\end{array}
$$

any integer value of *n* from 0 to 10 is allowable.

Sometimes strictly positive (non-zero) natural numbers are of interest. The notation

$$\mathbb{N}_1 = \mathbb{N} \setminus \{0\} = \{1, 2, 3, \ldots\}$$

is used for this.

Exercise: Write definitions for:

1. the square of an integer,
2. the factorial of a natural number.

The successor function *succ* returns the next number when applied to a natural number:

$$succ = \{0 \mapsto 1, 1 \mapsto 2, 2 \mapsto 3, \ldots\}$$

Thus $\operatorname{ran} succ = \mathbb{N}_1$. The successor function is often useful in specifications. Sometimes the inverse predecessor function is also useful. If this is so, we could define

$$pred == succ^{\sim}$$

The following laws apply:

$$\begin{aligned} succ &= \mathbb{N} \lhd (\_ + 1) \\ &= (\_ + 1) \rhd \mathbb{N}_1 \end{aligned}$$

$$\begin{aligned} pred &= \mathbb{N}_1 \lhd (\_ - 1) \\ &= (\_ - 1) \rhd \mathbb{N} \end{aligned}$$

$$\begin{aligned} succ \fatsemi succ &= \mathbb{N} \lhd (\_ + 2) \\ &= (\_ + 2) \rhd (\mathbb{N}_1 \setminus \{1\}) \end{aligned}$$

$$succ \fatsemi pred = \operatorname{id} \mathbb{N}$$

$$pred \fatsemi succ = \operatorname{id} \mathbb{N}_1$$

Iteration

Sometimes we wish to compose a relation $R : X \leftrightarrow X$ (i.e., one in which the types of the domain and range match) a certain number of times, *n*. As previously mentioned, we normally write this as R^n. Informally we have

$$R^n = R \fatsemi R \fatsemi \ldots \fatsemi R \qquad k \text{ times}$$

Using $succ : \mathbb{N} \rightarrow \mathbb{N}$ as a specific example, we can consider the following cases:

$succ^0 = \operatorname{id} \mathbb{N}$ Composing a relation zero times simply gives the identity relation. It is as if the relation is not there, so elements are just mapped onto themselves

$succ^1 = succ$ Composing a relation once is the same as the relation itself.

$succ^2 = succ \fatsemi succ$ This is the same as composing the relation with itself.

$succ^n = \mathbb{N} \lhd (\_ + n) = (\_ + n) \rhd \{i : \mathbb{N} \mid i \geq n\}$
For any $n : \mathbb{N}$, the above law applies.

$succ^{-1} = succ^{\sim}$ The inverse of a relation is a special case of iteration. As previously mentioned, the notations $R^{\sim}$ and R^{-1} may be used interchangeably if the domain and range have the same type.

Exercise: Provide another way of writing:

1. $R^m \fatsemi R^n$
2. $(R^m)^n$

Number range

A number range (a set of numbers) between two integers $a, b : \mathbb{Z}$ is denoted as

$$a \mathinner{..} b = \{a, a+1, a+2, \ldots, b-2, b-1, b\}$$

or more formally

$$a \mathinner{..} b = \{n : \mathbb{Z} \mid a \leq n \leq b\}$$

If $a > b$ then $a \mathinner{..} b = \emptyset$. Also $a \mathinner{..} a = \{a\}$.

Cardinality

The *cardinality* of a (finite) set is the number of elements in that set, or the size of the set. The cardinality of a set $s \in \mathbb{F}\, T$ (the set of all finite subsets of T – see page 39) is denoted:

$$\#s$$

Thus $\#\emptyset = 0$, $\#\{a\} = 1$, $\#\{a, b\} = 2$ (if $a \neq b$) and so on. For a number range $a \mathinner{..} b$,

$$\begin{array}{lcll} \#a \mathinner{..} b & = & 1 + b - a & \text{if} \quad a \leq b \\ & = & 0 & \text{if} \quad a > b \\ & = & max\{0, 1 + b - a\} & \end{array}$$

For a set to be 'finite', it must be possible to map from a natural number in the range $1 \mathinner{..} n$ uniquely onto each element in that set. n is then the cardinality or size of the set. This mapping can be done with a suitable finite partial injective function. For example, the set $a \mathinner{..} b$ (where $a \leq b$) may be mapped using the function $f : \mathbb{N} \rightarrowtail \mathbb{Z}$ such that $f = succ^{a-1} \rhd a \mathinner{..} b$. The range ($\mathrm{ran}\, f$) is $a \mathinner{..} b$, and the domain ($\mathrm{dom}\, f$) is $a - (a-1) \mathinner{..} b - (a-1)$ or $1 \mathinner{..} 1 + b - a$. Thus the cardinality is $1 + b - a$, as stated above.

Pictorially we have:

$$\begin{array}{ccccccc} 1 & 2 & 3 & \cdots & b-a & 1+b-a \\ \downarrow & \downarrow & \downarrow & \cdots & \downarrow & \downarrow \\ a & a+1 & a+2 & \cdots & b-1 & b \end{array}$$

3.5.2 Types revisited

In Z, integers ($\mathbb{Z}$) are normally supplied as a basic type (although see below) together with standard arithmetic operators. The natural numbers $\mathbb{N}$ ($0, 1, 2 \ldots$) are defined as a subset of the integers. Thus it is not a true type; rather the type of $n : \mathbb{N}$ is $\mathbb{Z}$ and n has the extra constraint that it must be greater than or equal to zero. For more information on determining types, see the beginning of Chapter 2 in [381].

By convention, $\mathbb{R}$ is sometimes used to denote real numbers if these are needed [411], although these are not defined in Spivey's Z toolkit [381]. They may be defined in the proposed Z standard [79]. However many applications for which Z is used do not involve real numbers. As well as numbers, we can also define our own types.

Sometimes we are not particularly interested in the exact values of a set, just the

fact that it exists. We introduce such sets as a *basic type* (or *given set*):

$$[\, Places \,]$$

or for several sets,

$$[\, A, B, C \,]$$

If one particular place is of interest, we could define

$$London : Places$$

or for more than one place (for example):

$$Oxford, Cambridge : Places$$
$$\vdash$$
$$Oxford \neq Cambridge$$

It is important to specify that $Oxford \neq Cambridge$ here if we want to ensure that the two are distinct. Otherwise they *could* be the same place with different names (heaven forbid!).

If we wish to be more precise we can use a *data type* definition. For example, *Methods* could be defined as

$$Methods ::= Z \mid VDM \mid RAISE \mid B\text{-}Method$$

Here, the type *Methods* may take one of four unique values. More complicated data types are possible. For example, we could define

$$\mathbb{N} ::= zero \mid succ\langle\langle \mathbb{N} \rangle\rangle$$

This is equivalent to defining

$$[\mathbb{N}]$$

$$zero : \mathbb{N}$$
$$succ : \mathbb{N} \rightarrowtail \mathbb{N}$$
$$\vdash$$
$$\{zero\} \cap \operatorname{ran} succ = \emptyset$$
$$\{zero\} \cup \operatorname{ran} succ = \mathbb{N}$$

Functions such as *succ* here are known as *constructors*. Such complications are often not needed for many specifications. You are referred to Section 3.10 starting on page 82 of [381] for more information on such *free type* definitions.

Question: What is the type of $\{i, j : \mathbb{N} \mid i < j\}$?

3.5.3 Sequences

Lists, arrays, files, sequences, trace histories are all different names for a single important data type. The important characterization is that the elements are indexed and normally contiguously numbered.

Consider how we might store, for any pair $(A, B) \in Places \times Places$, the least cost paths from *A* to *B*, (i.e., the relation $(A, B) \pfun path$). A common way to do this is

to store it as a sequence, say under key (A, B), ordered lexically to aid in fast access. This is an example of *data* refinement. A sequence has a 1^{st} element, a 2^{nd} element, 3^{rd} element, etc... Sequence elements are numbered from 1 rather than 0 in Z since this is usually more natural. If T is a set, we define the set of (finite) sequences with elements of type T as follows:

$$\operatorname{seq} T == \{s : \mathbb{N} \pfun T \mid \operatorname{dom} s = 1 \mathinner{..} \# s\}$$

This denotes the set of (partial) functions from $\mathbb{N}$ to T whose domain is a finite segment $1 \mathinner{..} k$ of $\mathbb{N}$. '..' denotes a number range as defined previously:

$$a \mathinner{..} b = \{n : \mathbb{N} \mid a \leq n \leq b\}$$

We call such sequences T-valued sequences, or T sequences. Note that sequences must have a finite (although arbitrary) length. If $s \in \operatorname{seq} T$, the length of s is simply the cardinality, $\# s$, of s considered as a function. The empty sequence, $s = \emptyset$ has $\# s = 0$ and is normally written as

$\langle\rangle$ – the empty sequence

Like the empty set $\emptyset$, the empty sequence is typed.

If non-empty sequences are required, we use the notation

$$\operatorname{seq}_1 T == \operatorname{seq} T \setminus \{\langle\rangle\}$$

If injective sequences (i.e., sequences which contain no repetitions of elements in their range) are needed, we use

$$\operatorname{iseq} T == \operatorname{seq} T \cap (\mathbb{N} \pinj T)$$

The sequence containing just one element, $s = \{1 \mapsto x\}$, has $\# s = 1$ and is written as

$\langle x \rangle$ – a singleton sequence

In general the sequence $\{1 \mapsto x_1, 2 \mapsto x_2, \ldots, n \mapsto x_n\}$ is written in a shorthand form as

$\langle x_1, x_2, \ldots, x_n \rangle$ – a multi-element sequence

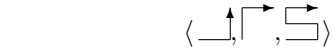
Here are some examples of sequences using this notation:

$\langle\, 11, 29, 3, 7 \,\rangle \in \operatorname{seq} \mathit{primes}$

Some of the prime numbers, in no particular order. Perhaps they are ready to be sorted into numerical order.

$\langle\, \mathrm{J}, \mathrm{O}, \mathrm{N}, \mathrm{A}, \mathrm{T}, \mathrm{H}, \mathrm{A}, \mathrm{N} \,\rangle \in \operatorname{seq} \mathit{CHAR}$

A string of characters. Note that unlike for sets enumerated using the (similar) $\{a, b, c, \ldots\}$ notation, the two occurrences of 'N' are each distinct. The two 'N' elements of the set are in fact the maplets $3 \mapsto \mathrm{N}$ and $8 \mapsto \mathrm{N}$ which form different elements in the set representing the sequence. The same applies to the two occurrences of 'A'.

$\langle$  $\rangle \in \operatorname{seq} \mathit{Path}$

A list of possible paths between A and B.

The length (and thus cardinality) of the three sequences above is 4, 7 and 3 respectively.

Note that unlike sets, sequences *can* have several repeated elements. E.g.:

$$\langle Emma \rangle \neq \langle Emma, Emma, Emma \rangle$$

Also, the order is significant:

$$\langle Alice, Emma \rangle \neq \langle Emma, Alice \rangle$$

Concatenation

We concatenate sequences $s, t \in \operatorname{seq} T$ (or append t to the end of s) by

$$s \frown t \quad \text{– the function } 1 \mathinner{..} (\#s + \#t) \to T \text{ with elements } j \mapsto \begin{cases} s(j) & \text{if } 1 \leq j \leq \#s \\ t(j - \#s) & \text{if } \#s < j \leq (\#s + \#t) \end{cases}$$

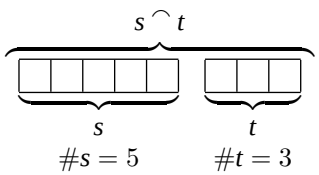
More formally, we could define concatenation as

$$s \frown t = s \cup (\_ - \#s) \fatsemi t$$

For another equivalent definition, see page 116 of [381].

Consider the concatenation of two sequences of length 5 and 3:

Diagrammatically:



Note that $\#(s \frown t) = \#s + \#t = 5 + 3 = 8$. Here is an examples of concatenation of sequences:

$$\langle \mathrm{A} \rangle \frown \langle \mathrm{L}, \mathrm{I}, \mathrm{C}, \mathrm{E} \rangle = \langle \mathrm{A}, \mathrm{L} \rangle \frown \langle \mathrm{I}, \mathrm{C}, \mathrm{E} \rangle = \langle \mathrm{A}, \mathrm{L}, \mathrm{I}, \mathrm{C}, \mathrm{E} \rangle$$

So $\langle a, b, c, \ldots \rangle$ is effectively shorthand for $\langle a \rangle \frown \langle b \rangle \frown \langle c \rangle \frown \ldots$

Laws

$$\langle \rangle \frown s = s \frown \langle \rangle = s$$
$$r \frown (s \frown t) = (r \frown s) \frown t$$
$$(r \frown s = r \frown t) \Rightarrow s = t$$

Prefix

Note that for $s, t \in \operatorname{seq} T$, $s \subseteq t$ is equivalent to $\exists r : \operatorname{seq} T \bullet s \frown r = t$. Thus $\subseteq$ applied to a pair of sequences effectively checks that the left hand sequence is a prefix of the right hand sequence. For example:

$\langle \mathrm{M}, \mathrm{A} \rangle \subseteq \langle \mathrm{M}, \mathrm{A}, \mathrm{N} \rangle$	(Freud's theorem!)
$\langle \rangle \subseteq s$	$\langle \rangle$ is always a prefix of any sequence.
$s \subseteq s$	A sequence is always a prefix of itself.

Laws

If one sequence is the prefix of another and vice versa, the two sequences are identical:

$$(s \subseteq t \land t \subseteq s) \Rightarrow s = t$$

If a sequence is the prefix of a sequence which is the prefix of yet another sequence, then the first sequence is also a prefix of this other sequence:

$$(r \subseteq s \land s \subseteq t) \Rightarrow r \subseteq t$$

If two sequences are a prefix of another sequence, then one of the two sequences must be a prefix of the other. Note that these laws also apply to sets:

$$(r \subseteq t \land s \subseteq t) \Rightarrow (r \subseteq s \lor s \subseteq r)$$

In Z, the relation s prefix t, defined in the Z toolkit (page 119 of [381]) can also be used to check for a sequence prefix.

Other sequence operations

It is often useful to be able to extract the first or last element from a sequence in specifications. The rest of the sequence may also be of interest. Four functions are available for these operations. If $s \in \operatorname{seq} T$ and $s \neq \langle\rangle$ (i.e., $s \in \operatorname{seq}_1 T$):

head s	$= s(1)$	●○○○○	first element of sequence
last s	$= s(\#s)$	○○○○●	last element of sequence
tail s	$= succ \fatsemi (\{1\} \lhd s)$	○●●●●	sequence without first element
front s	$= \{\#s\} \lhd s$	●●●●○	sequence without last element

It is best to avoid applying these functions to an empty sequence; for example, *head*$\langle\rangle$ is undefined.

As an example, if the sequence s is $\langle \mathrm{C}, \mathrm{O}, \mathrm{D}, \mathrm{E} \rangle$ (which is $\{1 \mapsto \mathrm{C}, 2 \mapsto \mathrm{O}, 3 \mapsto \mathrm{D}, 4 \mapsto \mathrm{E}\}$ then

$$\begin{aligned} \textit{head } s &= \mathrm{C} \\ \\ \textit{last } s &= \mathrm{E} \\ \\ \textit{tail } s &= \{0 \mapsto 1, 1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 4, \ldots\} \fatsemi \{2 \mapsto \mathrm{O}, 3 \mapsto \mathrm{D}, 4 \mapsto \mathrm{E}\} \\ &= \{1 \mapsto \mathrm{O}, 2 \mapsto \mathrm{D}, 3 \mapsto \mathrm{E}\} \\ &= \langle \mathrm{O}, \mathrm{D}, \mathrm{E} \rangle \\ \\ \textit{front } s &= \{4\} \lhd \{1 \mapsto \mathrm{C}, 2 \mapsto \mathrm{O}, 3 \mapsto \mathrm{D}, 4 \mapsto \mathrm{E}\} \\ &= \{1 \mapsto \mathrm{C}, 2 \mapsto \mathrm{O}, 3 \mapsto \mathrm{D}\} \\ &= \langle \mathrm{C}, \mathrm{O}, \mathrm{D} \rangle \end{aligned}$$

Exercise: State some laws concerning these operations. Try not to refer to [381]!

We could construct more general versions of *front* and *tail* if required for a particular application, allowing the length of sequence required to be specified, using *generic construction*:

$$
\begin{array}{|l}
[T] \\
\_ \underline{for} \_, \\
\_ \underline{after} \_ : (\operatorname{seq} T) \times \mathbb{N} \rightarrow (\operatorname{seq} T) \\
\hline
\forall s : \operatorname{seq} T;\ n : \mathbb{N} \bullet \\
\quad s \underline{for}\ n = (1 \mathinner{..} n) \lhd s \ \land \\
\quad s \underline{after}\ n = (\{0\} \lhd \mathit{succ}^n) \fatsemi s \\
\hline
\end{array}
$$

For example, these could be useful in extracting portions of files considered as a sequence of bytes. Note that for $s \in \operatorname{seq}_1 T$,

$$\mathit{front}\ s = s \underline{for}\ (\#s - 1)$$

$$\mathit{tail}\ s = s \underline{after}\ 1$$

Here are some laws:

$$
\begin{aligned}
s \underline{for}\ 0 &= \langle\rangle \\
s \underline{for}\ \#s &= s \\
s \underline{after}\ 0 &= s \\
s \underline{after}\ \#s &= \langle\rangle
\end{aligned}
$$

Reversal

If $s \in \operatorname{seq} T$, the reverse of s is given by *rev* s. For example, the sequence, $\langle \mathrm{D}, \mathrm{O}, \mathrm{G} \rangle$ is converted to $\langle \mathrm{G}, \mathrm{O}, \mathrm{D} \rangle$ using the *rev* function!

Laws

$$
\begin{aligned}
\mathit{rev}\langle\rangle &= \langle\rangle \\
\mathit{rev}\langle x \rangle &= \langle x \rangle \\
\mathit{rev}(\mathit{rev}\ s) &= s \\
\mathit{rev}(s \frown t) &= (\mathit{rev}\ t) \frown (\mathit{rev}\ s)
\end{aligned}
$$

For example

$$\mathit{rev}(\langle a, b \rangle \frown \langle c, d \rangle) = \langle d, c \rangle \frown \langle b, a \rangle$$

Distributed operations

Sometimes the concatenation of a sequence of sequences is useful:

$$\frown / \langle\rangle = \langle\rangle$$

$$\frown / \langle a, b, \ldots, n \rangle = a \frown b \frown \ldots \frown n$$

More formally, $\frown / : \operatorname{seq}(\operatorname{seq} T) \rightarrow \operatorname{seq} T$ satisfies

$$\frown / (\langle a \rangle \frown s) = \langle a \rangle \frown (\frown / s)$$

and also

$$\frown/(s \frown \langle a \rangle) = (\frown/ s) \frown \langle a \rangle$$

For the actual formal definition of $\frown/$, see page 121 of [381].

Question: What is $rev(\frown/\langle\langle \mathrm{N}, \mathrm{A}, \mathrm{H}\rangle, \langle \mathrm{T}, \mathrm{A}\rangle, \langle \mathrm{N}, \mathrm{O}\rangle, \langle \mathrm{J}\rangle\rangle)$?

Other distributed operators can also be defined for sequences if required for a particular specification. For example, a session of updating a database results in a sequence of partial functions. We could define a distributed overriding operator in terms of the standard dyadic overriding operator to generalize the overriding of a relation. See for example, a definition on page 172. Informally:

$$\oplus/\langle a, b, \ldots, n\rangle = a \oplus b \oplus \ldots \oplus n$$

Exercise: The composition operator '$\fatsemi$' may also be generalized to a distributed composition operator '$\fatsemi/$'. Again, informally:

$$\fatsemi/\langle a, b, \ldots, n\rangle = a \fatsemi b \fatsemi \ldots \fatsemi n$$

Write a formal definition for this. Can you think of an application for this operator in a specification?

Disjointness and partitioning

A sequence of sets is considered 'disjoint' if none of the sets in its range intersect. Formally:

$$\forall S : \operatorname{seq} \mathbb{P} T \bullet \\ \quad \mathsf{disjoint}\ S \Leftrightarrow (\forall i, j : \operatorname{dom} S \mid i \neq j \bullet (S\, i) \cap (S\, j) = \emptyset)$$

The empty and singleton sequences of sets are always disjoint since there are no two distinct elements to consider. I.e., $\mathsf{disjoint}\ \langle\rangle$ and $\mathsf{disjoint}\ \langle x \rangle$ are always true.

For a two-element sequence, $\mathsf{disjoint}\ S$ is equivalent to:

$$S(1) \cap S(2) = \emptyset$$

For a three-element sequence, $\mathsf{disjoint}\ S$ is

$$S(1) \cap S(2) = \emptyset \land S(2) \cap S(3) = \emptyset \land S(3) \cap S(1) = \emptyset$$

and so on.

We can extend this idea to consider the generalized union of all the sets in the range of the sequence. If it is equal to a particular set, the sequence is said to 'partition' that set. Formally:

$$\forall S : \operatorname{seq} \mathbb{P} T;\ P : \mathbb{P} T \bullet \\ \quad S\ \mathsf{partition}\ P \Leftrightarrow (\ \mathsf{disjoint}\ S \land P = \bigcup \{ i : \operatorname{dom} S \bullet S\, i \}\ )$$

Disjointness and partitioning may be generalized from sequences to any indexed set (i.e., $S : I \pfun \mathbb{P} T$ instead of $S : \operatorname{seq} \mathbb{P} T$).

3.5.4 Orders

The Z toolkit can be extended as required for a particular application. For example, a partial order, which is reflexive, antisymmetric, and transitive, may be a useful concept in some specifications:

$$
\begin{array}{l}
partial\_order[X] == \\
\quad \{ \_ \underline{R} \_ : X \leftrightarrow X \mid (\forall x, y, z : X \bullet \\
\qquad x \underline{R} x \land \\
\qquad (x \underline{R} y \land y \underline{R} x) \Rightarrow x = y \land \\
\qquad (x \underline{R} y \land y \underline{R} z) \Rightarrow x \underline{R} z) \}
\end{array}
$$

This can be extended to define a total order, in which all elements are related:

$$
\begin{array}{l}
total\_order[X] == \\
\quad \{ \_ \underline{R} \_ : partial\_order[X] \mid (\forall x, y : X \bullet \\
\qquad x \underline{R} y \lor y \underline{R} x) \}
\end{array}
$$

For example, it may be useful to model time as a total order in a particular specification.

3.5.5 Summary

We have briefly covered numbers and then used these to define the notion of sequences, an important data structure in many specifications and data refinement. We have also looked at ways in which Z lets us manipulate sequences.

This concludes the introduction to the mathematical notation of Z. Some lesser used parts of Z (such as *bags*) have not been covered here, but are included from page 124 onwards in [381]. As you gain confidence in writing Z specifications, you should investigate these other features of Z.

Using mathematics for specification is all very well for small examples, but for more realistically sized problems, things start to get out of hand. To deal with this, Z includes the *schema* notation to aid the structuring and modularization of specifications. We have seen an example of a schema box in passing. In the next section we shall see how to combine such schemas to produce larger specifications.

3.6 Schemas

A boxed notation called 'schemas' is used for structuring Z specifications. This has been found to be necessary to handle the information in a specification of any size. We saw a brief example on page 32. Here is another example of a schema:

Book

$$
\begin{array}{l}
author : People \\
title : \operatorname{seq} CHAR \\
readership : \mathbb{P}\, People \\
rating : People \pfun 0 \,..\, 10 \\
\hline
readership = \operatorname{dom} rating
\end{array}
$$

This defines a single *author*, a book *title*, and a number of people who make up the *readership* of a book (a *set* of people, as indicated by the 'power set' operator $\mathbb{P}$ in the type declaration), together with their *rating* of the book (out of 10). The bottom half of a schema optionally introduces extra constraints between the variables in the form of predicates. Here, the *readership* is the same as the domain of the *rating* function.

The top half of the *Book* schema box defines a number of named variables with associated constraints from which the type information can be mechanically derived. Here $\mathrm{seq}\, CHAR$ is a subset of $\mathbb{N} \pfun CHAR$ (see the definition of seq on page 115 of [381]), which itself is a subset of $\mathbb{Z} \rel CHAR$. This is the same as $\mathbb{P}(\mathbb{Z} \times CHAR)$ (see page 95 of [381]). The expression $0 \upto 10$ implies a type of integer $\mathbb{Z}$ with the extra constraint that the *rating* must take a value between 0 and 10. In fact this schema may be 'normalized' into the following form:

Book

$$
\begin{array}{l}
author : People \\
title : \mathbb{P}(\mathbb{Z} \times CHAR) \\
readership : \mathbb{P}\, People \\
rating : \mathbb{P}(People \times \mathbb{Z}) \\
\hline
title \in \mathrm{seq}\, CHAR \\
rating \in People \pfun 0 \upto 10 \\
readership = \mathrm{dom}\, rating
\end{array}
$$

Note that the predicates on separate lines in the second half of the schema above are conjoined together by default. Here, the declarations use the most general types possible for the components. These are the actual types for these components, which could be used for type-checking purposes if required. Mentally calculating such types can be a useful aid in understanding a specification.

Schemas are primarily used to specify state spaces and operations for the mathematical modelling of systems. For example, here is a schema called *StateSpace*:

StateSpace

$$
\begin{array}{l}
x_1 : S_1 \\
x_2 : S_2 \\
\quad \vdots \\
x_n : S_n \\
\hline
Inv(x_1, \ldots, x_n)
\end{array}
$$

This can also be written as follows to save space if desired:

StateSpace

$$
\begin{array}{l}
x_1 : S_1 ; \ \ldots \ ; \ x_n : S_n \\
\hline
Inv(x_1, \ldots, x_n)
\end{array}
$$

Even more space can be saved using a horizontal formal of schema definition, which is typically used if the entire definition can conveniently fit on a single line. E.g.:

$$StateSpace \mathrel{\widehat=} [x_1 : S_1 ; \ \ldots \ ; \ x_n : S_n \mid Inv(x_1, \ldots, x_n)]$$

This schema specifies a *state space* in which $x_1, \ldots, x_n$ are the state variables and $S_1, \ldots, S_n$ are expressions from which their types may be systematically derived. Z types are sets – $x_1, \ldots, x_n$ should not occur free in $S_1, \ldots, S_n$, or if they do, they refer instead to other occurrences of these variables already in scope (e.g., globally defined variables). $Inv(x_1, \ldots, x_n)$ is the state *invariant*, relating the variables in some way for all possible allowed states of the system during its lifetime.

Note that unlike in an ordered tuple, the variables in a schema are essentially unordered – reordering them would give the same schema – and the variable names do not come into scope until the bottom half of the schema. Thus any interdependencies must be defined here. (A common mistake by initial users of Z is to define interdependencies in the declaration part, but a type-checker will very quickly detect this.)

3.6.1 Example specification

The 'Birthday Book' is a well known example from Chapter 1 of Spivey's widely used book on Z [381]. It is a system for recording birthdays. It uses the following basic types (or given sets):

$$[NAME, DATE]$$

The *state space* of the Birthday Book is specified by:

BirthdayBook

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$

$$known = \mathrm{dom}\, birthday$$

The *state variables* are *known* (a number of people's names) and *birthday* (unique dates associated with each known person's name). The 'invariant' property of this schema is:

$$known = \mathrm{dom}\, birthday$$

I.e., every known person has a birth date associated with them.

Z makes use of identifier decorations to encode intended interpretations. A state variable with no decoration represents the current (before) state and a state variable ending with a prime ($'$) represents the next (after) state. A variable ending with a question mark (?) represents an input and a variable ending with an exclamation mark (!) represents an output. A typical schema specifying a *state change* is the following *operation schema*:

Operation

$$x_1 : S_1;\ \ldots\ ;\ x_n : S_n$$
$$x'_1 : S_1;\ \ldots\ ;\ x'_n : S_n$$
$$i_1? : T_1;\ \ldots\ ;\ i_m? : T_m$$
$$o_1! : U_1;\ \ldots\ ;\ o_p! : U_p$$

$$Pre(i_1?, \ldots, i_m?, x_1, \ldots, x_n)$$
$$Inv(x_1, \ldots, x_n)$$
$$Inv(x'_1, \ldots, x'_n)$$
$$Op(i_1?, \ldots, i_m?, x_1, \ldots, x_n, x'_1, \ldots, x'_n, o_1!, \ldots, o_p!)$$

The inputs are $i_1?, \ldots, i_m?$; the outputs are $o_1!, \ldots, o_p!$; the precondition is:

$$Pre(i_1?, \ldots, i_m?, x_1, \ldots, x_n)$$

The state change $(x_1, \ldots, x_n)$ to $(x'_1, \ldots, x'_n)$ is specified by:

$$Op(i_1?, \ldots, i_m?, x_1, \ldots, x_n, x'_1, \ldots, x'_n, o_1!, \ldots, o_p!)$$

Note that before and after states which are not constrained may take any allowed value. Thus, unlike most programming languages, after states are unconstrained by their matching before states unless explicitly stated to do so. Thus it is necessary to include the predicate $x'_1 = x_1$ if x_1 is to retain the same value after the operation. This convention can be confusing to some programmers, but is extremely useful in specifications. If required, there is a convention for constraining a number of state components to remain the same, as we shall see shortly.

As a more specific example of a operation schema, consider the adding of a birthday to the birthday book:

AddBirthday

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$
$$known' : \mathbb{P}\, NAME$$
$$birthday' : NAME \pfun DATE$$
$$name? : NAME$$
$$date? : DATE$$

$$name? \notin known$$
$$known = \mathrm{dom}\, birthday$$
$$known' = \mathrm{dom}\, birthday'$$
$$birthday' = birthday \cup \{name? \mapsto date?\}$$

The entire state with its invariant is repeated for both the before (undashed) and after (dashed) states. Fortunately schemas may be 'included' in other schemas, so this may be written much more concisely than above in a real specification, reducing six lines to a single included schema in the specification above in Spivey's original specification (see page 4 of [381]).

The precondition of *AddBirthday* is $name? \notin known$. The operation part of the specification is the predicate

$$birthday' = birthday \cup \{name? \mapsto date?\}$$

which specifies that in the state after the operation *AddBirthday* is performed, the new value ($birthday'$) of the state variable *birthday* is $birthday \cup \{name? \mapsto date?\}$.

Z schemas can be specified using other schemas with the Δ and Ξ conventions when specifying operations that respectively change the state or leave the state unchanged. Given a state space schema:

StateSpace

$$x_1 : S_1; \ \ldots \ ; \ x_n : S_n$$

$$Inv(x_1, \ldots, x_n)$$

then the schema:

Operation

$$\Delta StateSpace$$
$$i_1? : T_1; \ \ldots \ ; \ i_m? : T_m$$
$$o_1! : U_1; \ \ldots \ ; \ o_p! : U_p$$

$$Pre(i_1?, \ldots, i_m?, x_1, \ldots, x_n)$$
$$Op(i_1?, \ldots, i_m?, x_1, \ldots, x_n, x'_1, \ldots, x'_n, o_1!, \ldots, o_p!)$$

abbreviates:

Operation

$$x_1 : S_1; \ \ldots \ ; \ x_n : S_n$$
$$x'_1 : S_1; \ \ldots \ ; \ x'_n : S_n$$
$$i_1? : T_1; \ \ldots \ ; \ i_m? : T_m$$

$$Pre(i_1?, \ldots, i_m?, x_1, \ldots, x_n)$$
$$Inv(x_1, \ldots, x_n)$$
$$Inv(x'_1, \ldots, x'_n)$$
$$Op(i_1?, \ldots, i_m?, x_1, \ldots, x_n, x'_1, \ldots, x'_n, o_1!, \ldots, o_p!)$$

This use of one schema (Δ*StateSpace* here) within another schema is called *schema inclusion*, and is a useful and widely use structuring technique in Z. It adds all the state components and the associated constraining predicates to the including schema. If any component names match those already declared elsewhere, then their types must be compatible and they map in top of each other. This can be useful for sharing of components in the specification. The use of schema inclusion allows detailed declarations of state components to be hidden in subsequent parts of a specification once they have been declared and explained beforehand.

As a more concrete example of schema inclusion, *AddBirthday* can be specified using $\Delta BirthdayBook$:

AddBirthday

$$\Delta BirthdayBook$$
$$name? : NAME$$
$$date? : DATE$$

$$name? \notin known$$
$$birthday' = birthday \cup \{name? \mapsto date?\}$$

This abbreviates the version of the *AddBirthday* schema presented previously.

Some operations access the state, but do not change it (e.g., status operations, returning an output depending on the state). For example:

Operation

$$x_1 : S_1;\ \ldots\ ;\ x_n : S_n$$
$$x'_1 : S_1;\ \ldots\ ;\ x'_n : S_n$$
$$i_1? : T_1;\ \ldots\ ;\ i_m? : T_m$$
$$o_1! : U_1;\ \ldots\ ;\ o_p! : U_p$$

$$Pre(i_1?, \ldots, i_m?, x_1, \ldots, x_n)$$
$$Inv(x_1, \ldots, x_n)$$
$$(x'_1 = x_1 \land \ldots \land x'_n = x_n)$$
$$Op(i_1?, \ldots, i_m?, x_1, \ldots, x_n, o_1!, \ldots, o_p!)$$

Strictly, $Inv(x'_1, \ldots, x'_n)$ is also included as a predicate, but this is redundant because of the second and third predicate clauses above.

An example of such a status operation is *FindBirthday*, which looks up a birthday for a given name:

FindBirthday

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$
$$known' : \mathbb{P}\, NAME$$
$$birthday' : NAME \pfun DATE$$
$$name? : NAME$$
$$date! : DATE$$

$$name? \in known$$
$$known = \mathrm{dom}\, birthday$$
$$known' = known$$
$$birthday' = birthday$$
$$date! = birthday(name?)$$

The precondition is $name? \in known$ and the the operation part of the predicate is $date! = birthday(name?)$.

Ξ-inclusion is used to compactly specify such operations. This is similar to Δ-inclusion, together with the extra constraint that the state components do not change their values between the before and after states.

Consider a state space schema:

StateSpace

$x_1 : S_1;\ \ldots\ ;\ x_n : S_n$

$Inv(x_1, \ldots, x_n)$

Then the schema operation

Operation

$\Xi StateSpace$

$i_1? : T_1;\ \ldots\ ;\ i_m? : T_m$

$o_1! : U_1;\ \ldots\ ;\ o_p! : U_p$

$Pre(i_1?, \ldots, i_m?, x_1, \ldots, x_n)$

$Op(i_1?, \ldots, i_m?, x_1, \ldots, x_n, x'_1, \ldots, x'_n, o_1!, \ldots, o_p!)$

abbreviates the following expanded schema:

Operation

$x_1 : S_1;\ \ldots\ ;\ x_n : S_n$

$x'_1 : S_1;\ \ldots\ ;\ x'_n : S_n$

$i_1? : T_1;\ \ldots\ ;\ i_m? : T_m$

$o_1! : U_1;\ \ldots\ ;\ o_p! : U_p$

$Pre(i_1?, \ldots, i_m?, x_1, \ldots, x_n)$

$Inv(x_1, \ldots, x_n)$

$(x'_1 = x_1 \land \ldots \land x'_n = x_n)$

$Op(i_1?, \ldots, i_m?, x_1, \ldots, x_n, x'_1, \ldots, x'_n, o_1!, \ldots, o_p!)$

For example, *FindBirthday* can be specified using Ξ*BirthdayBook*:

FindBirthday

$\Xi BirthdayBook$

$name? : NAME$

$date! : DATE$

$name? \in known$

$date! = birthday(name?)$

This just abbreviates the previous version of *FindBirthday*.

3.6.2 Schema operators

There are schema operators matching the logical connectives on predicates, such as $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$, as well as quantification using $\forall$, $\exists$ and $\exists_1$. The schemas are first normalized. For the binary connectives to be used, there must be no conflicting declarations in the schemas being combined. For schema negation, the normalization is particularly important to ensure any hidden predicate constraints in the declarations are also negated. For binary operators, the declarations are merged in the resulting schema (remember that the order of the declarations is not important) and the predicate parts are combined depending on the operation involved. See [200] for a discussion of the issues involved, and pages 32 to 34 of [381] for further explanation and examples.

It is possible to declare a schema as a type (e.g., *state* : *StateSpace*). If such a declaration is in scope then a component may be selected from the schema. For example $state.x_1$ would return the component x_1, and so on.

It is also useful to return a schema tuple,tuple:of a schema which is similar to an ordered tuple, except the components are unordered, and named instead. The notation $\theta StateSpace$ is used to return the schema tuple for the *StateSpace*, for example. All the named components $x_1, \ldots, x_n$ are included in the schema tuple. As an example, a schema following the Ξ convention may be defined as follows:

$$\Xi StateSpace \mathrel{\widehat=} [\Delta StateSpace \mid \theta StateSpace' = \theta StateSpace]$$

One or more components of a schema may be hidden (i.e., existentially quantified) using a *schema hiding*. For example, $StateSpace \setminus (x_1, x_2)$ is the same as $\exists x_1 : S_1;\ x_2 : S_2 \bullet StateSpace$.

Conversely, schema components can be projected using components defined by a second schema using *schema projection*. For example, if $ProjectSpace \mathrel{\widehat=} [x_1 : S_1;\ x_2 : S_2]$ then $StateSpace \upharpoonright ProjectSpace$ hides all the components of *StateSpace* except x_1 and x_2.

Components of a schema may be renamed using *schema renaming*. For example, $StateSpace[y_1/x_1, y_2/x_2]$ returns a new schema with the x_1 component replaced by y_1 and the x_2 component replaced by y_2. This can be useful if there is a clash of names in a specification for some reason.

Z provides a 'pre' operator which may be used to return the precondition of a schema. pre *Operation* existentially quantifies all after state and output components. I.e.,

$$\exists x_1' : S_1;\ \ldots;\ x_n' : S_n;\ o_1! : T_1;\ \ldots;\ o_p! : T_p \bullet Operation$$

Another more complicated schema operation is *sequential composition* ('$\fatsemi$'). If *Operation*1 and *Operation*2 are two operation schemas such as:

*Operation*1

$$x_1 : S_1;\ \ldots;\ x_p : S_p$$
$$z_1 : U_1;\ \ldots;\ z_n : U_n$$
$$z_1' : U_1;\ \ldots;\ z_n' : U_n$$

$$Op1(x_1, \ldots, x_p, z_1, \ldots, z_n, z_1', \ldots, z_n')$$

and

*Operation*2

$$y_1 : T_1;\ \ldots\ ;\ y_q : T_q$$
$$z_1 : U_1;\ \ldots\ ;\ z_n : U_n$$
$$z'_1 : U_1;\ \ldots\ ;\ z'_n : U_n$$

$$Op2(y_1, \ldots, y_q, z_1, \ldots, z_n, z'_1, \ldots, z'_n)$$

then *Operation*1 ⨾ *Operation*2 is

*Operation*1 ⨾ *Operation*2

$$x_1 : S_1;\ \ldots\ ;\ x_p : S_p$$
$$y_1 : T_1;\ \ldots\ ;\ y_q : T_q$$
$$z_1 : U_1;\ \ldots\ ;\ z_n : U_n$$
$$z'_1 : U_1;\ \ldots\ ;\ z'_n : U_n$$

$$\exists z''_1 : U_1;\ \cdots\ ;\ z''_n : U_n \bullet$$
$$Op1(x_1, \ldots, x_p, z_1, \ldots, z_n, z''_1, \ldots, z''_n) \land$$
$$Op2(y_1, \ldots, y_q, z''_1, \ldots, z''_n, z'_1, \ldots, z'_n)$$

All the $z'_1, \ldots, z'_n$ after states in *Operation*1 which match $z_1, \ldots, z_n$ before states in *Operation*2 are combined and existentially quantified as a new intermediate state $z''_1 \ldots z''_n$. The $x_1, \ldots, x_p$ components in *Operation*1 and the $y_1, \ldots, y_q$ components in *Operation*2 are those do not match in this way (including any input and output components). If any of these components match in a type-compatible way, they are merged as for other standard schema operators such as conjunction.

Thus, *AddBirthday* ⨾ *FindBirthday* expands as:

AddThenFindBirthday

$$
\begin{array}{l}
known : \mathbb{P}\, NAME \\
birthday : NAME \pfun DATE \\
known' : \mathbb{P}\, NAME \\
birthday' : NAME \pfun DATE \\
name? : NAME \\
date? : DATE \\
date! : DATE \\
\hline
\exists\, known'' : \mathbb{P}\, NAME;\ birthday'' : NAME \pfun DATE \bullet \\
\quad known = \mathrm{dom}\, birthday \land \\
\quad known'' = \mathrm{dom}\, birthday'' \land \\
\quad name? \notin known \land \\
\quad birthday'' = birthday \cup \{name? \mapsto date?\} \land \\
\quad known'' = \mathrm{dom}\, birthday'' \land \\
\quad known' = known'' \land \\
\quad birthday' = birthday'' \land \\
\quad name? \in known'' \land \\
\quad date! = birthday''(name?)
\end{array}
$$

There is a similar *schema piping* operator in Z which matches the outputs of the first schema to the inputs of the second schema instead of the state components.

3.6.3 Properties

An example of a simple property that one might want to prove is:

$$AddThenFindBirthday \vdash date! = date?$$

I.e., a *FindBirthday* operation after an *AddBirthday*, with the same name input to both, outputs the same date as that input to the *AddBirthday* operation. This is the sort of property that, if proved, provides an extra level of confidence that a specification is correct, since it confirms our intuitions about the properties of the specifications that we expect to hold. If a given desired property cannot be proved, this may well indicate a flaw in the specification, which can then be rectified at an early stage before any implementation has been started. Using informal design techniques, such errors are not normally discovered until a later stage, such as coding, testing, or even after the system has been delivered, with all the extra costs that this involves.

Note that Z as defined by Spivey [381] includes no standard way to write theorems. In this book, we adopt the convention of writing $\vdash p$ where p is some predicate. Alternatively, $d \vdash p$ is sometimes used to include universally quantified declarations d.

3.7 Conclusion

This chapter has provided an extremely brief introduction to Z. Readers who still have reservations about their understanding of Z would be well advised to read an introduc-

tory textbook on Z before tackling the formal case studies presented in the rest of this book. See page 244 for a list of such books.

The Z notation includes set-theoretic definitions in the form of an extensive mathematical 'toolkit' as comprehensively presented in [381]. Much of Z as it is normally used is defined using itself in this toolkit, and it is forming the main basis for the proposed Z standard [79]. In addition, in may be desirable to create further toolkit libraries for certain applications. For example, the inclusion of real numbers has been considered [411].

It is important to remember than Z is based on *first order* logic. A common mistake of novice Z users is to attempt to form relations and functions on predicates. This is not legal in Z. In fact there is no predefined Boolean type in Z since it is normally unnecessary. It is possible to define a binary valued type in Z if required, but often there is a better way of approaching the specification if a developer finds him/herself tempted to do this, unless it is explicitly needed in a certain application (e.g., see Chapter 8). A Z type-checker will quickly discover any attempt by a specifier to bend the rules of Z and try to use it in a higher order manner, as in HOL [178] for example. However, unfortunately there are some untype-checked Z specifications are incorrect because of this problem.

Z has been a relatively fluid language during its lifetime. The *lack* of tools in the early days of its development was a positive asset since it allowed new ideas to be tried experimentally with little overhead. However as Z has become more established, and used increasingly in industry, the need for a standard notation to allow mechanical tool support has increased. The current *de facto* standard widely used by academics and industrial users alike, is that laid down in Spivey's *Z Notation: A Reference Manual* (often know as the Z Reference Manual, or 'ZRM' for short). This is now in its 2nd edition [381]. This is the notation generally used in this book. The development of an international Z standard is in progress [79]. Once this is accepted it is likely to supersede [381] since tool builders will probably then adopt it as the notation of choice.

This completes the introductory part of the book. As previously mentioned, a comprehensive introduction to Z is impossible in a single chapter. If you still require further grounding in the use of Z, please consult one or more of the numerous Z textbooks available (as listed on page 244) before proceeding with the rest of the book. Subsequent the chapters present a number of case studies of specifications in Z, nearly all of implemented (and hopefully useful!) systems.

II

Network Services

The use of the Z notation to document network services in a formal and precise manner is presented. In Chapter 4, a general introduction to User and Implementor Manuals is given, using a simple service as an example. A manual for a more substantial file system service is included in Chapter 5.

Chapter 4

Documentation using Z

The Z notation has been applied to the formal specification of resource managers or 'services' within a distributed system. A formal description gives a more precise understanding of the behaviour of a service and, when combined with informal text, is sufficiently readable for the specification to be used for documentation purposes. Two types of manual have been produced, one presenting the external view of the service (for users) and another describing the internal view (for implementors). A common service framework deals with standard aspects of services. Parts of a simple User and Implementor Manual are presented as an example.

4.1 Introduction

The aim of this chapter is to introduce and illustrate a formal style of specifying and documenting system components. The components considered in the study are services within a distributed computing system [57].

First an introduction to the style of specification is presented. It considers how a service can be modelled in the Z specification language, and the way in which such specifications can be used in documenting both the user's and the implementor's view of a service. Next an example of a service specification is given. This example describes a very simple service, the Reservation Service, and serves to demonstrate the style of User Manual developed by the project. Additionally a simple example of an implementation-oriented specification in the form of an Implementor Manual for the Reservation Service is presented. Finally some experience gained from the work is discussed.

More complex services have been documented and implemented but these are not presented here to keep the length to a manageable size. Chapter 5 presents a single more realistically sized manual.

The Reservation Service was developed as a part of the Distributed Computing Software (DCS) Project at the Programming Research Group within the Oxford University Computing Laboratory. The goal of the project was to investigate the use of formal techniques in the design and documentation of services for a loosely-coupled distributed operating system, based on the model of autonomous clients having access to a number of shared facilities. Several services have been designed and documented in the manner described here, and implementations of them have been provided for, and used by, members of the Programming Research Group.

A number of monographs were produced by the project [55, 56, 172]. These provide more information for those interested in the details of the work of the project. In particular, the User and Implementor Manuals for the Reservation Service and a (much larger) Block Storage Service, together with the Common Service Framework, covering standard facilities such as authentication and accounting, are presented in their entirety.

4.2 Motivation

It is fundamental to the design of any complex artefact, and of computer systems in particular, that an appropriate means of describing and communicating the design is used.

A very important line of communication is between the designer and the user of the system. It is only if this communication is accomplished satisfactorily that the designer can have any expectation of meeting the requirements of the user and, likewise, the user have any expectation of being able to make proper use of the finished product.

No less important is the communication from the designer to the implementor of the system. This is necessary to ensure that the finished product does indeed have the characteristics that the designer specified.

The aim of the work described here is the improved communication between designer, user and implementor which can be achieved by the use of formal specification in the design and documentation of computer systems.

4.2.1 Formal specification

Satisfactory communication relies firstly on the production of an unambiguous description. If a description is sufficiently precise, it can act as a contract between the designer, user and implementor, to ensure that they agree on what is to be provided.

A fundamental objective of the Distributed Computing Software Project has been to make use of mathematical techniques for program specification to assist the design, development and presentation of distributed system services.

The formal notation Z was used throughout the project. The Distributed Computing Software Project tested the application of the theoretical ideas behind Z to a realistic and practical system. As a result of this, the project was influential in the development of notational techniques which have now become a standard part of the Z style of specification.

The use of formal specification techniques, because of their rigour, tends to guide designs towards the conceptually simple. This has the advantage of making the designs easier to understand, but the possible disadvantage of making them harder to implement efficiently, since the simplest ideas do not necessarily have the most straightforward realization.

Formal techniques encourage a level of abstraction that is important in avoiding the introduction of unnecessary implementation bias into designs. In the initial design, implementation bias simply restricts the range of possible implementations. It is usually an indication that the designer allowed unnecessary knowledge of a potential implementation to become visible at the user level.

4.2.2 Documentation

Conventionally, various pieces of documentation are the main means of communication between designer and user. In order that the rigour of the specifications should not be lost, it was felt to be of great importance that the system documentation should incorporate the full formalism used in the design. However, it was also important to ensure that, as for any documentation, readability and accessibility were not sacrificed in the process.

A significant amount of effort has therefore been spent on developing a manual style which combines informal and formal text. The presentation of the User Manuals emphasizes the effect of each user-invoked operation on a service. The Implementor Manuals, on the other hand, concentrate on identifying the subcomponents from which an implementation of the service can be built.

4.3 Service Specification

A service of a distributed computing system can be modelled in much the same way as a component in a centralized system.

A service can be described in terms of a service state and a set of operations which will change the state in a well-defined way. Consider a service with a state S. The effect on the service of a given operation OP can be described in terms of the preceding state S and the subsequent state S'. (The dash is used by convention in Z to denote the state after an operation.) Thus, at any given time, the current state of the service can be determined from knowledge of the initial service state and of the sequence of operations executed in the lifetime of the service so far.

Two small but significant differences can exist in a distributed system, as compared to a centralized system. The first is that the individual services will usually be at least partly involved in tasks such as accounting, user authentication and access control, which would be more easily separable in a centralized system. Secondly, it is a characteristic feature of a distributed system that components in the system may continue to work after others have failed, so that the error notification and handling provided by services becomes important.

4.3.1 User's view

A user will in general be interested only in the externally observable behaviour of a service. In the case of a file storage service, for instance, a user will be concerned with files, file names and file contents, but will not be interested in details of how these items are represented and stored by the service. When specifying the requirements and the user interface for a service, it is useful to do so in terms of an abstract (i.e., not implementation specific) service state and corresponding abstract operations.

If the user's view of the (abstract) service state is AS, then each abstract operation will be described in terms of the preceding and subsequent abstract states AS and AS'. In order for the state of the service to be defined at all times, the initial state of the service $InitAS$ also needs to be established.

4.3.2 Implementor's view

Unlike a user, an implementor will need a much more detailed view of a service and will specifically be interested in the internal behaviour of the service. In the case of a file storage service for instance, the implementor will have to deal with items such as index blocks and data blocks.

If the implementor's view of the (concrete) service state is *CS*, then concrete operations are expressed in terms of the before and after states *CS* and CS'. As before, the initial state of the concrete service *InitCS* must be well-defined.

4.3.3 Common framework

In a distributed system consisting of a number of separate services connected by a network, it is useful for the services to have certain characteristics in common. These will include such facilities as service access, user authentication, accounting, accumulation of statistics and error reporting. Making the provision of such facilities the same across the collection of services means that the system as a whole will appear more homogeneous to the user and therefore easier to use. Also, the specification and implementation of the services becomes simpler since some parts are common to all services.

These common aspects of services have been collected together into a set of definitions known as the Common Service Framework. When required, these definitions can be incorporated into specifications of individual services in a standard way.

4.3.4 Correctness of implementation

In order to verify that the implementor's view of a service is compatible with the user's view of the same service, formal correctness arguments can be used.

These arguments depend on giving a formal definition of how the concrete and abstract representations of the service state relate to each other. In the following, *Rel* denotes the relation between *CS* and *AS*, and Rel' the same relation between CS' and AS'.

In order for the concrete service state to be capable of representing the state of the abstract service, it needs to have at least one concrete state for each possible abstract state:

$$\forall AS \bullet \exists CS \bullet Rel$$

The initial concrete service state must specifically represent the initial abstract service state:

$$\forall CS' \bullet InitCS \Rightarrow (\exists AS' \bullet InitAS \land Rel')$$

Note that in this book it is assumed that the initial state (defined in *InitAS* and *InitCS* above) is dashed since it is the state *after* initialization.

For each abstract operation *AOP*, we must supply a corresponding concrete operation *COP* which is applicable in the corresponding domain to the abstract operation and which will produce a result that satisfies the abstract specification. In other words, if *AOP* changes the abstract state from *AS* to AS', then the corresponding concrete

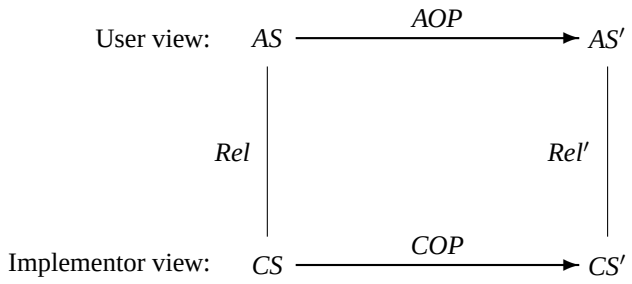


Figure 4.1 Relationship between abstract and concrete views.

operation *COP* must, given an initial state *CS* which relates to *AS* according to *Rel*, produce a new state *CS′* which relates to *AS′* according to *Rel′*. This can be expressed more formally as:

$$\forall AS;\ CS \bullet \mathrm{pre}\ AOP \land Rel \Rightarrow \mathrm{pre}\ COP$$

$$\forall AS;\ CS;\ CS' \bullet \mathrm{pre}\ AOP \land COP \land Rel \Rightarrow (\exists AS' \bullet AOP \land Rel')$$

Note that input and output variables have been ignored in the above for simplicity of presentation. For a fuller treatment, see page 138 of [381].

The 'pre ' schema operator gives the *precondition* of a schema in which all the after (dashed) state and output components (ending with '!') are existentially quantified.

The concrete state is thus considered as a data refinement of the abstract state, and each of the concrete operations must model the same behaviour on the concrete state as the corresponding abstract operation does on the abstract state.

The relationships between the two models can be illustrated as in Figure 4.1.

4.4 Service Documentation

In this section an outline of the structure adopted for the documentation of a service is presented. The documentation consists of two main parts, a *User Manual* and an *Implementor Manual*.

The manuals use Z throughout, and thus some effort is still required to transform the presented implementation into final code. Note that an Implementor Manual presents only one possible implementation, reflecting a particular set of design decisions. A programmer could choose to implement a service differently, provided it still satisfied the specification given in the user manual.

As an illustration of the manual style adopted by the project, some extended excerpts from the manuals for a very simple service are now presented. Occasional footnotes have been added to aid those not familiar with the Z notation. Some familiarity with set theory and first order predicate calculus (upon which Z is based) is assumed. Reading an introductory textbook such as [336] or one of those listed on page 244 beforehand is recommended if possible.

The manuals make use of some definitions from the Common Service Framework. In particular, some given sets of data values are not formally elaborated further:

$$[UserNum, Report]$$

UserNum is the set of numbers which identify system users; *Report* is the set of reports showing the outcome of an operation. Two individual unique special users are assumed:

$$ManagerNum, GuestNum : UserNum$$
$$\hline$$
$$ManagerNum \neq GuestNum$$

ManagerNum is the identity of the manager of a service; *GuestNum* is the identity of the guest (unauthenticated) user.

Time and intervals of time may be modelled are natural numbers for convenience, to allow standard arithmetic operations to be available on them:

$$Time == \mathbb{N}$$
$$Interval == \mathbb{N}$$

$$ZeroInterval == 0$$

Time is the set of date/time instants and *Interval* is the set of time intervals between such instants. *ZeroInterval* is a interval with no duration.

4.5 Reservation Service – User Manual

The Reservation Service allows clients to notify a manager how long they may require use of a system resource such as another service. A client may make a *reservation* for a specified period. Subsequently the reservation may be cancelled by the holder by requesting a reservation of zero interval. At any one time, there may be a number of client reservations.

The service manager may inspect the reservations whenever required. The manager may also set a *shutdown* time after which the availability of the resource being managed is no longer guaranteed (for example, because of maintenance). If any client reservations are threatened by the shutdown time, the manager will be notified, and can then negotiate with the clients concerned or set a new shutdown time. Note that a client cannot make a reservation past the current shutdown time.

Normally a client will make a reservation for some reasonable period before using the managed resource. However a client may still use the resource *without* making a reservation. In this case there is no guarantee about the availability of the resource.

4.5.1 Service state

A *reservation* records the user number (public identity) of the client who made it and the time at which it will expire. A number of reservations may exist at any one time. Each user may only have one reservation, and there is a limit on the total number of reservations (*Capacity*).

$$Capacity : \mathbb{N}_1$$

$$Reservations == \{r : UserNum \pfun Time \mid \#r \leq Capacity\}$$

(The above is an *abbreviation definition*, using a finite partial function from *UserNum* to *Time*.)

The state of the Reservation Service records the shutdown time most recently set by the service manager (*shutdown*) and the set of current reservations (*resns*). The guest user cannot make reservations.

RS

$$shutdown : Time$$
$$resns : Reservations$$
$$GuestNum \notin \mathrm{dom}\, resns$$

(*RS* is a *schema* with two declarations and a single predicate. This is a method of textually collecting together pieces of mathematics to aid structuring of specifications.)

Initially the shutdown is set to a default value and there are no reservations.

$$InitShutdownTime : Time$$

InitRS

$$RS'$$
$$shutdown' = InitShutdownTime$$
$$resns' = \emptyset$$

(Schemas may be included in other schemas. Declarations are merged (matching declarations combine into one), and predicates are conjoined. If a schema name is *decorated* (e.g., the $'$ above), all the components are also decorated. Note that predicates on successive lines in a schema are conjoined by default.)

The service is in its initial state every time it is powered up.

4.5.2 Operation parameters

For each operation requested by clients there is an output parameter reporting the outcome of the operation (*report*!). Additionally the current time (*now*) and the user number of the client (*clientnum*) are available.

ΦBasicParams

$$report! : Report$$
$$now : Time$$
$$clientnum : UserNum$$

('!' signifies outputs and '?' signifies inputs. Φ is just part of the name of the schema, used here to indicate an incomplete specification.)

Operations may change the state of the Reservation Service.

$$\Delta RS \mathrel{\widehat=} RS \land RS' \land \Phi BasicParams$$

(ΔRS is defined as a *horizontal* as opposed to a *vertical* schema definition here. Δ

conventionally indicates a change of state in Z. Here $\wedge$ is being applied to schemas and acts in much the same way as schema inclusion as described previously.)
Some operations may leave the state of the service unchanged.

$$\Xi RS \mathrel{\widehat=} [\, \Delta RS \mid \theta RS = \theta RS' \,]$$

(Again, the use of Ξ is a convention. θ indicates a *tuple* formed from a schema's components. Here the definition could be omitted since this is the default definition assumed by a Z specification if no explicit definition is made.)

Operations can return finite sets of users, the following definition is made for the convenience of subsequent specifications:

$$Users == \{u : \mathbb{F}\, UserNum \mid \#u \leq Capacity\}$$

4.5.3 Reports

There are a number of possible error reports from the operations, all of which have different values:

$$\begin{array}{l} SuccessReport, \\ NotAvailableReport, \\ TooManyUsersReport, \\ NotManagerReport, \\ NotKnownUserReport : Report \\ \hline \langle SuccessReport, NotAvailableReport, TooManyUsersReport, \\ \quad NotManagerReport, NotKnownUserReport \rangle \in \mathrm{iseq}\, Report \end{array}$$

The *report*! output parameter of each operation indicates either that the operation succeeded or suggests why it failed.

Success indicates successful completion of an operation.

Success

$$\begin{array}{l} report! : Report \\ \hline report! = SuccessReport \end{array}$$

If a reservation cannot be made due to early shutdown, the shutdown time itself is returned in *until*!. Note that a reservation of zero interval will not cause this error.

NotAvailable

$$\begin{array}{l} \Xi RS \\ interval? : Interval \\ until! : Time \\ \hline interval? \neq ZeroInterval \\ shutdown < now + interval? \\ until! = shutdown \\ report! = NotAvailableReport \end{array}$$

The service has finite capacity for recording reservations; the report *TooManyUsers* occurs when that capacity would be exceeded. The report *cannot* occur if the client has a reservation (since it is overwritten by the new one).

TooManyUsers

$$\Xi RS$$

$$\#resns = Capacity$$

$$clientnum \notin \operatorname{dom} resns$$

$$report! = TooManyUsersReport$$

Some operations can only be executed by the service manager, and return an error otherwise:

NotManager

$$clientnum : UserNum$$
$$report! : Report$$

$$clientnum \neq ManagerNum$$

$$report! = NotManagerReport$$

The guest user cannot make reservations, and an error is returned if this user tries to do so:

NotKnownUser

$$clientnum : UserNum$$
$$report! : Report$$

$$clientnum = GuestNum$$

$$report! = NotKnownUserReport$$

4.5.4 Service operations

Four operations are provided by the service. *Reserve*, which may be performed by any authentic client, is used to make or clear a reservation. *SetShutdown* and *Status*, which may be performed only by the service manager, set the shutdown time or allow the current reservations to be reviewed. *Scavenge*, which is performed by the service itself, removes expired reservations.

The description of each operation has three sections, entitled **Abstract**, **Definition** and **Reports**.

The **Abstract** section gives a procedure heading for the operation, with formal parameters, as it might appear in some programming language. The correspondence between this procedure heading and an implementation of it in some real programming language is designed to be obvious and direct. A short informal description of the operation may accompany the procedure heading.

The **Definition** section mathematically defines the successful behaviour of the operation. It does this by giving a schema which includes as a component every formal

parameter of the procedure heading, either explicitly or as components of included subschemas (such as ΔRS). A short explanation may accompany the schema.

The **Reports** section summarizes the report values which can be returned by the operation. This gives the definition of the total operation including the behaviour in the case of errors.

Only the definition of the *Reserve* operation is included here.

RESERVE

Abstract

```
Reserve ( interval? : Interval;
          until!    : Time;
          report!   : Report )
```

A reservation is made for a period of time (*interval*?), and returns the expiry time of the new reservation (*until*!).

A client can cancel a reservation by making a new reservation in which *interval*? is zero; this will then be removed by the next *scavenge*.

Definition *

$$
\begin{array}{|l}
\hline
\textit{Reserve}_{success} \\
\Delta RS \\
\textit{interval}? : \textit{Interval} \\
\textit{until}! : \textit{Time} \\
\hline
\textit{until}! = \textit{now} + \textit{interval}? \\
\textit{shutdown}' = \textit{shutdown} \\
\textit{resns}' = \textit{resns} \oplus \{\textit{clientnum} \mapsto \textit{until}!\} \\
\hline
\end{array}
$$

Reports †

$$
\begin{aligned}
\textit{Reserve} \mathrel{\widehat{=}} (&\textit{Reserve}_{success} \land \textit{Success}) \\
&\oplus \textit{TooManyUsers} \\
&\oplus \textit{NotAvailable} \\
&\oplus \textit{NotKnownUser}
\end{aligned}
$$

The client cannot be a guest user.

The reservation must expire before the shutdown time or be for a zero interval.

There may be no space for new reservations.

* In the **Definition** section, $\oplus$ is used for relational overriding. Any existing entry under *clientnum* in *resns* is removed and a new entry with value *until*! is added.

† In the **Reports** section, $\oplus$ is applied to schemas for schema overriding. Mathematically, this can be defined as $A \oplus B \mathrel{\widehat{=}} (A \land \neg \operatorname{pre} B) \lor B$, where $\operatorname{pre} B$ is the precondition of the B schema in which all after state and output components have been existentially quantified. In practice this means that the error conditions are 'checked' in reverse order.

4.5.5 Service charges

The basic parameters are supplemented by two *hidden* parameters, an operation identifier *op*? and the cost of executing the operation *cost*!. The latter can conveniently be defined in terms of natural numbers.

$[Op]$

$Money == \mathbb{N}$

ΦParams

$\Phi BasicParams$
$op? : Op$
$cost! : Money$

There is a fixed cost for each different successful operation. All clients who make a reservation will also be charged an amount depending on the requested interval.

$ReserveOp, SetShutdownOp, StatusOp : Op$
$ReserveCost, TimeCost, SetShutdownCost,$
$StatusCost, ErrorCost : Money$

$\langle ReserveOp, SetShutdownOp, StatusOp \rangle \in \operatorname{iseq} Op$

RSTariff

$\Phi Params$
$interval? : Interval$

$op? = ReserveOp \Rightarrow cost! = ReserveCost + (TimeCost * interval?)$
$op? = SetShutdownOp \Rightarrow cost! = SetShutdownCost$
$op? = StatusOp \Rightarrow cost! = StatusCost$

If an error occurs, a fixed amount may still be charged.

$ErrorTariff \mathrel{\widehat=} [\, \Phi Params \mid cost! = ErrorCost \,]$

These two schemas combine to form an overall 'tariff' schema.

$\Phi RSTariff \mathrel{\widehat=} Success \Rightarrow RSTariff \land \neg\, Success \Rightarrow ErrorTariff$

4.5.6 Complete service

The definition of the Reservation Service is completed by combining the definitions of the Reserve operation with the other operations and the tariff schema. In addition, components from the Common Service Framework are included to handle the service clock, accounting and statistics. Finally, the effect of invoking a non-existent service operation, or of a network error are specified.

The additional components needed to define the complete service are omitted in this overview but may be found in [55].

4.6 Reservation Service – Implementor Manual

In the Implementor Manual, the abstract specification of the User Manual is refined into a *concrete* specification of a possible implementation of the service. First the concrete state of the service is defined and then the concrete error and operation schemas are defined in terms of the concrete state components. Optimizations are included where desirable. The justification that the given concrete specification is a correct implementation of the abstract specification is discussed.

The specification given in the Implementor Manual is still not directly implementable. Predicates in schemas are given broadly in the order in which the corresponding statements of a procedure in a sequential programming language might be written, as a hint to the implementor. This is something that should be avoided in a truly abstract specification, but the Implementor Manual presents a more concrete internal view of the system. A particular programming language must be chosen by the implementor and then this design must be refined into that language. Even with the advent of the use of formal specification in the design of computer based systems, it is anticipated that the job of the programmer is safe for some time to come.

4.6.1 Concrete state

In the abstract state, the reservations are modelled as a partial function. We shall assume that the number of clients with reservations at any particular time is relatively small compared to the total number of clients (i.e., the function is sparse).

Hence in the concrete state, this partial function will be implemented as a pair of arrays containing matching user numbers and reservation times at corresponding array indices. Since not all entries in these arrays need be in use at any given moment, a special user number is needed to indicate an empty entry. The guest user is not allowed to make reservations, and cannot appear as a user number in the reservation table, so it may be used to denote unused entries in the array.

$$Unused : UserNum$$

$$Unused = GuestNum$$

Note that this design decision means that we cannot easily change our mind about whether guests can make reservations in this implementation. Thus a change in the requirements specification at a later date could necessitate a significant redesign. Design choices should bear such issues in mind if changes in the system are likely.

The arrays have indices limited to a maximum upper bound *Capacity* which determines the number of clients for whom the service can hold reservations simultaneously. This limit must be determined by the implementor according to the estimated usage of the service.

$$Index == 1 \,..\, Capacity$$

$$UserArray == Index \rightarrow UserNum$$

$$TimeArray == Index \rightarrow Time$$

The shutdown time may easily be implemented as a single variable (*shutd*), so that the concrete implemented service state consists of three components.

cRS

$$shutd : Time$$
$$users : UserArray$$
$$times : TimeArray$$

$$(users \rhd \{Unused\}) \in (Index \rightarrowtail UserNum)$$

($\rhd$ performs range anti-restriction of a function (in this case, all *Unused* entries are removed from the *users* array), and $\rightarrowtail$ indicates a partial injection or one-to-one function.)

Each authentic client can have at most one entry in the *users* array, all other entries being unused.

Initially the shutdown time has the default value and all the entries in the user array are unused. (It will not matter what values are held in the time array.)

cInitRS

$$cRS'$$

$$shutd' = InitShutdownTime$$
$$users' = (\lambda s : Index \bullet Unused)$$

Operations may change the state of the Reservation Service implementation:

$$\Delta cRS \mathrel{\widehat=} cRS \land cRS' \land \Phi BasicParams$$

Some operations may leave the state of the service unchanged:

$$\Xi cRS \mathrel{\widehat=} [\, \Delta cRS \mid \theta cRS = \theta cRS' \,]$$

This completes the specification of the concrete state, its initialization. and its change of state in general terms.

4.6.2 Operation implementations

The four service operations are redefined in this manual in terms of the refined concrete state. As in the User Manual, the description of each operation has three sections, entitled **Abstract**, **Definition** and **Reports**.

Each schema definition may be conveniently implemented as a procedure in the final program. Again, only the definition of the *Reserve* operation is included here. Note that the *TooManyUsers* report schema has been directly incorporated in the implementation of this operation.

The other error reports remain similar, except that the abstract state is replaced by the concrete state:

cNotAvailable

ΞcRS
$interval? : Interval$
$until! : Time$

$interval? \neq ZeroInterval$

$shutd < now + interval?$

$until! = shutd$

$report! = NotAvailableReport$

cNotKnownUser

ΞRS

$clientnum = GuestNum$

$report! = NotKnownUserReport$

RESERVE

Abstract

```
Reserve ( interval? : Interval;
          until!    : Time;
          report!   : Report )
```

Definition

In the concrete form of this operation, the *Success* and *TooManyUsers* report cases are optimized into a combined 'available' definition.

cReserve$_{avail}$

ΔcRS
$interval? : Interval$
$until! : Time$
$i, j : Index$

$shutd' = shutd$

$until! = now + interval?$

$clientnum \in \operatorname{ran} users \Rightarrow$
$\quad users(i) = clientnum$
$\quad users' = users$
$\quad times' = times \oplus \{i \mapsto until!\}$
$\quad report! = SuccessReport$

$clientnum \notin \operatorname{ran} users \Rightarrow$
$\quad Unused \in \operatorname{ran} users \Rightarrow$
$\quad\quad users(j) = Unused$
$\quad\quad users' = users \oplus \{j \mapsto clientnum\}$
$\quad\quad times' = times \oplus \{j \mapsto until!\}$
$\quad\quad report! = SuccessReport$

$\quad Unused \notin \operatorname{ran} users \Rightarrow$
$\quad\quad users' = users$
$\quad\quad times' = times$
$\quad\quad report! = TooManyUsersReport$

The shutdown time is unaffected.

A check is made to see whether an entry for the client already exists in the users array. If a client already has a reservation entry, then that entry in the array (with index *i*) is used. Otherwise, if there are any unused entries in the array, one of them (with index *j*) is used.

If the client does not have an existing entry and there are no unused entries, the state remains unchanged and an error report is given.

Reports

$cReserve \mathrel{\widehat=} cReserve_{avail} \oplus cNotAvailable \oplus cNotKnownUser$

4.7 Experience

The Reservation Service is one of a number of services designed by the project. Other services include a Time Service, a Block Storage Service and a File Service. Services can act as clients to other services if required. For example, the File Service is designed to make use of the Block Storage Service. Each service is designed to be simple and to perform one function. Taking the File Service again, this does not supply human readable file names or any file organization. If this is needed, a Directory Service could be used.

This section details some of the experiences of the project. Problems and advantages of using formal methods, and Z in particular, are discussed.

4.7.1 Manual format

The style of the User Manuals has evolved during the project. For comparison, an earlier version of the Reservation Service is presented in [203]. Since then, error conditions have been more exactly specified in the Reports section for each operation, using the schema overriding operator ($\oplus$) to define an order of checking for error conditions:

$$S \oplus T \mathrel{\widehat=} (S \land \neg \operatorname{pre} T) \lor T$$

The cost of performing operations has been gathered together in a tariff schema after the operations themselves have been presented. The cost of an operation is often of secondary interest to understanding *what* the operation does, and clutters its specification.

At the end of each manual, the operations specific to the service are combined with those incorporated from the Common Service Framework to produce an overall specification of the operations available in the service.

The initial state is now included formally for each service. The state of a service at any given time is the result of the initial state transformed by all the operations which have been performed to date.

4.7.2 Service implementation

With the introduction of Implementor Manuals, it has been possible to present an implementor's view of a service, showing how the abstract user's view can be refined towards a concrete implementation.

A significant amount of effort has been spent on the presentation of these manuals, since it is all too easy for them to become swamped by detail. The implementor manual for the Reservation Service, included here, is a relatively straightforward example because of the simplicity of the service itself. For a more realistically sized service, the reader is referred to [172] and to Chapter 5.

The ultimate goal of such specifications is to consider the refinement of the implementation of a service all the way down to the code of a particular programming language. Refinement, though better understood in theory [208, 230, 299], still requires the development of styles and techniques to manage the necessary mathematical manipulations in practice [295]. There has been much work on operation refinement

using Z [171, 246, 248, 308, 400, 423, 429, 431, 440, 441] and also data refinement [208, 238, 294]. Parallel refinement, linked with CSP [215], has also been considered [427, 438], as have object-oriented aspects [19, 257, 420]. The correctness of *real* programming languages, such as Ada, with respect to a Z specification is also an important issue [366]. In the case of safety-critical applications, timing can become an important issue for *hard real-time* systems where it is an essential part of the specification that the deadlines be met [271, 272].

Work at Oxford and elsewhere has considered the step from specification into programming language in more detail [295, 299]. For example, the *Reserve* operation has been systematically refined into Dijkstra's language of weakest preconditions [296]. The DCS project concentrated on the 'architectural' aspects of system design, taking a top-down approach in which the structure of the implementation was of greatest concern.

4.7.3 Representation of parameters

The types of parameters of service operations have been presented as Z sets. These can either be given sets, assumed to be unstructured, such as *Time* or *Report*, or they can be defined in Z as a set, sequence or other more complicated structure.

The issue of how such types will be represented in a specific programming language has been postponed. Clearly, at the lowest level, the parameter values must be transmitted over the network between client and service in some bit pattern. Since there is no assumption that all client applications and service code will be written in the same programming language, there would need to be a clear specification of the representation at this level so that data conversion functions could be applied if necessary.

Take, as an example, the set of *Reservation*s which is returned by the *Status* operation of the Reservation Service. This consists of a partial function (of limited size) from *UserNum* to *Time*. Most programming languages would not be able to implement this directly. Typically it could be implemented as an array with elements consisting of a record containing a user number and associated time. The ordering of the array could be arbitrary, or it may be ordered by user number or time.

Parameter refinement is an important topic which has received relatively little attention [355]. It could be considered as a relation between abstract and concrete parameters in a similar manner to the way abstract and concrete states are related. It would therefore form a second, orthogonal, dimension of refinement to that of the implementation of a service. It is crucial to investigate techniques that help avoid errors at interfaces since this is the point at which many problems that are difficult to detect beforehand occur.

4.7.4 Common service framework

The introduction of the *Common Service Framework* allows a number of definitions common to several services to be grouped together in one document. This means that the specifications of individual services can be made that much simpler.

The specification of the common service framework has illustrated how separate subsystems can be defined, with their own state and operations, and then incorporated into the definition of a complete service. It has addressed, at the specification level,

the issues of errors in the implementation of services or in the network over which they are accessed.

4.7.5 Service and network errors

There are two kinds of errors specified in the common framework which are non-deterministic. In other words, they do not arise because of some predicate which the client's parameters have failed to satisfy, but because of an error arising in the underlying implementation. Service errors are caused by a failure in the service implementation, such as a disk error in a storage service. Network errors are caused by a failure in communication over the network.

Both kinds of error have been made visible to the client through the return of corresponding error report values. It is left to the client's application to take appropriate action in the event of such errors arising. At a higher level of abstraction, it might be possible to hide transient errors from the client by automatically retrying operations until they achieved a definite result (i.e success, or a specific error report).

The specification of these non-deterministic errors is a problem. When a service error occurs, the state of the service is specified to remain unchanged. This may be hard to achieve in practice. For example, if a disk crashes and loses some of its data, the service will clearly not be able to maintain that part of its state. To keep within its specification, it would be obliged to return a service error for any subsequent operation which depended on information in the lost part of the state, effectively rendering it invisible to any client.

When a network error occurs, it is specified that either the state of the service remains unchanged or that the operation has been completed (though the result is not visible to the client). These two cases correspond to a communication failure in transmitting the operation request or reply respectively. On receiving such an error report, the client may re-attempt the operation. However, if the operation is not idempotent, such as one which creates or deletes a component in the service state, this will produce unwanted side-effects. A stricter specification might eliminate the second case, so that this error could be handled in the same way as a service error. The network implementation would then be obliged to provide a mechanism to recover from loss of operation replies.

4.7.6 Operators on basic sets

One area which is of concern in many Z specifications involves dealing with the partial nature of some of the underlying operators.

Operators such as addition, subtraction and comparison are assumed to exist for some of the sets, such as *Time* and *Money*, introduced in the Common Service Framework. These operators are defined to be total in the abstract specification to avoid having to introduce error checks and reports when they are invoked outside their domain.

Since these sets are to be implemented they must be finite. Hence 'overflow' or 'underflow' (i.e., the required result lies outside the defined range) could occur when adding or subtracting some values. Many arithmetic implementations in hardware simply wrap round when this occurs, producing undetectable invalid results. A more

sensible approach is to return some standard error value in these cases. Output parameters may be checked for this value by the client if desired.

4.7.7 A problem discovered

The Reservation Service was one of the first services to be implemented, and its original User Manual has been published previously [173, 174, 203]. However an error was discovered during the use of the service which was not anticipated during the design stage. This has led to a small revision in the specification of one of the error schemas for the service.

The problem arose when a client made a reservation successfully and subsequently tried to clear it by making a reservation of zero-interval in the normal way. However the service reported that it was '*Not Available*' and hence the reservation could not be removed.

The specification in the User Manual was examined to see how this state of affairs might transpire. To obtain the '*Not Available*' report, the following precondition in the *NotAvailable* schema had to hold:

$$shutdown < now + interval?$$

With *interval*? being zero, this implied that the shutdown time was set earlier than the current time. Given that the client had earlier successfully made a reservation that was still in force (and hence needed to be cleared), this implied that the shutdown time had been brought forward by the service manager, threatening the pending reservation. In fact, the client was a laser printing service which was known always to make half-hour reservations. The manager had set a shutdown time earlier than the end of the printing service's reservation time, assuming that it could clear its current reservation but not make any new reservations. The manager was prepared to wait until the reservation was cleared as an indication that the printer had finished its current job; but the reservation was never cleared.

4.7.8 The problem solved

To prevent the problem of not being able to cancel reservations after the shutdown time, two solutions were proposed. The choice between them illustrates the kind of design choice in which additional complexity in a single operation may be balanced against the use of an additional operation. A first solution involves adding an extra precondition to the original *NotAvailable* schema:

$$interval? \neq ZeroInterval$$

This means that a call to the *Reserve* operation with a *ZeroInterval* can no longer return with a *NotAvailableReport*.

This is the solution presented in the previous sections in which the *Reserve* operation serves the dual purpose of making a reservation (when $interval? \neq ZeroInterval$) and also clearing a reservation (when $interval? = ZeroInterval$).

An alternative solution to this would be to provide a new *Cancel* operation for clearing a reservation. This complicates the service by providing an extra operation, which

is the reason it was not included in the original version of the service. However it is likely that its inclusion would have prevented the problem just described from arising.

4.7.9 Proof of correctness

For the Reservation Service, the design in the Implementor Manual has been proven correct with respect to the User Manual [407]. However, the proofs are quite long, even for such a simple service. The document also shows how the operations can then be programmed in Dijkstra's guarded command language to meet the specifications in the Implementor Manual.

4.8 Conclusions

It is possible to use a formal specification language successfully both to guide the design of system components and also to document the resultant design. The specification language Z, after some experimentation to devise an appropriate style of manual presentation, has been used in both User and Implementor Manuals for system services.

As well as being more precise than conventional informal documentation, such formal designs are necessary if the correctness of implementations are of concern, though proof of correctness is still laborious.

The initial desire to present the formal specifications as part of the manuals for the services has forced the designs to be simple, and has concentrated our attention on easing the presentation of formal notation.

In Chapter 5, immediately following this chapter, a more substantial example of a User Manual is provided.

Chapter 5

A File Storage Service

This chapter presents the User Manual for a more substantial network service along the lines described in Chapter 4. The service provides file storage facilities to clients via a number of remote procedure calls. These are modelled as operations in the Z specification provided here.

5.1 Service State

The file storage service stores files on behalf of its clients. A client may submit some *data* consisting of an arbitrary sequence of byte values up to a maximum size:

$$[ByteVal]$$

$$MaxFileSize : \mathbb{N}$$

$$Data == \{s : \operatorname{seq} ByteVal \mid \#s \leq MaxFileSize\}$$

The service will store this data within a file:

File

$$owner : UserNum$$
$$created, updated, : Time$$
$$expires : Time$$
$$contents : Data$$

$$created \leq updated$$
$$created \leq expires$$

As well as containing the client's data, the file records as its *owner* the user number of the client who submitted it, and it records the time of its original creation and last update. Whenever a file is created, an *expiry* time must be given by the client; it is the time until which the service is obliged to store the file. After this time, a file is said to have *expired*, and can be discarded by the service without notification of the client.

This expiry time may be changed later if required. The creation time, last update time and expiry time are always ordered:

$$\_ \leq \_ : \mathit{Time} \leftrightarrow \mathit{Time}$$

$$(\_ \leq \_) \in \mathit{total\_order}$$

Here *total_order* defines a total order on *Time* (see page 54).

A file *identifier* will be issued by the service when the file is created, chosen from a set of such identifiers:

$$[\mathit{FileId}]$$

This becomes the client's reference to the file. Any subsequent operations on the file will require this identifier. Operations which update the file contents will return a new name. Hence files are *immutable* in the sense that a known valid file identifier will either access a single fixed file or return an error if it has been destroyed.

The service contains a mapping from file identifiers to files; it also contains a finite set of *new* file identifiers which have not yet been issued. When a new identifier is issued, it is taken from this set. There is a special *NullFileId* which is never issued by the service.

$$\mathit{NullFileId} : \mathit{FileId}$$

FS

$$\mathit{files} : \mathit{FileId} \pfun \mathit{File}$$
$$\mathit{newids} : \mathbb{F}\, \mathit{FileId}$$

$$\mathit{newids} \cap \mathrm{dom}\, \mathit{files} = \emptyset$$
$$\mathit{NullFileId} \notin \mathit{newids}$$

NullFileId can be used by clients' applications to indicate 'no file' (similarly to the use of the *nil* pointer in a programming language).

Initially there are no files, and all identifiers except the *NullFileId* are potentially available:

InitFS

$$\mathit{FS}'$$

$$\mathit{files}' = \emptyset$$
$$\mathit{newids}' = \mathit{FileId} \setminus \{\mathit{NullFileId}\}$$

Each file storage service operation can only be performed by an authentic client:

$$\mathit{clientnum} : \mathit{UserNum}$$

During an operation, the service can ask the time service for the current time:

$$\mathit{now} : \mathit{Time}$$

Every operation the service can perform for a client provides a report as output, normally *SuccessReport* (see later):

$$\mathit{report}! : \mathit{Report}$$

Finally, any identifier issued by an operation is removed from the set of new ids, and so can never be issued again:

$$newids' = newids \setminus \mathrm{dom}\, files'$$

These general aspects of operations on the file storage service are gathered together in a single schema:

ΔFS

$$FS$$
$$FS'$$
$$clientnum : UserNum$$
$$now : Time$$
$$report! : Report$$

$$newids' = newids \setminus \mathrm{dom}\, files'$$

Sometimes the state of the file storage service is left unaffected by an operation, particularly if an error is detected or it is a status operation:

$$\Xi FS \mathrel{\widehat=} [\Delta FS \mid \theta FS' = \theta FS]$$

Many file storage service operations require an existing file $id?$ to be supplied by the user. The *File* details are then available to the operation. A partially specified schema is used to include this information for all operations which take an $id?$ as input:

$\Phi FileId?$

$$\Delta FS$$
$$id? : FileId$$
$$File$$

$$\theta File = files\, id?$$

All operations which create a new file return a new file $id!$. The new $File'$ details are then available to the operation. Guest users cannot create files. The client owns the new file which has been updated *now*. The file is added to the set of files stored by the service. Another partial schema includes all this information for operations which produce a file $id!$ as output:

$\Phi FileId!$

$$\Delta FS$$
$$id! : FileId$$
$$File'$$

$$clientnum \neq GuestNum$$
$$owner' = clientnum$$
$$updated' = now$$
$$id! \in newids$$
$$files' = files \cup \{id! \mapsto \theta File'\}$$

Some of the file operations access sections of the file contents. Three functions are useful for these definitions:

$$
\begin{array}{|l}
[B] \\
\_ \underline{\mathit{after}} \_, \\
\_ \underline{\mathit{for}} \_ : (\operatorname{seq} B) \times \mathbb{N} \rightarrow (\operatorname{seq} B) \\
\_ \underline{\mathit{shifted}} \_ : (\operatorname{seq} B) \times \mathbb{N} \rightarrow (\mathbb{N} \pfun B) \\
\hline
\forall s : \operatorname{seq} B;\ n : \mathbb{N} \bullet \\
\quad s \underline{\mathit{after}}\ n = (\{0\} \ndres \mathit{succ}^n) \fatsemi s \land \\
\quad s \underline{\mathit{for}}\ n = (1 \upto n) \ndres s \land \\
\quad s \underline{\mathit{shifted}}\ n = \mathit{succ}^{-n} \fatsemi s \\
\end{array}
$$

Intuitively, an operation may access the *contents* of a file *after* a certain position, *for* a given number of bytes. Additionally, it is possible to update the *contents* of a file using *data* which has been *shifted* by a specified offset.

It is possible for '*holes*' to be created in a file's *contents* which have never been previously set to any value if data is written to a file at a position after its current length or the file length is set to be greater than the current length. A *Background* value will be supplied by the service if such a location is accessed subsequently.

$$
\begin{array}{|l}
\mathit{Background} : \mathit{ByteVal} \\
\mathit{holes} : \operatorname{seq} \mathit{ByteVal} \\
\hline
\operatorname{dom} \mathit{holes} = 1 \upto \mathit{MaxFileSize} \\
\operatorname{ran} \mathit{holes} = \{\mathit{Background}\}
\end{array}
$$

Note that a file size ranges from zero up to a maximum size.

$$\mathit{Size} == 0 \upto \mathit{MaxFileSize}$$

5.2 Error Reports

The *report*! parameter of each operation indicates either that the operation succeeded or suggests why it failed using different reports with unique values:

$$
\begin{array}{|l}
\mathit{SuccessReport}, \\
\mathit{NoSuchFileReport}, \\
\mathit{NoSpaceReport}, \\
\mathit{NotOwnerReport}, \\
\mathit{NotKnownUserReport}, \\
\mathit{BadOperationReport} : \mathit{Report} \\
\hline
\langle \mathit{SuccessReport}, \mathit{NoSuchFileReport}, \mathit{NoSpaceReport}, \mathit{NotOwnerReport}, \\
\quad \mathit{NotKnownUserReport}, \mathit{BadOperationReport} \rangle \in \operatorname{iseq} \mathit{Report}
\end{array}
$$

In most cases, failure leaves the service unchanged.

An operation can return only the report values listed in the Reports section of its definition. If it returns the value *Success*, it must satisfy its defining schema. If it returns any other value, it must satisfy instead the appropriate schema below.

NoSuchFile

NoSuchFile

ΞFS
$id? : FileId$

$id? \notin \mathrm{dom}\, files$
$report! = NoSuchFileReport$

This report is given if there is no file stored under *id*?; note that this may be because the file expired and has been scavenged.

NoSpace

NoSpace

ΞFS

$report! = NoSpaceReport$

A new file cannot be created when the service's storage capacity is exhausted. The file storage service capacity is not modelled here, but it is guaranteed that the state of the service will be unaffected in this case.

NotOwner

NotOwner

ΞFS
$owner : UserNum$

$owner \neq clientnum$
$report! = NotOwnerReport$

A client operations which destroys a file must be performed by the owner of the file.

NotKnownUser

This is defined in the *Common Service Framework* document [55].* In addition, the *Common Service Framework* defines:

$$BadOperation \mathrel{\widehat=} [report! : Report \mid report! = BadOperationReport]$$

together with *IsKnownUser*:

IsKnownUser

$clientnum : UserNum$

$clientnum \neq GuestNum$

* Also on page 75 in this book for convenience.

5.3 Service Operations

5.3.1 Client operations

There are a number of operations which a client may ask the service to perform:

Null – null operation (detailed in the *Common Service Framework* document [55]).

NewFile – create a new file of zero length.

WriteFile – write data to a stored file.

ReadFile – read data from a stored file.

DestroyFile – remove a stored file from the service.

FileStatus – obtain the complete status of a stored file.

SetFileExpiry – set the expiry time of a stored file.

SetFileLength – set the length of a stored file.

The following pages contain descriptions of each of the service specific operations. The descriptions have three sections, entitled Abstract, Definition and Reports.

The **Abstract** section gives a procedure heading for the operation, with formal parameters, as it might appear in a programming language. The correspondence between this procedure, and an implementation of it in a real programming language, must be obvious and direct.

Each formal parameter is given a name ending with either ? or !. Those ending with ? are inputs, and those ending with ! are outputs by convention. A short description accompanies the procedure heading.

The **Definition** section mathematically defines the operation, by giving a schema which includes as a component every formal parameter of the procedure heading; within the schema also appears a subschema (ΔFS or ΞFS) whose components include the service state before (FS) and after (FS') the operation. Partial schemas may also appear here. These partial schemas contain components appearing in the schema which are local to the operation (that is, temporary) and may assume any values consistent with the predicates.

The client is directly aware only of the components which are formal parameters of the procedure heading. A short description accompanies the schema.

The **Reports** section given the definition of the *total* operation including all the possible (success or failure reporting) values which the *report*! formal parameter may assume. These are effectively given in reverse order, the last error report schema *overriding* the previous ones and so on back through the list. Section 5.2 discusses report values in more detail by giving a mathematical definition of each of their occurrences.

Section 5.4 details the costs involved in performing operations. Section 5.5 gives schema definitions for *total* operations taking into account all possible file storage service errors. These sections use some generally available schema definitions which are defined in the *Common Service Framework* document [55].

NEWFILE

Abstract

```
NewFile ( expires? : Time;
          id!      : FileId;
          report!  : Report )
```

A file is formed with a specified expiry time, and is stored by the service under the new file *id*!. The new file contains no data.

Definition

NewFile

$$\Delta FS$$
$$expires? : Time$$
$$\Phi FileId!$$

$$created' = now$$
$$expires' = max\{expires?, now\}$$
$$contents' = \langle\rangle$$

The owner of the file is the client. Guest users cannot create new files.

If an expiry time in the past is given, then the expiry time of the file is set to *now*.

A new identifier is chosen which has never before been issued, and the new empty file is stored under that id.

Reports

$$NewFile_1 \mathrel{\widehat=} (NewFile \land IsKnownUser \land Success) \oplus NoSpace \oplus (\Xi FS \land NotKnownUser)$$

WRITEFILE

Abstract

```
WriteFile ( id?     : FileId;
            offset? : Size;
            data?   : Data;
            id!     : FileId;
            report! : Report )
```

An existing file with the given *id*? is updated with the new *data*? at the specified *offset*? in the file to produce a new file with a new *id*!. The original file is unaffected.

Definition

WriteFile

$$\Delta FS$$
$$\Phi FileId?$$
$$offset? : Size$$
$$data? : Data$$
$$\Phi FileId!$$

$$created' = created$$
$$expires' = expires$$
$$contents' = (holes\ \underline{for}\ offset?) \oplus contents \oplus (Size \lhd (data?\ \underline{shifted}\ offset?))$$

The creation and expiry times of the updated file are the same as the original file.

Any client apart from the guest user may write to a file. The owner of the new file is the client.

A new identifier is chosen which has never previously been issued, and the new file is stored under that id.

The new file will contain 'holes' if the offset given is past the end of the existing file.

Reports

$$WriteFile_1 \mathrel{\widehat=} (WriteFile \land IsKnownUser \land Success) \oplus NoSpace \oplus NoSuchFile \oplus (\Xi FS \land NotKnownUser)$$

READFILE

Abstract

```
ReadFile ( id?     : FileId;
           offset? : Size;
           length? : Size;
           data!   : Data;
           report! : Report )
```

Data at the specified *offset*? and of the given *length*? in the file called *id*? is returned.

Definition

ReadFile

ΞFS
$\Phi FileId?$
offset?,
length? : *Size*
data! : *Data*

---

data! = *contents* *after* *offset*? *for* *length*?

The service is unchanged by this operation.

Any client, including the guest user, may read a file if they know its file id.

No error is reported if the requested section lies partially or totally outside the bounds of the file. In this case the data returned will simply have a length of less than the requested *length*?.

Reports

$$ReadFile_1 \mathrel{\widehat{=}} (ReadFile \land Success) \oplus NoSuchFile$$

DESTROYFILE

Abstract

```
DestroyFile ( id?     : FileId;
              report! : Report )
```

The file stored under *id*? is removed from the service.

Definition

DestroyFile

ΔFS
$\Phi FileId?$

$clientnum = owner$
$files' = \{id?\} \lhd files$

A file may be destroyed only by its owner.

Reports

$$DestroyFile_1 \mathrel{\widehat=} (DestroyFile \land IsKnownUser \land Success) \oplus NotOwner \oplus NoSuchFile \oplus (\Xi FS \land NotKnownUser)$$

FILESTATUS

Abstract

```
FileStatus ( id?      : FileId;
             owner!   : UserNum;
             created! : Time;
             updated! : Time;
             expires! : Time;
             length!  : Size;
             report!  : Report )
```

The status of the file stored under *id*? is returned to the client.

Definition

FileStatus

ΞFS
$\Phi FileId?$
$owner! : UserNum;$
$created!,$
$updated!,$
$expires! : Time;$
$length! : Size$

$owner! = owner$
$created! = created$
$updated! = updated$
$expires! = expires$
$length! = \#contents$

The service is unchanged by this operation.

Reports

$$FileStatus_1 \mathrel{\widehat=} (FileStatus \land Success) \oplus NoSuchFile$$

SETFILEEXPIRY

Abstract

```
SetFileExpiry ( id?      : FileId;
                expires? : Time;
                id!      : FileId;
                report!  : Report )
```

An existing file stored under *id*? is used to created a new file with a new *id*! and a new expiry time.

Definition

SetFileExpiry

$$\Delta FS$$
$$\Phi FileId?$$
$$expires? : Time$$
$$\Phi FileId!$$

$$created' = created$$
$$expires' = max\{expires?, now\}$$
$$contents' = contents$$

The existing file is unaffected.

If an expiry time in the past is given, then the expiry time of the file is set to *now*.

Reports

$$SetFileExpiry_1 \mathrel{\widehat=} (SetFileExpiry \land IsKnownUser \land Success) \oplus NoSuchFile \oplus (\Xi FS \land NotKnownUser)$$

SETFILELENGTH

Abstract

```
SetFileLength ( id?     : FileId;
                length? : Size;
                id!     : FileId;
                report! : Report )
```

The length of the file stored under *id*? is changed to *length*?.

Definition

SetFileLength

ΔFS
$\Phi FileId?$
$length? : Size$
$\Phi FileId!$

$created' = created$
$expires' = expires$
$contents' = (holes \oplus contents)$ *for* $length?$

A new identifier is chosen which has never previously been issued, and the new file is stored under that identifier.

The new file contains 'holes' after the previous contents if the new *length*? is greater than the previous length of the file.

Reports

$$\begin{aligned} SetFileLength_1 \mathrel{\widehat{=}} & (SetFileLength \land IsKnownUser \land Success) \\ & \oplus NoSpace \\ & \oplus NoSuchFile \\ & \oplus (\Xi FS \land NotKnownUser) \end{aligned}$$

5.3.2 Manager operations

There is a service manager who has several special operations available to help with the running of the service:

Enable – enable file storage service.

Disable – disable file storage service.

These are described in the *Common Service Framework* document [55].

5.3.3 Implementation operations

Some operations are needed because of the implementation of the file storage service. For example, there is a *scavenge* operation which the service may perform at any time: it cannot, however, be requested by clients.

SCAVENGEFILE

ScavengeFile – remove an expired file.

Abstract

```
ScavengeFile ( id? : FileId )
```

The file stored under $id?$ is removed from the service; only expired files may be scavenged.

A scavenge may be invoked by the file service at any time; it can never be invoked by clients.

$$FileService : UserNum$$

Definition

ScavengeFile

$$\Phi FileId?$$

$$clientnum = FileService$$
$$expires < now$$
$$files' = \{id?\} \lhd files$$

5.4 Costs and Accounting

Costs Every operation which can be performed by a client costs a certain amount of *money*, which may have two components. One is the overhead of performing the operation itself:

$$NewFileCost,$$
$$WriteFileCost,$$
$$ReadFileCost,$$
$$DestroyFileCost,$$
$$FileStatusCost,$$
$$SetFileExpiryCost,$$
$$SetFileLengthCost : Money$$

If an error occurs, a small cost will still be charged:

$$FSErrorCost : Money$$

The other component, if present, is related to the service requested by the operation. For example, the *WriteFile* operation charges *in advance* for the storage of the data submitted, and the *DestroyFile* operation may give a *rebate* (negative expense) if the file is destroyed before its expiry time.

The expense of storing data in a file is determined by applying a *tariff* function using the file's update and expiry times. Similar tariff functions apply to reading data and obtaining file identifiers. There are various constants involved with this.

$$StoreByteCost,$$
$$ReadByteCost,$$
$$GetIdCost : Money$$

The tariff schema in Figure 5.1 defines the costs involved when a client performs file storage service operations. These are described using the constants introduced above and the scheme outlined in the *Common Service Framework* document [55].

5.4.1 Accounting policy

The values of *NewFileCost*, etc., and of the tariff functions may be varied; their precise values at any time will be published separately. Expenditure will be recorded in a log, and clients will be expected to observe any limits placed upon them.

5.5 Total Operations

This section provides a full definition of all the total operations which clients may request the file storage service to perform. This definition uses schemas which are defined in this document as well as those which check authentication etc., as defined in the *Common Service Framework* document [55].

$$\Phi Op \mathrel{\widehat=} [op? : Op]$$

$$NewFileOp, WriteFileOp, ReadFileOp, DestroyFileOp,$$
$$FileStatusOp, SetFileExpiryOp, SetFileLengthOp : Op$$

$$\langle NewFileOp, WriteFileOp, ReadFileOp, DestroyFileOp,$$
$$\quad FileStatusOp, SetFileExpiryOp, SetFileLengthOp\rangle \in \mathrm{iseq}\, Op$$

The set of operations implemented by the file storage service for clients is given in Figure 5.2.

5.6 Security

The service provides limited security in two areas; in both cases it depends on certain values being chosen from such a large set that they are hard to guess.

A client may not access a file unless he knows its id, and file ids are hard to guess. The identifier of any file is initially known only to its creator; the service will never tell any client the identifier of a file he does not own.

Tariff

ΔFS
$op? : Op$
$\Phi FileId?$
$\Phi FileId!$
$data! : Data$
$idset! : \mathbb{F}\, FileId$
$cost! : Money$

$report! = SuccessReport \Rightarrow$
$\quad op? = NewFileOp \Rightarrow cost! = NewFileCost$

$\quad op? = WriteFileOp \Rightarrow cost! = WriteFileCost$
$\qquad + StoreByteCost * (expires' - updated') * \#contents'$

$\quad op? = ReadFileOp \Rightarrow cost! = ReadFileCost + ReadByteCost * \#data!$

$\quad op? = DestroyFileOp \Rightarrow cost! = DestroyFileCost$
$\qquad - StoreByteCost * (expires' - updated') * \#contents'$

$\quad op? = FileStatusOp \Rightarrow cost! = FileStatusCost$

$\quad op? = SetFileExpiryOp \Rightarrow cost! = SetFileExpiryCost$
$\qquad + StoreByteCost * (expires' - expires) * \#contents'$

$\quad op? = SetFileLengthOp \Rightarrow cost! = SetFileLengthCost$
$\qquad + StoreByteCost * (expires' - updated')$
$\qquad\quad * (\#contents' - \#contents)$

$report! \in \{NoSuchFileReport, NoSpaceReport, NotOwnerReport,$
$\qquad NotKnownUserReport\} \Rightarrow cost! = FSErrorCost$

Figure 5.1 File Storage Service tarriff schema.

$$
\begin{aligned}
Ops \mathrel{\widehat=} Tariff \land ((&\Xi FS \land BadOperation) \oplus \\
&([NewFile_1;\ \Phi Op \mid op? = NewFileOp] \lor \\
&[WriteFile_1;\ \Phi Op \mid op? = WriteFileOp] \lor \\
&[ReadFile_1;\ \Phi Op \mid op? = ReadFileOp] \lor \\
&[DestroyFile_1;\ \Phi Op \mid op? = DestroyFileOp] \lor \\
&[FileStatus_1;\ \Phi Op \mid op? = FileStatusOp] \lor \\
&[SetFileExpiry_1;\ \Phi Op \mid op? = SetFileExpiryOp] \lor \\
&[SetFileLength_1;\ \Phi Op \mid op? = SetFileLengthOp]))
\end{aligned}
$$

Figure 5.2 Combined file system operations.

Files may be destroyed only by their owners, and user identifiers are hard to guess. Files may be updated, but this creates a new file with a new file id. The original file is left unaffected. Files are only removed from the service when they are destroyed or after their expiry time has passed.

III

UNIX Software

The UNIX operating system is widely used on workstations throughout the world. Here, two pieces of software designed to run under UNIX are presented. In Chapter 6 a text justification tool is formalized. An event-based input system designed for workstations is presented in Chapter 7.

Chapter 6

A Text Formatting Tool

In this chapter a simple text processing tool which allows left, centred and right justification of lines within an ASCII *text file is formalized in Z. Implementation details such as the use of* tab *characters and* newline *sequences are covered. The program has been implemented under the widely used* UNIX *operating system [37]. Readers may like to peruse the* UNIX *manual page for the tool, reproduced on page 118, before reading the chapter.*

6.1 Basic Concepts

The UNIX filing system has been specified in Z elsewhere (see [298]). Here we consider individual UNIX text files which consist of ASCII characters. We denote the set of characters under consideration as *CHAR*.

$$[\, CHAR \,]$$

One of these characters is a blank space.

$$space : CHAR$$

Characters are organized as lines in a text file. A complete text file, or document, consists of a number of lines of characters.

$$LINE == \operatorname{seq} CHAR$$

$$DOC == \operatorname{seq} LINE$$

We have modelled a line as a sequence of characters and a document as a sequence of lines above. Suppose a sequence needs to be repeated a certain number of times. The Z specification language may be extended, as done in its mathematical 'toolkit' for many operators, by defining a generic function to do this.

$$[X]$$

$$\_ : (\operatorname{seq} X) \times \mathbb{N} \rightarrow \operatorname{seq} X$$

$$\forall s : \operatorname{seq} X;\ n : \mathbb{N}_1 \bullet$$
$$s0 = \langle\rangle \land$$
$$sn = s \frown (s(n-1))$$

This definition will prove useful in the subsequent specification in this chapter. A number of theorems apply to *rep*:

$$
\begin{array}{rcl}
s : \operatorname{seq} X & \vdash & s0 = \langle\rangle \\
s : \operatorname{seq} X & \vdash & s1 = s \\
n : \mathbb{N} & \vdash & \langle\rangle n = \langle\rangle \\
s : \operatorname{seq} X;\ n : \mathbb{N} & \vdash & \#(sn) = \#s * n \\
s : \operatorname{seq} X;\ n : \mathbb{N}_1 & \vdash & \operatorname{ran}(sn) = \operatorname{ran} s \\
s : \operatorname{seq} X;\ n : \mathbb{N} & \vdash & \forall m : \mathbb{N} \mid m < n \bullet \\
 & & \quad (sn)\ \underline{after}\ (\#s * m)\ \underline{for}\ n = s
\end{array}
$$

6.2 Processing the Input

6.2.1 Lines

Consider a function which removes all leading spaces (if any) from a line:

$$
\begin{array}{|l}
\hline
clipleft : LINE \rightarrow LINE \\
\hline
clipleft\langle\rangle = \langle\rangle \\
\forall l : LINE \mid l \neq \langle\rangle \bullet \\
\quad head\ l \neq space \Rightarrow clipleft\ l = l \wedge \\
\quad head\ l = space \Rightarrow clipleft\ l = clipleft\ (tail\ l) \\
\hline
\end{array}
$$

All the trailing spaces may be removed by reversing the line, repeating the function above and then reversing the line back again:

$$
\begin{array}{|l}
\hline
clipright : LINE \rightarrow LINE \\
\hline
clipright = rev \fatsemi clipleft \fatsemi rev \\
\hline
\end{array}
$$

To clip both leading and trailing spaces, these functions may be combined as follows:

$$
\begin{array}{|l}
\hline
clip : LINE \rightarrow LINE \\
\hline
clip = clipleft \fatsemi clipright \\
\hline
\end{array}
$$

Note that the functions may be combined in either order with the same effect:

$$\vdash\ clipleft \fatsemi clipright = clipright \fatsemi clipleft$$

The left-hand end of a line may be indented to a given column position.

$$
\begin{array}{|l}
\hline
left_0 : \mathbb{N} \rightarrow LINE \rightarrow LINE \\
\hline
\forall n : \mathbb{N};\ l : LINE \bullet \\
\quad left_0\ n\ l = (\langle space\rangle n) \frown l \\
\hline
\end{array}
$$

The line could be centred on a given column position if the line is not too long. If it is too long, the line is left unaffected.

$$
\begin{array}{|l}
centre_0 : \mathbb{N} \rightarrow LINE \rightarrow LINE \\
\hline
\forall\, n : \mathbb{N};\ l : LINE \bullet \\
\quad \#l \leq 2 * n \Rightarrow centre_0\ n\ l = (\langle space \rangle (n - (\#l \operatorname{div} 2))) \frown l \wedge \\
\quad \#l > 2 * n \Rightarrow centre_0\ n\ l = l
\end{array}
$$

Similarly the right-hand end of the line can be aligned, again if it is not too long.

$$
\begin{array}{|l}
right_0 : \mathbb{N} \rightarrow LINE \rightarrow LINE \\
\hline
\forall\, n : \mathbb{N};\ l : LINE \bullet \\
\quad \#l \leq n \Rightarrow right_0\ n\ l = (\langle space \rangle (n - \#l)) \frown l \wedge \\
\quad \#l > n \Rightarrow right_0\ n\ l = l
\end{array}
$$

Corresponding functions which clip the line first may also be defined.

$$
\begin{array}{|l}
left, \\
centre, \\
right : \mathbb{N} \rightarrow LINE \rightarrow LINE \\
\hline
\forall\, n : \mathbb{N};\ l : LINE \bullet \\
\quad left\ n\ l = left_0\ n\ (clip\ l) \wedge \\
\quad centre\ n\ l = centre_0\ n\ (clip\ l) \wedge \\
\quad right\ n\ l = right_0\ n\ (clip\ l)
\end{array}
$$

6.2.2 Documents

Operations will be applied to a particular text file document, defined as a named *schema*.

$$
\begin{array}{|l}
\hline
text : DOC \\
\hline
\end{array}
\qquad TEXT
$$

During operations, changes in this document need to be recorded as a change of state:

$$\Delta TEXT \ \widehat{=}\ TEXT \wedge TEXT'$$

The *left*, *centre* and *right* functions may be combined and selected using an option to define which operation is required.

$$Option ::= LeftOption \mid CentreOption \mid RightOption$$

$$
\begin{array}{|l}
\textit{POS}_0 \\
\hline
\Delta \textit{TEXT} \\
\textit{option}? : \textit{Option} \\
\textit{column}? : \mathbb{N} \\
\hline
\textit{option}? = \textit{LeftOption} \Rightarrow \textit{text}' = \textit{text} \fatsemi (\textit{left column}?) \\
\textit{option}? = \textit{CentreOption} \Rightarrow \textit{text}' = \textit{text} \fatsemi (\textit{centre column}?) \\
\textit{option}? = \textit{RightOption} \Rightarrow \textit{text}' = \textit{text} \fatsemi (\textit{right column}?) \\
\hline
\end{array}
$$

This defines an operation which takes as input (actually as command arguments under UNIX) an option specifying the type of operation required (left, centered, or right justification), and a column position to be used for the justification of the text.

6.3 Implementation Details

In the previous section, it has been assumed that all the characters in lines are printable characters (of the same printing width). In practice, an additional *tab* character is often used for horizontal tabbing.

6.3.1 Tabs

The horizontal *tab* may be included in the character set as a new unique character:

$$
\begin{array}{|l}
\textit{tab} : \textit{CHAR} \\
\hline
\textit{tab} \neq \textit{space} \\
\end{array}
$$

The visual effect of this special tab character is to 'tabulate' to the next tab column position from the current position. This normally occurs at, for example, every eighth column under UNIX. The distance between tab column positions is always greater than one character position. Otherwise it would be equivalent to a space, and thus of little use.

$$
\begin{array}{|l}
\textit{tabsize} : \mathbb{N} \\
\hline
\textit{tabsize} > 1 \\
\end{array}
$$

The tab character introduces additional complication into the specification of manipulation of text files. We shall now consider some functions useful for specifying such manipulation.

Below is a generic function which splits a sequence into a series of segments of a given non-zero length. Flattening these segments gives the original sequence. The last segment may be of smaller size than the rest.

$$
\begin{array}{|l}
[X] \\
\_ \underline{split} \_ : (\operatorname{seq} X) \times \mathbb{N}_1 \rightarrow \operatorname{seq}(\operatorname{seq} X) \\
\hline
\forall s : \operatorname{seq}_1 X;\ n : \mathbb{N}_1;\ ss : \operatorname{seq}_1(\operatorname{seq}_1 X) \bullet \\
\quad \langle\rangle \ \underline{split}\ n = \langle\rangle \ \land \\
\quad s \ \underline{split}\ n = ss \Leftrightarrow \\
\qquad \frown\!/\, ss = s \ \land \\
\qquad (\forall t : \operatorname{seq} X \mid t \in \operatorname{ran}(front\ ss) \bullet \#t = n) \ \land \\
\qquad 0 < \#(last\ ss) \leq n
\end{array}
$$

The trailing spaces of a line segment can be converted to a tab if the line segment has the length of the distance between tab column positions. There should also be more than one trailing space since this will reduce the length of the line and make the substitution using a tab character worthwhile.

$$
\begin{array}{|l}
addtab : LINE \pfun LINE \\
\hline
\forall l : LINE \mid tab \notin \operatorname{ran} l \bullet \\
\quad (\#l = tabsize \land \#(clipright\ l) < \#l - 1) \Rightarrow \\
\qquad addtab\ l = (clipright\ l) \frown \langle tab \rangle \ \land \\
\quad (\#l \neq tabsize \lor \#(clipright\ l) \geq \#l - 1) \Rightarrow \\
\qquad addtab\ l = l
\end{array}
$$

These functions may be combined together to convert spaces in a line to tabs where appropriate:

$$
\begin{array}{|l}
unexpand : LINE \pfun LINE \\
\hline
\forall l : LINE \mid tab \notin \operatorname{ran} l \bullet \\
\quad unexpand\ l = \frown\!/((l \ \underline{split}\ tabsize) \fatsemi addtab)
\end{array}
$$

This function should only be applied to lines not containing tabs and in this case always reduces the length of the line, or leaves it the same.

$$\vdash \ \forall l : LINE \mid tab \notin \operatorname{ran} l \bullet \#(unexpand\ l) \leq \#l$$

Now consider a function to cut a sequence into a series of segments after each occurrence of a non-empty pattern.

$$
\begin{array}{|l}
[X] \\
\_ \ \underline{cut}\ \_ : (\operatorname{seq} X) \times (\operatorname{seq}_1 X) \pfun \operatorname{seq}(\operatorname{seq} X) \\
\hline
\forall p : \operatorname{seq}_1 X;\ s, t : \operatorname{seq} X \mid \neg\ (p \subseteq front(s \frown p)) \bullet \\
\quad \langle\rangle \ \underline{cut}\ p = \langle\rangle \ \land \\
\quad s \neq \langle\rangle \Rightarrow s \ \underline{cut}\ p = \langle s \rangle \ \land \\
\quad (s \frown p \frown t) \ \underline{cut}\ p = \langle s \frown p \rangle \frown (t \ \underline{cut}\ p)
\end{array}
$$

Note that combining these segments back together again results in the original sequence:

$$a, b : \operatorname{seq} X \vdash \frown / (a \underline{cut} b) = a$$

Trailing spaces can be substituted in a line segment with a trailing tab to bring its length to a tab column position.

$$addspace : LINE \rightarrow LINE$$

$$addspace \langle\rangle = \langle\rangle$$

$$\forall l : LINE \mid l \neq \langle\rangle \bullet$$
$$last\ l = tab \Rightarrow addspace\ l =$$
$$(front\ l) \frown (\langle space \rangle (tabsize + 1 - (\#l \bmod tabsize))) \wedge$$
$$last\ l \neq tab \Rightarrow addspace\ l = l$$

These functions may be combined to remove tabs from a line and replace them with spaces.

$$expand : LINE \rightarrow LINE$$

$$\forall l : LINE \bullet$$
$$expand\ l = \frown / ((l \underline{cut} \langle tab \rangle) \fatsemi addspace)$$

This function always increases the length of the line, or leaves it the same.

$$l : LINE \vdash \#(expand\ l) \geq \#l$$

Converting spaces to tabs and then back again leaves a line without tabs in it unchanged.

$$\vdash \forall l : LINE \mid tab \notin \operatorname{ran} l \bullet expand\ (unexpand\ l) = l$$

However converting tabs to spaces and back may not result in the same line.

$$\vdash \exists l : LINE \bullet unexpand\ (expand\ l) \neq l$$

The column width of a line can be defined.

$$width : LINE \rightarrow \mathbb{N}$$

$$\forall l : LINE \bullet$$
$$width\ l = \#(expand\ l)$$

The width of a line with tabs added is the same as the line with spaces in it.

$$l : LINE \vdash width\ (unexpand\ l) = width\ l$$

6.3.2 Lines

A document may be implemented by separating lines with a non-empty unique *newline* character sequence:

$$nl : \operatorname{seq}_1 CHAR$$

$$space \notin \operatorname{ran} nl$$
$$tab \notin \operatorname{ran} nl$$

These characters are not normally printable. For example, under UNIX this sequence

normally consists of the ASCII *line-feed* 'control' character. Under some other operating systems it can consist of a combination of the *carriage return* and *line-feed* characters. The characters should not occur within any line.

Consider a function which combines a sequence of segments using another sequence as a terminator for each of the segments.

$$
\begin{array}{|l}
\hline
[X] \\
\_ \underline{comb} \_ : (\operatorname{seq}(\operatorname{seq} X)) \times (\operatorname{seq} X) \rightarrow \operatorname{seq} X \\
\hline
\forall s, p : \operatorname{seq} X;\ ss : \operatorname{seq}(\operatorname{seq} X) \bullet \\
\quad \langle\rangle \ \underline{comb}\ p = \langle\rangle \ \land \\
\quad \langle s \rangle \ \underline{comb}\ p = s \frown p \ \land \\
\quad (\langle s \rangle \frown ss) \ \underline{comb}\ p = s \frown p \frown (ss \ \underline{comb}\ p) \\
\hline
\end{array}
$$

By using a *terminator* rather than a *separator* the empty sequence (e.g., an empty document) and a sequence containing the empty sequence (e.g. a document containing a single empty line) may be differentiated.

$$p : \operatorname{seq} X \vdash \langle\rangle \ \underline{comb}\ p = \langle\rangle$$

$$p : \operatorname{seq} X \vdash \langle\langle\rangle\rangle \ \underline{comb}\ p = p$$

Conversely, consider a function to separate a sequence into a series of segments using a non-empty pattern.

$$
\begin{array}{|l}
\hline
[X] \\
\_ \underline{sep} \_ : (\operatorname{seq} X) \times (\operatorname{seq}_1 X) \rightarrow \operatorname{seq}(\operatorname{seq} X) \\
\hline
\forall s : \operatorname{seq} X;\ p : \operatorname{seq}_1 X;\ ss : \operatorname{seq}(\operatorname{seq} X) \bullet \\
\quad s \ \underline{sep}\ p = ss \Leftrightarrow \\
\qquad ss \ \underline{comb}\ p = s \ \land \\
\qquad (\forall t : \operatorname{seq} X \mid t \in \operatorname{ran} ss \bullet \neg\ (p \subseteq t)\ ) \\
\hline
\end{array}
$$

Note that the sequence to be separated must be terminated with the pattern or be empty to be valid.

6.4 Files

A UNIX file is implemented as a sequence of characters [298], possibly containing *tab* characters and *newline* sequences.

$$FILE == \operatorname{seq} CHAR$$

On input, a number of such files are to be converted into a single document without *tab* characters or *newline* sequences.

$$
\begin{array}{|l}
\hline
\textit{PrePOS}_0 \\
\textit{TEXT} \\
\textit{input}? : \operatorname{seq} \textit{FILE} \\
\hline
\textit{text} = ((\frown / \textit{input}?)\ \underline{\textit{sep}}\ \textit{nl}) \fatsemi \textit{expand} \\
\hline
\end{array}
$$

On output, a single document is produced. Blanks may optionally be converted to tabs when this reduces the size of the document.

$$\textit{Blanks} ::= \textit{Yes} \mid \textit{No}$$

$$
\begin{array}{|l}
\hline
\textit{PostPOS}_0 \\
\textit{blanks}? : \textit{Blanks} \\
\textit{output}! : \textit{FILE} \\
\textit{TEXT}' \\
\hline
\textit{blanks}? = \textit{Yes} \Rightarrow \textit{output}! = \textit{text}'\ \underline{\textit{comb}}\ \textit{nl} \\
\textit{blanks}? = \textit{No} \Rightarrow \textit{output}! = (\textit{text}' \fatsemi \textit{unexpand})\ \underline{\textit{comb}}\ \textit{nl} \\
\hline
\end{array}
$$

These schemas may be combined to produce a new specification for the behaviour of an implementation of the tool under UNIX.

$$\textit{POS} \mathrel{\widehat=} \textit{PrePOS}_0 \land \textit{POS}_0 \land \textit{PostPOS}_0$$

6.5 Conclusion

This specification is based on a simple text processing tool which was written in the programming language C [244] for use under the UNIX operating system. There are some differences, but these and the *manual page* for this tool are included on page 118 for comparison.

Of course, normally a formal specification should be written *before* an implementation is produced. In this case this was not so because the author was not enlightened when he wrote the original program some years ago. However there are still benefits in producing a *post hoc* specification. This can be useful if a product is to be re-engineered [319]. It can also help in improving the documentation of an existing system [41, 80].

This chapter shows that text processing concepts may be documented using a formal notation. By formalizing a system, ambiguity and misunderstanding are reduced. Additionally, reasoning about the properties of the system becomes easier and can be done with more confidence. Software developed from formal specifications is likely to contain less errors. Studies on industrially sized examples of software development have confirmed this view [247]. For these reasons, it is hoped that formal specification will become more widely used in the field of practical software engineering in the future.

The specification was previous published as an article [44]; it is included by kind permission of the Institution of Electrical Engineers (IEE), UK.

6.6 UNIX Manual Page

The *manual entry* for the actual UNIX tool on which this specification is based is included here to allow the reader to compare the Z specification with a more familiar English style description. Note however that the specification given and the UNIX tool described are not identical. If the tool had originally been designed using Z then it would have undoubtedly been more like the specification given in this chapter. The main differences may be summarized as follows:

- *Pos* allows the positioning to be optionally determined by the position of the first line in each input file.
- *Pos* does not format lines with a '**.**' in the first column.
- Only the leading spaces are converted to tabs on output by *pos*.

It would be a relatively mundane matter to update the specification presented here to match *pos* exactly, but this would increase its length unduly and is thus left as an exercise for the interested reader. For this reason, the specification and the program do not correspond exactly. It is unlikely that the program will be updated in this instance since modifications would affect users unduly; the study was undertaken as an exercise in clear specification rather than re-engineering.

POS(1) **UNIX Manual** **POS(1)**

NAME

pos – simple text column positioning

SYNOPSIS

pos [**–b**] [column] [file ...]

DESCRIPTION

Pos is a simple text formatter which reads the concatenation of input files (or standard input if none are given) and produces on standard output a version of its input with lines positioned according to the *column* parameter. *Column* is a number specifying a column position starting from column 1. If it is unsigned then text is centered about that position. If it is negative, `-`*column* then the left hand margin of the text is positioned at the specified column. If it is signed positive, `+`*column* then the right hand margin of the text is positioned at that column. If the column parameter is '–', '`+`' or '`0`' then the positioning is calculated from the first line in each file. The default value for *column* is 40.

All tabs are converted to spaces, although tabs are used for leading space where appropriate unless the **–b** option is used in which case only spaces are used on output. Interword spacing on each line is preserved. If a line is too long then it is simply outputted with its leading white spaces removed so that no non-blank text is lost. Lines starting with a '**.**' are not formatted but are simply outputted as read.

Pos is meant to format headings and other short pieces of text. For instance, within visual mode of the *ex* editor (e.g., *vi*) the command

```
!}pos
```

will center a paragraph.

AUTHOR

Jonathan Bowen, Oxford University

SEE ALSO

colrm(1), expand(1), fmt(1), nroff(1)

BUGS

The program is designed to be simple and fast – for more complex operations, the standard text processors are likely to be more appropriate. Control characters other than tabs will confuse *pos*.

Chapter 7

An Event-based Input System

In this chapter a design is given for an event-based input system for use on graphics workstations that overcomes problems associated with some earlier designs. An interface is specified which allows a client to select events to be queued from the set of those available, get events from the input queue, put events in the input queue, flush the input queue, and connect a set of polled events to a queued event. The system has been designed with the intention of being implementable on a number of systems and has also been formally specified using the Z specification language. The system has been implemented on UNIX *workstations.*

7.1 Motivation

Many existing input systems have been based on the design of a particular device or subsystem and/or have been specialized to the needs of a particular window system. This has several problems which are discussed in [80]. The event queue system is independent of any particular application or window system. It provides a set of general purpose interfaces and is extensible to accommodate new input devices and event types. This results in a system that is useful to a diverse range of applications.

The event queue is a system module analogous to the UNIX operating system [37] `tty` handler for dealing with the standard ASCII terminal user interface. It provides a set of procedures and data structures which a physical device driver, or client (user process) may access. The event queue provides a set of logical devices with fixed semantics. A given logical device abstracts from the large variety of physical devices which may possibly implement it. The event queue system was first implemented under UNIX on a large VAX system (11/785) with a large variety of attached input devices, and later on MicroVAX II workstations [80].

Several existing systems were considered in the design of the present system. Shortcomings of existing input systems (such as those associated with the X Window System [357, 358, 359]) influenced this design.

In this chapter, Z is used to describe the operation of the system formally. An abstract state of the system is presented, and then the effect of individual library routines on the state is given. In addition, the corresponding C programming language [244] declarations are included in italics before the formal description where appropriate to aid those familiar with the C programming language.

7.2 Type Definitions

The types used in the Z specification in this chapter are all ranges of integers and have been implemented as follows:

$$FALSE == 0$$
$$TRUE == 1$$
$$boolean == \{FALSE, TRUE\}$$

$$char == 0 \,..\, 2^7 - 1$$
$$short == -2^{15} \,..\, 2^{15} - 1$$
$$int == -2^{31} \,..\, 2^{31} - 1$$

$$unsigned\_char == 0 \,..\, 2^8 - 1$$
$$unsigned\_short == 0 \,..\, 2^{16} - 1$$
$$unsigned\_int == 0 \,..\, 2^{32} - 1$$

7.3 Input Device Events

Activity on input devices can cause entries to be made in the event queue. An event is represented by a device identifier (an unsigned 16-bit 'short' word) and an associated value (a signed short word).

```
typedef unsigned short qDevice
typedef short qValue
```

$$qDevice == unsigned\_short$$
$$qValue == short$$

Event

$$device : qDevice$$
$$value : qValue$$

There are three different types of device in the system described here: button, valuator and keyboard. Valuators may be absolute or relative.

$$qType == unsigned\_char$$

$$Button, RelValuator, AbsValuator, Keyboard : qType$$

$$\langle Button, RelValuator, AbsValuator, Keyboard \rangle \in \mathrm{iseq}\, qType$$

$$Valuator == \{AbsValuator, RelValuator\}$$

All devices return a *qValue* and are hence constrained to have a 16 bit integer range.* Button devices return *boolean* (true/false) values. The *qValue* associated with a button

* The *qValue* type can be changed if it is decided that there is a need for a device with additional range. However, the goal is to keep a *qValue* as small as possible and still be sufficient to contain the value for any possible device.

event will be either 0 indicating that the button went up, or 1 indicating that the button went down. Valuator devices have a range value associated with them. For example *MOUSEX*, *TABLETY*, *KNOB*15, or *CLOCK* are valuator devices. Valuator devices may be relative or absolute. For example we might have a *MOUSEX* device which returns absolute mouse *x* positions, or a *MOUSEDX* device which returns relative mouse *x* positions (deltas) or both. Keyboard devices return character code values. For example, the device *ASCIIKEYBOARD* is a device which returns ASCII values in the low order 7 bits of *qValue*.

7.4 Abstract State

The conceptual state of the system is introduced briefly here. Each component is covered more fully when the operations are introduced.

The state of the system includes a finite number of devices. Each of these has a *type* and a *value*. Valuator devices also have a *delta* resolution.

Device

$$\begin{array}{l} type : qType \\ value : qValue \\ delta : short \end{array}$$

The devices may be enabled. A sequence of events awaits processing by the client. A number of devices may be bound to a device. A pair of devices may be associated with the *x* and *y* coordinates of the cursor.

State

$$\begin{array}{l} devices : qDevice \pfun Device \\ enabled : \mathbb{F}\, qDevice \\ events : \mathrm{seq}\, Event \\ bindings : qDevice \pfun \mathrm{seq}\, qDevice \\ cursor : qDevice \times qDevice \\ \hline enabled \subseteq \mathrm{dom}\, devices \\ \mathrm{dom}\, bindings = \mathrm{dom}\, devices \\ 0 \notin \mathrm{dom}\, devices \end{array}$$

Initially, much of the state is zero or empty. A number of devices are configured in the system, but there are no enabled or bound devices.

InitState

$$
\begin{array}{l}
State' \\
\hline
devices' \neq \emptyset \\
enabled' = \emptyset \\
events' = \langle\rangle \\
\operatorname{ran} bindings' = \{\langle\rangle\} \\
cursor' = (0, 0) \\
\forall\, dev : qDevice \mid dev \in \operatorname{dom} devices' \bullet \\
\quad (devices'\ dev).value = 0 \land \\
\quad (devices'\ dev).delta = 0
\end{array}
$$

7.5 Changes of State

Operations change the state of the system. However the device types always remain the same. They depend on their values at initialization time.

ΔState

$$
\begin{array}{l}
State \\
State' \\
\hline
\operatorname{dom} devices' = \operatorname{dom} devices \\
\forall\, dev : qDevice \mid dev \in \operatorname{dom} devices \bullet \\
\quad (devices'\ dev).type = (devices\ dev).type
\end{array}
$$

Some operations do not affect the state. (Note that in practice the actual device values change asynchronously but this does not affect the abstract specification.)

$$\Xi State \mathrel{\widehat=} [\Delta State \mid \theta State' = \theta State]$$

For most operations, only a small part of the state is changed. The following schemas are useful in the definition of subsequent operations, by specifying that all but one of the state components is unaffected.

$$
\begin{array}{l}
\Phi Device \mathrel{\widehat=} \Delta State \land (\, \Xi State \setminus (devices) \,) \\
\Phi Enable \mathrel{\widehat=} \Delta State \land (\, \Xi State \setminus (enabled) \,) \\
\Phi Event \mathrel{\widehat=} \Delta State \land (\, \Xi State \setminus (events) \,) \\
\Phi Binding \mathrel{\widehat=} \Delta State \land (\, \Xi State \setminus (bindings) \,) \\
\Phi Cursor \mathrel{\widehat=} \Delta State \land (\, \Xi State \setminus (cursor) \,)
\end{array}
$$

7.5.1 Asynchronous events

Device values may change asynchronously. Button values can only be 0 and 1. For valuator devices, the change must be greater than the *delta* resolution of the device. Keyboards only return ASCII characters.

AsyncChange

$\Phi Device$
$device : qDevice$
$Device$
$Device'$

$\theta Device = devices(device)$
$type' = type$
$delta' = delta$

$type = Button \Rightarrow value' = \{0 \mapsto 1, 1 \mapsto 0\} value$
$type = AbsValuator \Rightarrow abs(value' - value) \geq delta$
$type = RelValuator \Rightarrow value' \geq delta$
$type = Keyboard \Rightarrow value' \in char$

$devices' = devices \oplus \{device \mapsto \theta Device'\}$

If the device is enabled, this causes an event. This is added to the end of the event queue together with any associated enabled bound device events. The corresponding current device value is recorded in each event.

QueueEvent

$\Phi Event$
$Event$
$bound\_devs : \operatorname{seq} qDevice$
$bound\_events : \operatorname{seq} Event$

$device \in enabled$
$value = (devices\, device).value$
$events' = events \frown \langle \theta Event \rangle \frown bound\_events$
$bound\_devs = bindings(device) \upharpoonright enabled$
$\#bound\_events = \#bound\_devs$

$\forall i : \mathbb{N} \mid i \in \operatorname{dom} bound\_devs \bullet$
$\quad (bound\_events\, i).device = bound\_devs(i) \land$
$\quad (bound\_events\, i).value = (devices\, (bound\_devs\, i)).value$

An asynchronous event consists of an asynchronous change in a value of a device followed by the addition of the event and bound events if the device is enabled.

$$AsyncEvent \mathrel{\widehat=} AsyncChange \fatsemi (\Xi State \oplus QueueEvent)$$

The notation $AsyncEvent^*$ is used to denote an arbitrary number of consecutive asynchronous events connected using *schema composition*. We can define

$$AsyncEvent0 \mathrel{\widehat=} \Xi State$$
$$AsyncEvent1 \mathrel{\widehat=} AsyncEvent$$
$$AsyncEvent2 \mathrel{\widehat=} AsyncEvent \fatsemi AsyncEvent$$

and so on. Using these definitions,

$$AsyncEvent^* \mathrel{\widehat=} AsyncEvent0 \lor AsyncEvent1 \lor AsyncEvent2 \lor \ldots$$

These asynchronous events may be thought of as occurring in sequence with operations invoked by the client. Each operation can be considered as being *atomic*.

7.6 System Operations

7.6.1 Device types

Each device has an associated identifier. It is possible to determine the type of the device with the use of the following requests:

```
boolean  isbutton(device)
    qDevice  device;

boolean  isvaluator(device)
    qDevice  device;

boolean  isrelative(device)
    qDevice  device;

boolean  isabsolute(device)
    qDevice  device;

boolean  iskeyboard(device)
    qDevice  device;
```

A partially specified schema may be used to capture the general features of these operations. Then each operation may be defined in terms of this schema.

ΦIs

$\Xi State$
$device? : qDevice$
$qtype : \mathbb{F}\, qType$
$is : boolean$

$(devices\, device?).type \in qtype \Rightarrow is = TRUE$
$(devices\, device?).type \notin qtype \Rightarrow is = FALSE$

$$IsButton \mathrel{\widehat=} [\, \Phi Is[isbutton!/is] \mid qtype = \{Button\}\,]$$

$$IsValuator \mathrel{\widehat=} [\, \Phi Is[isvaluator!/is] \mid qtype = Valuator\,]$$

$$IsRelative \mathrel{\widehat=} [\, \Phi Is[isrelative!/is] \mid qtype = \{RelValuator\}\,]$$

$$IsAbsolute \mathrel{\widehat=} [\, \Phi Is[isabsolute!/is] \mid qtype = \{AbsValuator\}\,]$$

$$IsKeyboard \mathrel{\widehat=} [\, \Phi Is[iskeyboard!/is] \mid qtype = \{Keyboard\}\,]$$

The client will know how to interpret the value associated with an event, based on the type of device and perhaps the detailed device identifier. The semantics of the value are device dependent.

7.6.2 Queuing and dequeuing input events

When a device value changes, an event is entered in the input event-queue if this device has been selected for queuing by the client with a *qdevice*() request.

```
int qdevice(device)
    qDevice  device;
```

QDevice

$$\Phi Enable$$
$$device? : qDevice$$
$$enabled' = enabled \cup \{device?\}$$

Once a device has been selected by *qdevice*() an event will be placed in the event queue whenever the device changes value. Devices which have not been queued will not cause events to be entered in the event queue.

Events which are presently being queued may be deselected for queuing with the *unqdevice*() request.

```
int unqdevice(device)
     qDevice  device;
```

UnqDevice

$$\Phi Enable$$
$$device? : qDevice$$
$$enabled' = enabled \setminus \{device?\}$$

7.6.3 Event filtering

Certain devices such as *MOUSEX* or *CLOCK* are either high resolution or 'noisy' and may thus generate more events than the client actually wishes to see. The *threshold*() interface allows the client to specify that a minimum change must occur in the device's value before an event is entered in the queue.

```
int threshold(device, delta)
     qDevice  device;
     short  delta;
```

Threshold

$\Phi Device$
$device? : qDevice$
$delta? : short$
$Device$
$Device'$

$\theta Device = devices(device?)$
$type' = type$
$value' = value$
$delta' = delta?$
$devices' = devices \oplus \{device? \mapsto \theta Device'\}$

If the *device* has been selected for queuing by *qdevice*() then a change of at least *delta* must occur on its associated value before an event will be placed in the queue.†

7.6.4 Reading events

The client obtains the next input event from the event queue with the *getevent*() request.

```
qDevice getevent(pValue)
     qValue  *pValue;
```

GetEvent

$\Phi Event$
$pValue! : qValue$
$getevent! : qDevice$
$Event$

$\theta Event = head\ events$
$pValue! = value$
$getevent! = device$
$events' = tail\ events$

This operation blocks until an event is available. Hence a number of events may be necessary beforehand.

$$GetEvent_1 \mathrel{\widehat=} [AsyncEvent^* \mid \#events' > 0] \fatsemi GetEvent$$

Recall that an input event consists of two values – one identifying the device which generated the event and one giving the value associated with the device at the time of the event. *getevent*() returns the device identifier, and places the value associated with the device for this event in the location pointed to by *pValue*. The client's usage is:

```
short  value;
```

† This interface only really makes sense for valuator type devices. There is also a problem for devices which are relative versus absolute as this interface implies that the system knows how to determine what is a change as opposed to an absolute value from the device.

```
qDevice  device;
     ...
device = getevent(&value);
```

For efficiency we are often interested in reading a number of events in one procedure invocation. The *getevents*() request performs this function.

```
qDevice getevents(count, EventArray)
     int  count;
     struct {
         qDevice  Device;
         qValue  Value;
     } EventArray[count];
```

GetEvents

$\Phi Event$
$count? : int$
$EventArray! : int \pfun Event$
$getevents! : qDevice$

$EventArray! = succ \fatsemi (events \ \underline{for} \ count?)$
$getevents! = (head \ events).device$
$events' = events \ \underline{after} \ count?$

getevents returns the device associated with the first event and returns an array of *count* events in *EventArray*. *getevents*() does not return until all *count* requested events have been returned.

Similarly to *getevent*(), a number of events may be necessary beforehand.

$$GetEvents_1 \mathrel{\widehat=} [AsyncEvent^*; \ count? : int \mid \#events' \geq count?] \fatsemi GetEvents$$

7.6.5 Posting input events

Events may be 'posted' (i.e., placed) at the end of the input queue with the *postevent* request.

```
int postevent(count, pDevice, pValue)
     int  count;
     qDevice  *pDevice;
     qValue  *pValue;
```

PostEvent

$$\Phi Event$$
$$count? : int$$
$$pDevice? : int \pfun qDevice$$
$$pValue? : int \pfun qValue$$

$$events' = events \frown (\mu s : \mathrm{seq}\, Event \mid \#s = count? \land (\forall i : 1 \mathinner{..} count? \bullet (s\, i).device = pDevice?(i-1) \land (s\, i).value = pValue?(i-1)))$$

count events are entered in the event queue with device id's given by the list *pDevice* and values given in the list *pValue*. *postevent*() allows a client to simulate physical or virtual events generated by some other device, or to implement a pseudo-device to synthesize some new class of events (e.g., for a window manager or other software process). It is important that the client be able to enter several events into the queue as an atomic action. *postevent*() assures that the list of events passed in will be placed contiguously in the event queue, uninterrupted by other events which may occur during the call.

7.6.6 Testing the input queue

Since the *getevent*() request is synchronous (i.e., it will suspend the caller until there is an event in the input queue to be read), the client may not wish to use it. The *qtest*() request allows the client to determine whether there is an event in the queue.

```
qDevice qtest()
```

QTest

$$\Xi State$$
$$qtest! : qDevice$$

$$events \neq \langle\rangle \Rightarrow qtest! = (head\, events).device$$
$$events = \langle\rangle \Rightarrow qtest! = 0$$

qtest() returns the device identifier associated with the first input event in the queue, or 0 if the queue is empty. The client may use *qtest*() to provide a non-blocking use of the input queue. This is implemented by using *qtest*() to see that there is an event and only then performing a *getevent*() or *getevents*() if input is available.

7.6.7 Grouping input events

It is often necessary to observe the state of several devices when a particular event occurs. With the *connectevent*() request the client directs the system that another input event is to be placed in the event queue when a particular event occurs.

```
int  connectevent(device, bound_device)
     qDevice  device, bound_device;
```

$$MaxBindings : \mathbb{N}_1$$

ConnectEvent

$$\Phi Binding$$
$$device?,$$
$$bound\_device? : qDevice$$
$$connectevent! : int$$
$$bound\_devs : \operatorname{seq} qDevice$$

$$bound\_devs = bindings(device?)$$

$$\#bound\_devs < MaxBindings \Rightarrow$$
$$\quad bindings' =$$
$$\quad\quad bindings \oplus \{device? \mapsto bound\_devs \frown \langle bound\_device? \rangle\} \wedge$$
$$\quad connectevent! > 0$$

$$\#bound\_devs \geq MaxBindings \Rightarrow$$
$$\quad bindings' = bindings \wedge$$
$$\quad connectevent! < 0$$

The *bound_device* will be examined and register an event whenever the device *device* generates an event. When an event occurs on *device*, an event will be placed in the event queue for that device, and for each device bound to this device by *connectevent*().

This interface allows the client to cause the information associated with several devices to be entered in the event queue whenever a certain event occurs. *connectevent*() is simply called more than once, once for each device to be bound to the queued device. Devices which have been bound to a queued device by *connectevent*() will cause the system to enter events associated with the bound devices immediately after the queued event, without any intervening events, and in the order established by the invocations to *connectevent*(). There is an implementation limit on the number of devices which can be bound to another device with *connectevent*().[‡] *connectevent*() returns an error code less than 0 if it was unable to honour the request, positive non-zero if successful.

The association created between *device* and *bound_device* by *connectevent*() is directed. That is, if we bind *TABLETX* to *BUTTON*156 with:

```
connectevent( BUTTON156, TABLETX );
```

This establishes that when *BUTTON*156 changes we also produce a *TABLETX* event, but not the converse. We may of course establish the other relationship with:

```
connectevent( TABLETX, BUTTON156 );
```

The relationships may be thought of as a depth-one directed graph with device as the root node and the *bound_device*(s) as the related leaf node(s).

An example of the use of *connectevent*() is as follows – we wish to note the mouse's *x* and *y* position whenever the mouse's left button is depressed. We do not wish to know the mouse's position at any other time but when the left button is depressed, and

[‡] This should be some reasonable number such as 8 or 16, based on the type of devices, and the things that clients are likely to want to do. These implementation limits may be changed as application requirements direct.

therefore do not wish to select input events based on *MOUSEX* or *MOUSEY*. We will select to queue the mouse left button and then bind the mouse *x* value and mouse *y* value to it with *connectevent*():

```
qdevice( MOUSELEFT );
connectevent( MOUSELEFT, MOUSEX );
connectevent( MOUSELEFT, MOUSEY );
```

It may be necessary to undo the binding between two devices. This function is performed by the *disconnectevent*() request.

```
int disconnectevent(device, bound_device)
     qDevice device, bound_device;
```

DisconnectEvent

$$
\begin{array}{l}
\Phi Binding \\
device?, \\
bound\_device? : qDevice \\
disconnectevent! : int \\
bound\_devs : \operatorname{seq} qDevice \\
\hline
bound\_devs = bindings(device?) \\
bound\_device? \in \operatorname{ran} bound\_devs \Rightarrow \\
\quad bindings' = bindings \oplus \\
\qquad \{device? \mapsto bound\_devs \upharpoonright (qDevice \setminus \{bound\_device?\})\} \land \\
\quad disconnectevent! \neq 0 \\
bound\_device? \notin \operatorname{ran} bound\_devs \Rightarrow \\
\quad bindings' = bindings \land \\
\quad disconnectevent! = 0
\end{array}
$$

The request returns non-zero on success, and 0 for failure. Failure will be due to the fact that the *bound_device* referred to was not bound to the device.

7.6.8 Cursor position

The cursor is closely related to the event queue mechanism. The cursor must derive its current position (*CURSORX* and *CURSORY*) based on two input devices with range value (tablet *x* and *y*, knobs, mouse *x* and *y*, or the like). The *attachcsr*() request allows the client to determine which pair of valuators will be used to determine the cursor position.

```
int attachcsr(xdevice, ydevice)
     qDevice xdevice, ydevice;
```

AttachCsr

$$
\begin{array}{l}
\Phi Cursor \\
xdevice?, \\
ydevice? : qDevice \\
\hline
cursor' = (xdevice?, ydevice?)
\end{array}
$$

CURSORX and *CURSORY* are devices in their own right and may be queued with *qdevice* like any other device.

7.7 Implementation Notes

Pseudo-devices and extensibility of the event-queue system: *ASCIIKEYBOARD* is implemented as a 'pseudo-device' in the system. That is, it is implemented as a simple software finite state machine running on top of an array of buttons (the buttons of the raw up-down encoded keyboard). The system is easily extensible both by adding new devices of the existing classes or by implementing new pseudo-devices. If a new pseudo-device is to be built in the system the appropriate software module which implements it must be added. In addition, for new devices added to the system a device identifier should be placed (defined) in the system file (`qevent.h`). Pseudo-devices may also be implemented by the client. This may be done either as a layer on top of the system's queue mechanism, or by the client entering events into the system event queue. This is true since the interfaces described above allow the client to enter events into the system event queue as well as get them. Although the system implements the ASCII keyboard device, it would be equally easy for a client layer to implement it. To do this the client would queue all of the button devices associated with the keyboard and then implement the keyboard state machine as a layer on top of the system's queue mechanism. It would in turn present a queue mechanism to a higher level client. All of the system defined events would be passed through to the higher level queue as usual.

Implementation issues for connectevent() and client devices: In order for a device to be bound to another device with *connectevent*() it must be possible for the system to poll the *bound_device*. For a client-implemented pseudo-device this has the implication that it may not be possible for the system to bind that pseudo-device to another device with *connectevent*(). There would have to be a mechanism for the system to obtain a value for the given pseudo-device. One possible solution is to have the system keep track of the last value for non system-implemented devices which have had entries made with *postevent*(). The system could then use this value at the time of an event for which that pseudo-device had been bound. Another scenario is to have the pseudo-device implementation provide a routine which the system can call to obtain its value on demand (a polling routine). Implementing pseudo-devices which integrate with the system level queue is similar to writing system device drivers in any case.

7.8 Types Revisited

In this chapter, *qValue* has been implemented as a signed short integer. A more flexible approach would be to allow *qValue* either to be a single value, or a sequence of values. We could define *qValue* as a labelled disjoint union and update the specification given accordingly.

$$qValue ::= val\langle\langle short\rangle\rangle \mid str\langle\langle \mathrm{seq}\, char\rangle\rangle$$

In practice, this could consist of Boolean flag (in the sign bit) followed by a 15-bit value or a 15-bit length (in 8-bit bytes) optionally followed by a byte string.

IV

Instruction Sets

Hardware as well as software can be specified using the Z notation. The basic concepts concerning machine words and their manipulation are given in Chapter 8. These definitions are used in Chapter 9 to specify part of the instruction set for the Inmos Transputer microprocessor family.

Chapter 8

Machine Words

Generic operations on machine words are described at the bit-level that are useful in the specification of microprocessors at the instruction set level and below. These definitions are used for the specification of part of an instruction set in Chapter 9.

8.1 Word Organization

The basic unit of data manipulated by a microprocessor is the bit.

$$Bit ::= \mathtt{0} \mid \mathtt{1}$$

These are organized into words. Bit positions within a word are numbered consecutively from zero upwards:

$$Word == \{ w : \mathbb{N} \pfun Bit \mid \#w > 0 \land \operdom w = 0 \upto \#w - 1 \}$$

The values of the *least significant bit* (*LSB*) and the *most significant bit* (*MSB*) of a word are often of special interest.

$$\begin{array}{|l} LSB, MSB : Word \to Bit \\ \hline \forall w : Word \bullet \\ \quad LSB\, w = w\, 0 \land \\ \quad MSB\, w = w\, (\#w - 1) \end{array}$$

Bit values correspond to numeric values:

$$\begin{array}{|l} bitval : Bit \rightarrowtail \mathbb{N} \\ \hline bitval = \{ \mathtt{0} \mapsto 0, \mathtt{1} \mapsto 1 \} \end{array}$$

The *unsigned* value of a word may be defined by:

$$\begin{array}{|l} val : Word \to \mathbb{N} \\ \hline \forall w : Word \bullet \\ \quad \#w = 1 \Rightarrow val\, w = bitval(LSB\, w) \land \\ \quad \#w > 1 \Rightarrow val\, w = bitval(LSB\, w) + 2 * val(succ \fatsemi w) \end{array}$$

In general, *val* is not an injective function for words of variable size:

$$\vdash \exists w_1, w_2 : Word \mid w_1 \neq w_2 \bullet val\, w_1 = val\, w_2$$

However if attention is restricted to words of a fixed size, the restricted function is injective. Thus a word may be characterized by specifying its value and its size.

$$\vdash \forall w_1, w_2 : Word \mid \#w_1 = \#w_2 \bullet val\, w_1 = val\, w_2 \Rightarrow w_1 = w_2$$

All the bits in a word may be set to a particular value:

$$\_ \underline{set} \_ : Word \times Bit \rightarrow Word$$

$$\forall w : Word;\ b : Bit \bullet$$
$$w \underline{set}\, b = w \fatsemi \{ \mathtt{0} \mapsto b, \mathtt{1} \mapsto b \}$$

Often this value is zero:

$$zero : Word \rightarrow Word$$

$$\forall w : Word \bullet$$
$$zero\, w = w \underline{set}\, \mathtt{0}$$

The maximum unsigned value that may be stored in a word is given by *maxval*:

$$maxval : Word \rightarrow \mathbb{N}$$

$$\forall w : Word \bullet$$
$$maxval\, w = val(w \underline{set}\, \mathtt{1})$$

It is convenient to define a function to generate words containing a particular value.

$$wrd : \mathbb{N}_1 \rightarrow \mathbb{Z} \rightarrowtail Word$$

$$\forall size : \mathbb{N}_1;\ value : \mathbb{Z};\ w : Word \bullet$$
$$wrd\, size\, value = w \Leftrightarrow$$
$$(\#w = size \land$$
$$(\exists_1 v : \mathbb{Z} \bullet val\, w = value + succ(maxval\, w) * v))$$

If a value which is too large or too small (i.e., negative) is provided, it is adjusted by a multiple of one more than the greatest unsigned value that the word will hold so that it will fit.

As well as the successor function *succ*, the inverse predecessor function *pred* is often useful:

$$pred == succ^{\sim}$$

Words may be concatenated to produce a longer word.

$$\_ \frown \_ : Word \times Word \rightarrow Word$$

$$\forall w_1, w_2 : Word \bullet$$
$$w_1 \frown w_2 = w_1 \cup (pred^{\#w_1} \fatsemi w_2)$$

Note that this definition of *concatenation* together with the definition of *LSB* given earlier mean that the machine is 'little-endian'; for words of any size the least significant bit is always the 0^{th} bit of the word.

Generalized concatenation allows a (non-empty) sequence of words to be concatenated into one word.

$$\frown/ : \mathrm{seq}_1\, \mathit{Word} \to \mathit{Word}$$

$$\forall\, w : \mathit{Word} \bullet \frown/\langle w \rangle = w$$

$$\forall\, s, t : \mathrm{seq}_1\, \mathit{Word} \bullet \frown/(s \frown t) = (\frown/\, s) \frown (\frown/\, t)$$

Sometimes the highest (most significant) bit set in a word is of interest:

$$\mathit{HighestSetBit} : \mathit{Word} \to \mathbb{N}$$

$$\forall\, w : \mathit{Word} \bullet \mathit{val}\, w = 0 \Rightarrow \mathit{HighestSetBit}\, w = 0 \land \mathit{val}\, w \neq 0 \Rightarrow \mathit{HighestSetBit}\, w = \mathit{max}(\mathrm{dom}(w \rhd \{1\}))$$

The Transputer described in Chapter 9 operates on 8-bit bytes of data and also on words each consisting of a small number of bytes. The constant *BytesPerWord* is left undefined here.

$$\mathit{BytesPerWord} : \mathbb{N}$$

From this the number of bits in a word may be derived:

$$\mathit{WordLength} : \mathbb{N}$$

$$\mathit{WordLength} = 8 * \mathit{BytesPerWord}$$

Bytes and Transputer words may be defined by:

$$\mathit{Byte} == \{\, w : \mathit{Word} \mid \#w = 8 \,\}$$

$$\mathit{Tword} == \{\, w : \mathit{Word} \mid \#w = \mathit{WordLength} \,\}$$

Sometimes it is convenient to construct words with a particular value using the following abbreviation:

$$\mathit{Twrd} == (\mathit{wrd}\ \mathit{WordLength})$$

It is helpful to give names to the largest positive and smallest negative *signed* integers and other commonly used values that can be represented in a *Tword*.

$$\mathit{MostNeg}, \mathbf{-1}, \mathbf{0}, \mathbf{1}, \mathit{MostPos} : \mathit{Tword}$$

$$\mathit{MostNeg} = \mathit{Twrd}(-2^{\mathit{WordLength}-1})$$
$$\mathbf{-1} = \mathit{Twrd}(-1)$$
$$\mathbf{0} = \mathit{Twrd}(0)$$
$$\mathbf{1} = \mathit{Twrd}(1)$$
$$\mathit{MostPos} = \mathit{Twrd}(2^{\mathit{WordLength}-1} - 1)$$

8.2 Operations on Words

8.2.1 Bitwise logical functions

'$\sim$' simply complements a bit (logical *NOT*).

$$\sim : \mathit{Bit} \rightarrowtail\!\!\!\!\!\!\rightarrow \mathit{Bit}$$
$$\sim = \{\mathtt{0} \mapsto \mathtt{1}, \mathtt{1} \mapsto \mathtt{0}\}$$

Bits may be combined by logical *AND* ('$\bullet$'), bitwise logical (inclusive) *OR* ('$^+$') and bitwise logical (exclusive) *XOR* ('$\oplus$').

$$\_ \bullet \_, \_ ^+ \_, \_ \oplus \_ : \mathit{Bit} \times \mathit{Bit} \rightarrow \mathit{Bit}$$
$$(\_ \bullet \_) = \{(\mathtt{0},\mathtt{0}) \mapsto \mathtt{0}, (\mathtt{0},\mathtt{1}) \mapsto \mathtt{0}, (\mathtt{1},\mathtt{0}) \mapsto \mathtt{0}, (\mathtt{1},\mathtt{1}) \mapsto \mathtt{1}\}$$
$$(\_ ^+ \_) = \{(\mathtt{0},\mathtt{0}) \mapsto \mathtt{0}, (\mathtt{0},\mathtt{1}) \mapsto \mathtt{1}, (\mathtt{1},\mathtt{0}) \mapsto \mathtt{1}, (\mathtt{1},\mathtt{1}) \mapsto \mathtt{1}\}$$
$$(\_ \oplus \_) = \{(\mathtt{0},\mathtt{0}) \mapsto \mathtt{0}, (\mathtt{0},\mathtt{1}) \mapsto \mathtt{1}, (\mathtt{1},\mathtt{0}) \mapsto \mathtt{1}, (\mathtt{1},\mathtt{1}) \mapsto \mathtt{0}\}$$

These definitions are easily upgraded to bitwise logical operations on words. The *one's complement* of a word is given by inverting all its bits:

$$\sim : \mathit{Word} \rightarrowtail\!\!\!\!\!\!\rightarrow \mathit{Word}$$
$$\forall w : \mathit{Word} \bullet \sim w = w \fatsemi \sim$$

For dyadic operators, pairs of bits must be considered:

$$\mathit{WordPair} == \{w : \mathbb{N} \pfun (\mathit{Bit} \times \mathit{Bit}) \mid \#w > 0 \land \operatorname{dom} w = 0 \mathinner{..} \#w - 1\}$$

$$\_ \underline{\mathit{pair}} \_ : \mathit{Word} \times \mathit{Word} \rightarrow \mathit{WordPair}$$
$$\forall w_1, w_2 : \mathit{Word} \bullet w_1 \underline{\mathit{pair}} w_2 = \{i : \mathbb{N} \mid i \in \operatorname{dom} w_1 \cap \operatorname{dom} w_2 \bullet i \mapsto (w_1\, i, w_2\, i)\}$$

$$\_ \bullet \_, \_ ^+ \_, \_ \oplus \_ : \mathit{Word} \times \mathit{Word} \rightarrow \mathit{Word}$$
$$\forall w_1, w_2 : \mathit{Word} \bullet w_1 \bullet w_2 = (w_1 \underline{\mathit{pair}} w_2) \fatsemi (\_ \bullet \_) \land w_1 ^+ w_2 = (w_1 \underline{\mathit{pair}} w_2) \fatsemi (\_ ^+ \_) \land w_1 \oplus w_2 = (w_1 \underline{\mathit{pair}} w_2) \fatsemi (\_ \oplus \_)$$

The logical values *false* and *true* are represented by *Word*s with values 0 and 1 respectively:

$$\mathit{False} == \mathbf{0}$$
$$\mathit{True} == \mathbf{1}$$

8.2.2 Shift functions

A word may be shifted *left* or *right*. Zeroes are shifted into the vacant positions.

$$
\begin{array}{|l}
\_ \ll \_ : \mathit{Word} \times \mathbb{N} \to \mathit{Word} \\
\_ \gg \_ : \mathit{Word} \times \mathbb{N} \to \mathit{Word} \\
\hline
\forall\, w : \mathit{Word} \bullet \\
\quad w \ll 0 = w \land \\
\quad w \gg 0 = w \land \\
\quad w \ll 1 = (\{\, \#w \,\} \lhd \mathit{pred} \fatsemi w) \cup \{\, 0 \mapsto \mathtt{0} \,\} \land \\
\quad w \gg 1 = \{\, \#w - 1 \mapsto \mathtt{0} \,\} \cup (\mathit{succ} \fatsemi w) \land \\
\quad (\forall\, n : \mathbb{N} \bullet \\
\qquad w \ll (n+1) = (w \ll n) \ll 1 \land \\
\qquad w \gg (n+1) = (w \gg n) \gg 1\,)
\end{array}
$$

8.2.3 Arithmetic functions

Most instruction sets include instructions to handle the standard arithmetic operations on integers.

A word may be incremented or decremented:

$$
\begin{array}{|l}
\mathit{inc} : \mathit{Word} \rightarrowtail\!\!\!\!\!\to \mathit{Word} \\
\mathit{dec} : \mathit{Word} \rightarrowtail\!\!\!\!\!\to \mathit{Word} \\
\hline
\forall\, w : \mathit{Word} \bullet \\
\quad \mathit{inc}\, w = \mathit{wrd}\, (\#w)\, ((\mathit{succ} \oplus \{\, \mathit{maxval}\, w \mapsto 0 \,\})(\mathit{val}\, w)) \land \\
\quad \mathit{dec}\, w = \mathit{wrd}\, (\#w)\, ((\{\, 0 \mapsto \mathit{maxval}\, w \,\} \cup \mathit{pred})(\mathit{val}\, w))
\end{array}
$$

Incrementing and decrementing a word leaves it unchanged:

$$\vdash\ \mathit{inc} \fatsemi \mathit{dec} = \mathit{dec} \fatsemi \mathit{inc} = \mathrm{id}\, \mathit{Word}$$

Every word may be expressed as a multiply incremented all-zero word:

$$\vdash\ \forall\, w : \mathit{Word} \bullet w = \mathit{inc}^{\mathit{val}\, w}(\mathit{zero}\, w)$$

Addition and subtraction may be defined in terms of repeated incrementing or decrementing.

$$
\begin{array}{|l}
\_ + \_ : \mathit{Word} \times \mathbb{N} \to \mathit{Word} \\
\_ - \_ : \mathit{Word} \times \mathbb{N} \to \mathit{Word} \\
\hline
\forall\, w : \mathit{Word};\ n : \mathbb{N} \bullet \\
\quad w + n = \mathit{inc}^n\, w \land \\
\quad w - n = \mathit{dec}^n\, w
\end{array}
$$

Similarly these can be applied to two words:

$$
\begin{array}{|l}
\_ + \_ : \mathit{Word} \times \mathit{Word} \to \mathit{Word} \\
\_ - \_ : \mathit{Word} \times \mathit{Word} \to \mathit{Word} \\
\hline
\forall\, w_1, w_2 : \mathit{Word} \bullet \\
\quad w_1 + w_2 = w_1 + (\mathit{val}\, w_2) \land \\
\quad w_1 - w_2 = w_1 - (\mathit{val}\, w_2)
\end{array}
$$

Note that addition is not commutative in the general case for words of differing length since the word size of the result is determined by the size of the first work to be added:

$$\vdash \exists w_1, w_2 : \mathit{Word} \bullet w_1 + w_2 \neq w_2 + w_1$$

However if attention is restricted to words of a fixed size the commutativity property does hold:

$$\vdash \forall w_1, w_2 : \mathit{Word} \mid \#w_1 = \#w_2 \bullet w_1 + w_2 = w_2 + w_1$$

Multiplication may be defined inductively in terms of addition.

$$\_ * \_ : \mathit{Word} \times \mathbb{N} \rightarrow \mathit{Word}$$

$$\begin{array}{l} \forall w : \mathit{Word} \bullet \\ \quad w * 0 = \mathit{zero}\, w \wedge \\ \quad (\forall n : \mathbb{N} \bullet w * n = w + w * (n - 1)) \end{array}$$

$$\_ * \_ : \mathit{Word} \times \mathit{Word} \rightarrow \mathit{Word}$$

$$\begin{array}{l} \forall w_1, w_2 : \mathit{Word} \bullet \\ \quad w_1 * w_2 = w_1 * (\mathit{val}\, w_2) \end{array}$$

The same restricted form of commutativity holds for multiplication as for addition:

$$\vdash \forall w_1, w_2 : \mathit{Word} \mid \#w_1 = \#w_2 \bullet w_1 * w_2 = w_2 * w_1$$

The *two's complement* of a word is defined by:

$$\text{-} : \mathit{Word} \rightarrow \mathit{Word}$$

$$\begin{array}{l} \forall w : \mathit{Word} \bullet \\ \quad \text{-}\, w = (\mathit{zero}\, w) \text{ - } w \end{array}$$

Words have absolute values:

$$\mathit{abs} : \mathit{Word} \rightarrow \mathbb{N}$$

$$\begin{array}{l} \forall w : \mathit{Word} \bullet \\ \quad \mathit{MSB}\, w = \mathtt{0} \Rightarrow \mathit{abs}\, w = \mathit{val}\, w \wedge \\ \quad \mathit{MSB}\, w = \mathtt{1} \Rightarrow \mathit{abs}\, w = \mathit{val}(\text{-}\, w) \end{array}$$

Integer division is defined by:

$$\_ \div \_ : \mathit{Word} \times \mathit{Word} \pfun \mathit{Word}$$

$$\begin{array}{l} \forall w_1, w_2 : \mathit{Word} \mid \mathit{val}(w_2) \neq 0 \bullet \\ \quad \mathit{abs}(w_1) < \mathit{abs}(w_2) \Rightarrow w_1 + w_2 = \mathit{zero}\, w_1 \wedge \\ \quad \mathit{abs}(w_1) > \mathit{abs}(w_2) \Rightarrow \\ \qquad (\mathit{abs}(w_1 \div w_2) = 1 + (\mathit{abs}\, w_1 - \mathit{abs}\, w_2) \mathbin{\mathsf{div}} \mathit{abs}\, w_2 \wedge \\ \qquad \mathit{MSB}(w_1 \div w_2) = \mathit{MSB}\, w_1 \oplus \mathit{MSB}\, w_2) \end{array}$$

The *remainder* function is defined in terms of division.

$$\_ \operatorname{rem} \_ : \mathit{Word} \times \mathit{Word} \pfun \mathit{Word}$$

$$\forall w_1, w_2 : \mathit{Word} \mid \mathit{val}(w_2) \neq 0 \bullet w_1 \operatorname{rem} w_2 = w_1 \text{ - } (w_2 * (w_1 \div w_2))$$

8.2.4 Signed integers

The function *val* maps words to unsigned integers. Some instructions use *signed* integers. The function *num* is used to map a word to a signed integer.

$$num : Word \rightarrowtail \mathbb{Z}$$

$$\begin{aligned}
&\forall\, w : Word \bullet \\
&\quad MSB\, w = \mathtt{0} \Rightarrow num\, w = val(w) \land \\
&\quad MSB\, w = \mathtt{1} \Rightarrow num\, w = -(val(\mathtt{-}\, w))
\end{aligned}$$

Signed numbers in a word are rounded off depending on the word length.

$$\_\ \underline{roundoff}\ \_ : \mathbb{Z} \times \mathbb{N}_1 \rightarrow \mathbb{Z}$$

$$\begin{aligned}
&\forall\, n : \mathbb{Z};\ l : \mathbb{N}_1 \bullet \\
&\quad n\ \underline{roundoff}\ l = (n + 2^{l-1}) \bmod 2^l - 2^{l-1}
\end{aligned}$$

Hence the standard signed arithmetic operators can be defined on *Twords*.

$$\begin{aligned}
&_ +_s _, _ -_s _, _ *_s _ : Tword \times Tword \rightarrow Tword \\
&_ \div_s _, _ |_s _ : Tword \times Tword \nrightarrow Tword
\end{aligned}$$

$$\begin{aligned}
&\forall\, w_1, w_2 : Tword \bullet \\
&\quad num(w_1 +_s w_2) = (num(w_1) + num(w_2))\ \underline{roundoff}\ WordLength \land \\
&\quad num(w_1 -_s w_2) = (num(w_1) - num(w_2))\ \underline{roundoff}\ WordLength \land \\
&\quad num(w_1 *_s w_2) = (num(w_1) * num(w_2))\ \underline{roundoff}\ WordLength \land \\
&\quad w_2 \neq \mathbf{0} \Rightarrow \\
&\qquad num(w_1 \div_s w_2) = \\
&\qquad\quad (num(w_1) \operatorname{div} num(w_2))\ \underline{roundoff}\ WordLength \land \\
&\quad w_2 \neq \mathbf{0} \Rightarrow \\
&\qquad num(w_1 |_s w_2) = (num(w_1) \bmod num(w_2))\ \underline{roundoff}\ WordLength
\end{aligned}$$

8.3 Hexadecimal Notation

Hexadecimal notation is traditionally used in computer documentation since it has base a power of 2 and is more concise than binary notation. Hexadecimal digits are drawn from the set of characters *HEXCHAR*.

$$HEXCHAR ::= \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \mathtt{3} \mid \mathtt{4} \mid \mathtt{5} \mid \mathtt{6} \mid \mathtt{7} \mid \mathtt{8} \mid \mathtt{9} \mid \mathtt{A} \mid \mathtt{B} \mid \mathtt{C} \mid \mathtt{D} \mid \mathtt{E} \mid \mathtt{F}$$

Each hexadecimal digit maps onto a unique numerical value:

$$hex : HEXCHAR \rightarrowtail \mathbb{N}$$

$$\begin{aligned}
hex = \{\ &\mathtt{0} \mapsto 0, \mathtt{1} \mapsto 1, \mathtt{2} \mapsto 2, \mathtt{3} \mapsto 3, \mathtt{4} \mapsto 4, \mathtt{5} \mapsto 5, \mathtt{6} \mapsto 6, \mathtt{7} \mapsto 7, \\
&\mathtt{8} \mapsto 8, \mathtt{9} \mapsto 9, \mathtt{A} \mapsto 10, \mathtt{B} \mapsto 11, \mathtt{C} \mapsto 12, \mathtt{D} \mapsto 13, \mathtt{E} \mapsto 14, \mathtt{F} \mapsto 15\ \}
\end{aligned}$$

For convenience the following postfix function is used to distinguish hexadecimal and decimal number strings.

$$
\begin{array}{|l}
\_\,\mathsf{H} : \operatorname{seq} \mathit{HEXCHAR} \to \mathbb{N} \\
\hline
\langle\rangle\,\mathsf{H} = 0 \land \\
(\,\forall\, x : \mathit{HEXCHAR};\ s : \operatorname{seq} \mathit{HEXCHAR} \bullet \\
\qquad (s \frown \langle x \rangle)\,\mathsf{H} = 16 * s\,\mathsf{H} + \mathit{hex}\, x\,)
\end{array}
$$

In Chapter 9, the formal definitions provided in this chapter are used as a basis for defining some of the instructions in the Inmos Transputer instruction set.

Chapter 9

The Transputer Instruction Set

In this chapter, a subset of the Transputer processor instruction set is presented, building on the basic definitions given in Chapter 8. The Transputer is a commercial microprocessor family developed and marketed by Inmos Limited (now SGS-Thomson Microelectronics) [224]. This specification is based on a partial specification of the Transputer instruction set in Z [149]. This itself is based on a Z specification of the Motorola 6800 microprocessor [38, 39] and a description of the Transputer instruction set using a Z-like notation produced by Inmos [224].

9.1 Instructions

The following Transputer instructions are covered in this chapter:

- `pfix` *val* – prefix value.
- `nfix` *val* – negative prefix value.
- `ldc` *con* – load constant.
- `ldl` *adr* – load local from memory address.
- `stl` *adr* – store local to memory address.
- `ldlp` *adr* – load local pointer.
- `adc` *con* – add constant.
- `eqc` *con* – equals constant.
- `cj` *adr* – conditional jump to a memory address.
- `j` *adr* – unconditional jump to a memory address.
- `opr` – arithmetic and other operations.

Note that a sequence of `pfix` and `nfix` instructions is used to increase the size of constant or address available to the next instruction. They are not used at the Assembly Language level.

The `opr` instruction above performs a number of arithmetic and logical operations, and also the following operations:

- `gt` – greater than test.
- `rev` – reverse registers.
- `in` – input a message from a channel.

- `out` – output a message to a channel.
- `seterr` – set error flag true.
- `testerr` – test error flag and clear it.
- `stoperr` – stop if error flag is set.
- `stopp` – stop process.

9.2 Machine State

The state of the parts of the Transputer modelled here consists of several components:

1. Registers
2. Memory
3. System clock
4. Error flag
5. Status

These components are required for almost all instructions. Other state components of a more specialized nature are introduced in the sections in which they are used.

9.2.1 Registers

The Transputer has six principle working *registers*. Other special purpose registers are introduced when required. Three of the working registers form an *evaluation stack* which is used as a workspace by most instructions.

EVALUATION_STACK

$$Areg : Tword$$
$$Breg : Tword$$
$$Creg : Tword$$

The other three working registers are the *instruction pointer*, which points to the next instruction to be executed, the *operand register*, which is described in a later section, and the *workspace pointer*. The process running on the Transputer has a workspace in memory and the workspace pointer references the workspace of the executing process.

REGISTERS

$$EVALUATION\_STACK$$
$$Oreg : Tword$$
$$Wptr : Tword$$
$$Iptr : Tword$$

As with a conventional stack, the basic operations on the evaluation stack are pushes and pops.

PUSH

$$\Delta EVALUATION\_STACK$$

$$Breg' = Areg$$
$$Creg' = Breg$$

POP

$$\Delta EVALUATION\_STACK$$

$$Areg' = Breg$$
$$Breg' = Creg$$

The *Areg* is at the top of the stack and the result is normally returned in this register as $Areg'$.

9.2.2 Memory

The *memory* of the Transputer may be regarded as a function from addresses to bytes. An address is a *Tword*. Each address is composed of a word address component and a byte selector component. The byte selector selects a byte within a machine word. To allow each byte within a word to be addressed the number of bits within a *Tword* allocated to the byte selector is the smallest power of 2 which is at least as large as *BytesPerWord*. The byte selector occupies the least significant end of the word.

The number of bits in a word allocated to the byte selector is determined as follows:

$$ByteSelectLength : \mathbb{N}$$

$$2^{ByteSelectLength-1} < BytesPerWord \leq 2^{ByteSelectLength}$$

Using this the functions *ByteSelector* and *WordAddress* can be defined:

$$ByteSelector : Tword \rightarrow Word$$

$$\forall w : Tword \bullet$$
$$\quad ByteSelector\, w = (0 \,..\, ByteSelectLength - 1) \lhd w$$

$$WordAddress : Tword \rightarrow Word$$

$$\forall w : Tword \bullet$$
$$\quad WordAddress\, w = succ^{ByteSelectLength}\, {}_{9}^{9}\, w$$

Using these definitions *Address* may be defined:

$$Address == \{ w : Word \mid \#w = WordLength \wedge val(ByteSelector\, w) < BytesPerWord \}$$

A subset of these addresses are on word boundaries:

$$TwordAddress == \{ a : Address \mid val(ByteSelector\, a) = 0 \}$$

Each *Address* is thus composed of a *WordAddress* part and a *ByteSelector* part:

$$\vdash \forall a : Address \bullet a = (WordAddress\, a) \frown (ByteSelector\, a)$$

A consequence of this definition is that if *BytesPerWord* is not a power of 2 then there are *Tword* values which are not *Address*es. In this case addresses are not continuous across word boundaries. Two functions, *Index* and *ByteIndex* are used to increment through addresses taking account of this feature. *Index* takes a base address and calculates a new address at a given word offset:

$$Index : TwordAddress \rightarrow \mathbb{Z} \rightarrow Address$$

$$\begin{aligned}&\forall z : \mathbb{Z};\ a : TwordAddress \bullet \\ &\quad WordAddress(Index\, a\, z) = (WordAddress\, a) + z \land \\ &\quad val(ByteSelector(Index\, a\, z)) = 0\end{aligned}$$

It is convenient to define a similar function with the second parameter a word instead of an integer:

$$Index : TwordAddress \rightarrow Word \rightarrow Address$$

$$\begin{aligned}&\forall w : Word;\ a : TwordAddress \bullet \\ &\quad Index\, a\, w = Index\, a\, (num\, w)\end{aligned}$$

ByteIndex takes a base address and calculates a new address at a given byte offset. In calculating a byte offset account must be taken of possible 'holes' as described above. *ByteInc* describes the byte at an offset of $+1$ byte:

$$ByteInc : Address \rightarrow Address$$

$$\begin{aligned}&\forall a : Address \bullet \\ &\quad val(ByteSelector\, a) < BytesPerWord - 1 \Rightarrow ByteInc\, a = a + 1 \land \\ &\quad val(ByteSelector\, a) = BytesPerWord - 1 \Rightarrow \\ &\qquad (\, WordAddress(ByteInc\, a) = (WordAddress\, a) + 1 \land \\ &\qquad\ val(ByteSelector(ByteInc\, a)) = 0\,)\end{aligned}$$

$$ByteIndex : Address \rightarrow \mathbb{Z} \rightarrow Address$$

$$\begin{aligned}&\forall a : Address;\ z : \mathbb{Z} \bullet \\ &\quad ByteIndex\, a\, z = ByteInc^z\, a\end{aligned}$$

Again the same function may be defined with the second parameter a word instead of an integer:

$$ByteIndex : Address \rightarrow Word \rightarrow Address$$

$$\begin{aligned}&\forall a : Address;\ w : Word \bullet \\ &\quad ByteIndex\, a\, w = ByteIndex\, a\, (num\, w)\end{aligned}$$

Most instructions consider memory to consist of words and so it is convenient to include both a byte representation and a word representation of memory in the formal definition. Clearly these two definitions must be linked. The link is established by noting that a *Tword* is a sequence of bytes. Then $\frown/(Bytes\, a)$ can be seen to be equivalent to *WordMem a*.

MEMORY_REP

$$
\begin{array}{l}
ByteMem : Address \rightarrow Byte \\
WordMem : TwordAddress \rightarrow Tword \\
Bytes : TwordAddress \rightarrow \mathrm{seq}\, Byte \\
\hline
\forall\, a : TwordAddress \bullet \\
\quad Bytes\, a = \{\, n : \mathbb{N} \mid n < BytesPerWord \bullet \\
\qquad\qquad n + 1 \mapsto ByteMem\,(ByteIndex\, a\, n) \,\} \land \\
\quad WordMem\, a = \frown / (Bytes\, a)
\end{array}
$$

An operation changing memory need only update *WordMem* or *ByteMem* as appropriate and the other representation of memory gets updated automatically through this schema.

Occasionally it is convenient to refer to word memory relative to the workspace pointer *Wptr*. The function *WorkSpace* allows this:

MEMORY_REP

$$
\begin{array}{l}
MEMORY\_REP \\
WorkSpace : Tword \pfun Tword \\
Wptr : Tword \\
\hline
val(ByteSelector\, Wptr) = 0 \\
\mathrm{dom}\, WorkSpace = \{\, a : Tword \mid val(a) < 2^{WordLength - ByteSelectLength} \,\} \\
(\, \forall\, a : \mathrm{dom}\, WorkSpace \bullet \\
\qquad WorkSpace\, a = WordMem(Index\, Wptr\, a)\,)
\end{array}
$$

Note that *Wptr* must always point to a word boundary.

A final point to note about memory is that the address space will be divided into areas such as ROM, on-chip and off-chip RAM, and memory mapped peripherals (e.g., I/O channels).

MEMORY_MAP

$$
\begin{array}{l}
RAM, ROM : \mathbb{F}_1\, Address \\
onchipRAM, offchipRAM : \mathbb{F}\, Address \\
\hline
RAM \cap ROM = \emptyset \\
onchipRAM \cap offchipRAM = \emptyset \\
onchipRAM \cup offchipRAM = RAM
\end{array}
$$

The ROM area will normally hold the program code and the RAM area will normally hold the program variables.

Combining the structure and the representation of memory together gives:

$$MEMORY \mathrel{\widehat=} MEMORY\_REP \land MEMORY\_MAP$$

When memory is updated, the memory map (i.e., the RAM and ROM addresses) are unaffected. In addition, the contents of ROM is always left unchanged. Some instructions write to memory; these update $ByteMem''$ below which only updates areas of memory address space which contain RAM. Areas outside ROM and RAM contain undefined values.

$$
\begin{array}{|l}
\hline
\Delta MEMORY \\
\hline
MEMORY \\
MEMORY' \\
\Xi MEMORY\_MAP \\
MEMORY\_REP'' \\
\hline
offchipRAM' = offchipRAM \\
onchipRAM' = onchipRAM \\
ROM \lhd ByteMem' = ROM \lhd ByteMem \\
RAM \lhd ByteMem' = RAM \lhd ByteMem'' \\
Wptr' = Wptr'' \\
\hline
\end{array}
$$

Sometimes the contents of RAM is also left unchanged.

$$\Xi MEMORY \mathrel{\widehat=} [\Delta MEMORY \mid RAM \lhd ByteMem' = RAM \lhd ByteMem]$$

9.2.3 System clock

The machine contains a *clock* which controls the timing of the microprocessor. This consists of a sequence of pulses and may be modelled by the number of clock pulses which have occurred since the machine was powered up:

$$
\begin{array}{|l}
\hline
CLOCK \\
\hline
Clk : \mathbb{N} \\
\hline
\end{array}
$$

When an instruction is executed it takes a certain number of clock cycles:

$$
\begin{array}{|l}
\hline
\Delta CLOCK \\
\hline
CLOCK \\
CLOCK' \\
Cycles : \mathbb{N} \\
\hline
Clk' = Clk + Cycles \\
\hline
\end{array}
$$

Inclusion of a clock is not strictly necessary in the specification but it allows reasoning about the timing of combinations of instructions, should this be desirable in the future. The clock cycles given for subsequent instructions are taken from [224].

9.2.4 Errors

The Transputer provides a single *error flag* which may be set by a number of instructions.

$$
\begin{array}{|l}
\hline
ERROR \\
\hline
ErrorFlag : Bit \\
\hline
\end{array}
$$

The error flag may take the values *Clear* or *Set*:

$$Clear == \mathtt{0}$$
$$Set == \mathtt{1}$$

9.2.5 Status

The machine may be *running* or *stopped*.

$$Mode ::= running \mid stopped$$

$$\begin{array}{|l} \textit{STATUS} \\ \hline Status : Mode \\ \hline \end{array}$$

For an instruction to be executed, the machine must be running:

$$\begin{array}{|l} \Delta \textit{STATUS} \\ \hline \textit{STATUS} \\ \textit{STATUS}' \\ \hline Status = running \\ \hline \end{array}$$

After an instruction, the machine may be running or stopped depending on the instruction itself. Most instructions leave the processor running:

$$\begin{array}{|l} \Xi \textit{STATUS} \\ \hline \Delta \textit{STATUS} \\ \hline Status' = Status \\ \hline \end{array}$$

9.2.6 Combined state

Combining the separate state component schema gives:

$$TRANS \mathrel{\widehat=} REGISTERS \land MEMORY \land CLOCK \land ERROR \land STATUS$$

provides a simplified description of the Transputer.

The change of state is defined as:

$$\Delta TRANS \mathrel{\widehat=} TRANS \land TRANS' \land \Delta REGISTERS \land \Delta MEMORY \land \Delta CLOCK \land \Delta STATUS$$

Many instructions leave the memory and error flag and status unaffected:

$$\Xi TRANS \mathrel{\widehat=} \Delta TRANS \land \Xi MEMORY \land \Xi ERROR \land \Xi STATUS$$

9.3 Instructions

The following basic instructions are included in the Transputer:

$$Instruction ::= \texttt{pfix} \mid \texttt{nfix} \mid \texttt{ldc} \mid \texttt{adc} \mid \texttt{ldl} \mid \texttt{stl} \mid \texttt{ldlp} \mid \texttt{j} \mid \texttt{cj} \mid \texttt{eqc} \mid \texttt{opr}$$

The `opr` instruction allows further ALU and other operations which can be expanded as required. For the Transputer instruction set defined here, the following are specified:

$$
\begin{array}{l}
\mathit{Operation} ::= \\
\quad \mathtt{add} \mid \mathtt{sub} \mid \mathtt{mul} \mid \mathtt{div} \mid \mathtt{rem} \mid \mathtt{sum} \mid \mathtt{diff} \mid \mathtt{prod} \mid \mathtt{gt} \mid \mathtt{rev} \mid \\
\quad \mathtt{and} \mid \mathtt{or} \mid \mathtt{xor} \mid \mathtt{not} \mid \mathtt{shl} \mid \mathtt{shr} \mid \mathtt{in} \mid \mathtt{out} \mid \\
\quad \mathtt{seterr} \mid \mathtt{testerr} \mid \mathtt{stoperr} \mid \mathtt{stopp}
\end{array}
$$

Each instruction and operation is allocated some unique op-code, here defined using hexadecimal notation:

$$
\begin{array}{|l}
\mathit{InstructionOpCode} : \mathit{Instruction} \rightarrowtail \mathbb{N} \\
\mathit{OperationOpCode} : \mathit{Operation} \rightarrowtail \mathbb{N} \\
\hline
\mathit{InstructionOpCode} = \\
\quad \{ \mathtt{pfix} \mapsto \langle \mathtt{2} \rangle \, \mathtt{H}, \mathtt{nfix} \mapsto \langle \mathtt{6} \rangle \, \mathtt{H}, \mathtt{ldc} \mapsto \langle \mathtt{4} \rangle \, \mathtt{H}, \mathtt{adc} \mapsto \langle \mathtt{8} \rangle \, \mathtt{H}, \\
\quad \mathtt{ldl} \mapsto \langle \mathtt{7} \rangle \, \mathtt{H}, \mathtt{stl} \mapsto \langle \mathtt{D} \rangle \, \mathtt{H}, \mathtt{ldlp} \mapsto \langle \mathtt{1} \rangle \, \mathtt{H}, \\
\quad \mathtt{j} \mapsto \langle \mathtt{0} \rangle \, \mathtt{H}, \mathtt{cj} \mapsto \langle \mathtt{A} \rangle \, \mathtt{H}, \mathtt{eqc} \mapsto \langle \mathtt{C} \rangle \, \mathtt{H}, \mathtt{opr} \mapsto \langle \mathtt{F} \rangle \, \mathtt{H} \, \} \\
\mathit{OperationOpCode} = \\
\quad \{ \mathtt{add} \mapsto \langle \mathtt{0}, \mathtt{5} \rangle \, \mathtt{H}, \mathtt{sub} \mapsto \langle \mathtt{0}, \mathtt{C} \rangle \, \mathtt{H}, \mathtt{mul} \mapsto \langle \mathtt{5}, \mathtt{3} \rangle \, \mathtt{H}, \mathtt{div} \mapsto \langle \mathtt{2}, \mathtt{C} \rangle \, \mathtt{H}, \\
\quad \mathtt{rem} \mapsto \langle \mathtt{1}, \mathtt{F} \rangle \, \mathtt{H}, \mathtt{sum} \mapsto \langle \mathtt{5}, \mathtt{2} \rangle \, \mathtt{H}, \mathtt{diff} \mapsto \langle \mathtt{0}, \mathtt{4} \rangle \, \mathtt{H}, \mathtt{prod} \mapsto \langle \mathtt{0}, \mathtt{8} \rangle \, \mathtt{H}, \\
\quad \mathtt{gt} \mapsto \langle \mathtt{0}, \mathtt{9} \rangle \, \mathtt{H}, \mathtt{rev} \mapsto \langle \mathtt{0}, \mathtt{0} \rangle \, \mathtt{H}, \mathtt{and} \mapsto \langle \mathtt{4}, \mathtt{6} \rangle \, \mathtt{H}, \mathtt{or} \mapsto \langle \mathtt{4}, \mathtt{B} \rangle \, \mathtt{H}, \\
\quad \mathtt{xor} \mapsto \langle \mathtt{3}, \mathtt{3} \rangle \, \mathtt{H}, \mathtt{not} \mapsto \langle \mathtt{3}, \mathtt{2} \rangle \, \mathtt{H}, \mathtt{shl} \mapsto \langle \mathtt{4}, \mathtt{1} \rangle \, \mathtt{H}, \mathtt{shr} \mapsto \langle \mathtt{4}, \mathtt{0} \rangle \, \mathtt{H}, \\
\quad \mathtt{seterr} \mapsto \langle \mathtt{1}, \mathtt{0} \rangle \, \mathtt{H}, \mathtt{testerr} \mapsto \langle \mathtt{2}, \mathtt{9} \rangle \, \mathtt{H}, \mathtt{stoperr} \mapsto \langle \mathtt{5}, \mathtt{5} \rangle \, \mathtt{H}, \\
\quad \mathtt{in} \mapsto \langle \mathtt{0}, \mathtt{7} \rangle \, \mathtt{H}, \mathtt{out} \mapsto \langle \mathtt{0}, \mathtt{B} \rangle \, \mathtt{H}, \mathtt{stopp} \mapsto \langle \mathtt{1}, \mathtt{5} \rangle \, \mathtt{H} \, \}
\end{array}
$$

All Transputer instructions are single bytes. The most significant 4 bits of the byte form an *op-code* and the least significant 4 bits form an *operand*.

$$
\begin{array}{|l}
\mathit{OpCode}, \mathit{Operand} : \mathit{Byte} \rightarrow \mathbb{N} \\
\hline
\forall \, b : \mathit{Byte} \bullet \\
\quad \mathit{OpCode}\, b = \mathit{val}(\mathit{succ}^4 \, ^{\S} \, b) \land \\
\quad \mathit{Operand}\, b = \mathit{val}(\, (0 \,..\, 3) \lhd b \,)
\end{array}
$$

This design of instruction leads to only 16 *instructions* being available. Other functions are invoked by means of the `OPR` instruction and are thus *operations* rather than instructions.

The operand is not operated on directly but instead it is added into the Operand Register (*Oreg*). The instruction then operates on the contents of *Oreg*:

$$
\begin{array}{|l}
\textit{DECODE} \\
\hline
\mathit{REGISTERS} \\
\mathit{MEMORY} \\
\mathit{Oreg}^o : \mathit{Tword} \\
\hline
\mathit{Oreg}^o = \mathit{Oreg} + \mathit{Twrd}(\mathit{Operand}(\mathit{ByteMem}\, \mathit{Iptr}))
\end{array}
$$

It is now possible to define partial schema definitions for instructions and operations.

$\Phi INSTRUCTION$

$\Delta TRANS$
$Instr : Instruction$
$DECODE$

$OpCode(ByteMem\ Iptr) = InstructionOpCode\ Instr$

$\Phi OPERATION$

$\Phi INSTRUCTION$
$Opr : Operation$

$Instr = \texttt{opr}$
$Oreg^o = Twrd(OperationOpCode\ Opr)$

The instruction pointer (*Iptr*) points to the current instruction to be executed. Most instructions simply increment *Iptr* on completion. As was noted previously addresses are not necessarily continuous and so the function *NextInst* increments *Iptr* to the next value which is a valid address.

$NextInst : Address \rightarrow Address$

$\forall a : Address \bullet$
$\quad NextInst\ a = ByteIndex\ a\ 1$

Simple instructions are classified as those which increment *Iptr* and leave *Wptr* unchanged and set $Oreg'$ to 0:

$\Phi SIMPLE$

$\Delta REGISTERS$

$Oreg' = \mathbf{0}$
$Wptr' = Wptr$
$Iptr' = NextInst\ Iptr$

$\Phi SIMPLE\_INSTRUCTION \mathrel{\widehat=} \Phi SIMPLE \land \Phi INSTRUCTION$

$\Phi SIMPLE\_OPERATION \mathrel{\widehat=} \Phi SIMPLE \land \Phi OPERATION$

9.3.1 Instructions using the evaluation stack

Simple manipulation of the evaluation stack using *PUSH* and *POP* has been covered in an earlier section. Depending on the number of operands an instruction has it may perform a sequence of *POP*s and *PUSH*es. Some of these sequences are common enough to define general partially specified schemas for inclusion in subsequent definitions.

Single operand instructions

Some instructions take one parameter from the evaluation stack, perform some operation on the parameter and return a result to the evaluation stack. These instructions are characterized by $\Phi SINGLE$.

$$\Phi SINGLE \mathrel{\widehat=} POP \fatsemi PUSH$$

Expanding this schema shows how the registers comprising the evaluation stack are affected:

$\Phi SINGLE$

$$Areg, Breg, Creg : Tword$$
$$Areg', Breg', Creg' : Tword$$

$$Breg' = Breg$$
$$Creg' = Creg$$

Double operand instructions

Double operand instructions take two parameters from the evaluation stack, perform some operation on them and return a result to the evaluation stack.

$$\Phi DOUBLE \mathrel{\widehat=} POP \fatsemi POP \fatsemi PUSH$$

This expands to:

$\Phi DOUBLE$

$$Areg, Breg, Creg : Tword$$
$$Areg', Breg', Creg' : Tword$$

$$Breg' = Creg$$

$Creg'$ is not constrained by the predicate and is therefore undefined.

9.3.2 Operation creation instructions

The `OPR` instruction uses the contents of *Oreg* to determine which operation to execute. Two instructions are provided to manipulate *Oreg* and set up these operations.

`PFIX`

$$\Phi INSTRUCTION$$
$$\Xi EVALUATION\_STACK$$
$$\Xi TRANS$$

$$Oreg' = Oreg^o \ll 4$$
$$Wptr' = Wptr$$
$$Iptr' = Iptr + 1$$
$$Cycles = 1$$
$$Instr = \texttt{pfix}$$

NFIX

$\Phi INSTRUCTION$
$\Xi EVALUATION\_STACK$
$\Xi TRANS$

$Oreg' = (\sim Oreg^o) \ll 4$
$Wptr' = Wptr$
$Iptr' = Iptr + 1$
$Cycles = 1$
$Instr = \texttt{nfix}$

9.3.3 Memory access instructions

This section describes those instructions which load values from memory to the evaluation stack or store values from the evaluation stack into memory. The timings for instructions which read from or write to memory are applicable to on-chip RAM only.

`ldc` *con* instruction

`LDC` loads a constant into the evaluation stack.

LDC

$\Phi SIMPLE\_INSTRUCTION$
$PUSH$
$\Xi TRANS$

$Areg' = Oreg^o$
$Cycles = 1$
$Instr = \texttt{ldc}$

`ldl` *adr* instruction

`LDL` loads a local variable into the evaluation stack. A local variable is addressed by its offset from *Wptr*.

LDL

$\Phi SIMPLE\_INSTRUCTION$
$PUSH$
$\Xi TRANS$

$Areg' = WorkSpace\, Oreg^o$
$(Wptr + Oreg^o) \in onchipRAM \Rightarrow Cycles = 2$
$Instr = \texttt{ldl}$

`stl` *adr* instruction

`STL` stores the value at the top of the evaluation stack into a local variable.

STL

$\Phi SIMPLE\_INSTRUCTION$
POP
$\Xi ERROR$
$\Xi STATUS$

$Oreg^o \in \mathrm{dom}\, WorkSpace \Rightarrow$
$\quad WorkSpace'' = WorkSpace \oplus \{Oreg^o \mapsto Areg\}$

$(Wptr + Oreg^o) \in onchipRAM \Rightarrow Cycles = 1$
$Instr = \texttt{stl}$

`ldlp` *adr* instruction

`LDLP` loads a local pointer into the evaluation stack.

LDLP

$\Phi SIMPLE\_INSTRUCTION$
$PUSH$
$\Xi TRANS$

$Areg' = Index\ Wptr\ Oreg^o$
$Cycles = 1$
$Instr = \texttt{ldlp}$

9.3.4 Integer arithmetic instructions

Arithmetic operations may lead to overflows. Many of the instructions in this section check for overflow and set the *ErrorFlag* if required. Instructions make use of the dyadic arithmetic operators defined earlier:

$$\mathrm{dyad}[X] == X \times X \nrightarrow X$$

A one-to-one mapping from the arithmetic operators on words is made to the corresponding operators on integers:

$ArithOp : \mathrm{dyad}[Tword] \rightarrowtail \mathrm{dyad}[\mathbb{Z}]$
$WordOp : \mathbb{P}\, \mathrm{dyad}[Tword]$

$ArithOp = \{ (\_ +_s \_) \mapsto (\_ + \_), (\_ -_s \_) \mapsto (\_ - \_), (\_ *_s \_) \mapsto (\_ * \_) \}$

$WordOp = \mathrm{dom}\, ArithOp$

The set of arithmetic word operators is also useful.

InRange checks for arithmetic overflow:

$InRange : \mathbb{P}(Tword \times WordOp \times Tword)$

$\forall w_1, w_2 : Tword;\ op : WordOp \bullet$
$\quad (w_1, op, w_2) \in InRange \Leftrightarrow$
$\quad\quad (ArithOp\ op)(val\ w_1, val\ w_2) = val(\ op(w_1, w_2)\ )$

This is used to set the error flag appropriately after arithmetic instructions.

adc *con* instruction

ADC adds a constant to the value at the top of the evaluation stack.

ADC

$\Phi SIMPLE\_INSTRUCTION$
$\Phi SINGLE$
$\Xi MEMORY$
$\Xi STATUS$

$Areg' = Areg +_s Oreg^o$
$Cycles = 1$
$Instr = \mathsf{adc}$
$(Areg, (\_ +_s \_), Oreg^o) \in InRange \Rightarrow ErrorFlag' = ErrorFlag$
$(Areg, (\_ +_s \_), Oreg^o) \notin InRange \Rightarrow ErrorFlag' = Set$

add operation

ADD sums *Areg* and *Breg* writing the result to *Areg*.

ADD

$\Phi SIMPLE\_OPERATION$
$\Phi DOUBLE$
$\Xi MEMORY$
$\Xi STATUS$

$Areg' = Breg +_s Areg$
$Cycles = 1$
$Opr = \mathsf{add}$
$(Areg, (\_ +_s \_), Breg) \in InRange \Rightarrow ErrorFlag' = ErrorFlag$
$(Areg, (\_ +_s \_), Breg) \notin InRange \Rightarrow ErrorFlag' = Set$

sub operation

SUB subtracts *Areg* from *Breg* writing the result to *Areg*.

SUB

$\Phi SIMPLE\_OPERATION$
$\Phi DOUBLE$
$\Xi MEMORY$
$\Xi STATUS$

$Areg' = Breg -_s Areg$
$Cycles = 1$
$Opr = \mathsf{sub}$
$(Breg, (\_ -_s \_), Areg) \in InRange \Rightarrow ErrorFlag' = ErrorFlag$
$(Breg, (\_ -_s \_), Areg) \notin InRange \Rightarrow ErrorFlag' = Set$

`mul` operation

`MUL` multiplies *Areg* and *Breg* writing the result to *Areg*.

MUL

$$\Phi SIMPLE\_OPERATION$$
$$\Phi DOUBLE$$
$$\Xi MEMORY$$
$$\Xi STATUS$$

$$Areg' = Breg *_s Areg$$
$$Cycles = WordLength + 6$$
$$Opr = \texttt{mul}$$
$$(Breg, (\_ *_s \_), Areg) \in InRange \Rightarrow ErrorFlag' = ErrorFlag$$
$$(Breg, (\_ *_s \_), Areg) \notin InRange \Rightarrow ErrorFlag' = Set$$

`div` operation

`DIV` performs an integer division of *Breg* by *Areg*. The case $Areg = \mathbf{0}$ leads to an error. An error also occurs if $Areg = \textbf{-1}$ and $Breg = MostNeg$ since $MostNeg \div_s \textbf{-1} = MostPos +_s \mathbf{1}$ which is not representable in a *Tword*.

DIV

$$\Phi SIMPLE\_OPERATION$$
$$\Phi DOUBLE$$
$$\Xi MEMORY$$
$$\Xi STATUS$$

$$(Areg \neq \mathbf{0}) \land (Areg \neq \textbf{-1} \lor Breg \neq MostNeg) \Rightarrow$$
$$\quad (\, Areg' = Breg \div_s Areg \;\land\; ErrorFlag' = ErrorFlag \,)$$

$$(Areg = \mathbf{0}) \lor (Areg = \textbf{-1} \land Breg = MostNeg) \Rightarrow$$
$$\quad ErrorFlag' = Set$$

$$Cycles = WordLength + 7$$
$$Opr = \texttt{div}$$

`rem` operation

`REM` gives the remainder when *Breg* is divided by *Areg*. The error cases are the same as for `DIV`.

REM

$\Phi SIMPLE\_OPERATION$
$\Phi DOUBLE$
$\Xi MEMORY$
$\Xi STATUS$

$(Areg \neq \mathbf{0}) \land (Areg \neq \textbf{-1} \lor Breg \neq MostNeg) \Rightarrow$
$\quad (\, Areg' = Breg \mid_s Areg \;\land\; ErrorFlag' = ErrorFlag \,)$

$(Areg = \mathbf{0}) \lor (Areg = \textbf{-1} \land Breg = MostNeg) \Rightarrow$
$\quad ErrorFlag' = Set$

$Cycles = WordLength + 5$
$Opr =$ `rem`

`SUM`, `DIFF` and `PROD` do the same as `ADD`, `SUB` and `MUL` respectively except that they never set *ErrorFlag*. `PROD` can also be faster than `MUL` in certain circumstances.

`sum` operation

SUM

$\Phi SIMPLE\_OPERATION$
$\Phi DOUBLE$
$\Xi TRANS$

$Areg' = Breg +_s Areg$
$Cycles = 1$
$Opr =$ `sum`

`diff` operation

DIFF

$\Phi SIMPLE\_OPERATION$
$\Phi DOUBLE$
$\Xi TRANS$

$Areg' = Breg -_s Areg$
$Cycles = 1$
$Opr =$ `diff`

`prod` operation

PROD

$\Phi SIMPLE\_OPERATION$
$\Phi DOUBLE$
$\Xi TRANS$

$Areg' = Breg *_s Areg$
$Cycles = (HighestSetBit\, Areg) + 4$
$Opr =$ `prod`

9.3.5 Bitwise logical instructions

Four bitwise logical operations are provided which invoke the usual bitwise logical functions of AND, OR and NOT.

`and` operation

AND

$$\Phi SIMPLE\_OPERATION$$
$$\Phi DOUBLE$$
$$\Xi TRANS$$

$$Areg' = Breg \bullet Areg$$
$$Cycles = 1$$
$$Opr = \texttt{and}$$

`or` operation

OR

$$\Phi SIMPLE\_OPERATION$$
$$\Phi DOUBLE$$
$$\Xi TRANS$$

$$Areg' = Breg \ ^{+} \ Areg$$
$$Cycles = 1$$
$$Opr = \texttt{or}$$

`xor` operation

XOR

$$\Phi SIMPLE\_OPERATION$$
$$\Phi DOUBLE$$
$$\Xi TRANS$$

$$Areg' = Breg \oplus Areg$$
$$Cycles = 1$$
$$Opr = \texttt{xor}$$

`not` operation

NOT

$$\Phi SIMPLE\_OPERATION$$
$$\Phi SINGLE$$
$$\Xi TRANS$$

$$Areg' = \sim Areg$$
$$Cycles = 1$$
$$Opr = \texttt{not}$$

9.3.6 Shift instructions

Two shift instructions are provided. Both shift zeroes into the vacated positions. Other kinds of shift must be manufactured from a combination of shift and logical instructions.

`shl` operation

SHL

$\Phi SIMPLE\_OPERATION$
$\Phi DOUBLE$
$\Xi TRANS$

$Areg' = Breg \ll (val\, Areg)$
$Cycles = (val\, Areg) + 2$
$Opr = \texttt{shl}$

`shr` operation

SHR

$\Phi SIMPLE\_OPERATION$
$\Phi DOUBLE$
$\Xi TRANS$

$Areg' = Breg \gg (val\, Areg)$
$Cycles = (val\, Areg) + 2$
$Opr = \texttt{shr}$

9.3.7 Simple test instructions

This section describes a variety of instructions which return Boolean values according to the result obtained from their operations.

`eqc` *con* instruction

`EQC` tests *Areg* against the operand in $Oreg^o$, returning the result in *Areg*.

EQC

$\Phi SIMPLE\_INSTRUCTION$
$\Phi SINGLE$
$\Xi TRANS$

$Areg = Oreg^o \Rightarrow Areg' = True$
$Areg \neq Oreg^o \Rightarrow Areg' = False$
$Cycles = 2$
$Instr = \texttt{eqc}$

gt operation

GT tests *Breg* against *Areg* returning the result in *Areg*. The operands are treated as signed integers.

GT

$\Phi SIMPLE\_OPERATION$
$\Phi DOUBLE$
$\Xi TRANS$

$num\,Breg > num\,Areg \Rightarrow Areg' = True$
$num\,Breg \leq num\,Areg \Rightarrow Areg' = False$
$Cycles = 1$
$Opr = \texttt{gt}$

9.3.8 Error instructions

seterr operation

SETERR sets the *ErrorFlag*.

SETERR

$\Phi OPERATION$
$\Xi EVALUATION\_STACK$
$\Xi MEMORY$
$\Xi STATUS$

$ErrorFlag' = Set$
$Cycles = 1$
$Opr = \texttt{seterr}$

testerr operation

TESTERR tests the *ErrorFlag* returning the result on the evaluation stack.

TESTERR

$\Phi SIMPLE\_OPERATION$
$PUSH$
$\Xi TRANS$
$\Xi STATUS$

$ErrorFlag = Set \Rightarrow (ErrorFlag' = Clear \land Areg' = False)$
$ErrorFlag = Clear \Rightarrow (ErrorFlag' = Set \land Areg' = True)$
$Cycles = 3$
$Opr = \texttt{testerr}$

stoperr operation

STOPERR stops the machine if the *ErrorFlag* is set.

STOPERR

$$\Phi OPERATION$$
$$\Xi EVALUATION\_STACK$$
$$\Xi MEMORY$$
$$\Xi ERROR$$

$$ErrorFlag = Set \Rightarrow Status' = stopped$$
$$ErrorFlag = Clear \Rightarrow Status' = Status$$
$$Cycles = 2$$
$$Opr = \texttt{stoperr}$$

9.3.9 Branch instructions

This section describes the branch instructions. Since all the branch instructions potentially cause a discontinuity in the value of *Iptr* the schema $\Phi SIMPLE$ is no longer appropriate. Instead the schema $\Phi BRANCH$ is defined.

$\Phi BRANCH$

$$\Phi INSTRUCTION$$
$$\Xi TRANS$$
$$JumpAddr : Address$$

$$Wptr' = Wptr$$
$$Oreg' = \mathbf{0}$$
$$JumpAddr = ByteIndex(NextInst\ Iptr)\ Oreg^{o}$$

`j` *adr* instruction

`J` jumps to an address specified as a byte offset from the instruction following the `J` instruction.

J

$$\Phi BRANCH$$

$$\Xi EVALUATION\_STACK$$
$$Iptr' = JumpAddr$$
$$Cycles = 3$$
$$Instr = \texttt{j}$$

`cj` *adr* instruction

`CJ` causes a jump to a byte offset from the instruction following the `CJ` instruction if *Areg* is 0.

CJ

$$\Phi BRANCH$$

$$Areg = \mathbf{0} \Rightarrow (\Xi EVALUATION\_STACK \land Iptr' = JumpAddr \land Cycles = 4)$$

$$Areg \neq \mathbf{0} \Rightarrow (POP \land Iptr' = NextInst\, Iptr \land Cycles = 2)$$

$$Instr = \texttt{cj}$$

9.3.10 Communication instructions

Communication at the instruction set level is carried out using high level links. The method of communication is based on CSP [215] and avoids the need for complicated communications protocols common in other microprocessors. Four hard links are provided on the Transputer allowing external communication. The links provide two channels, one in each direction.

$$InChannels, OutChannels : \mathbb{F}\, TwordAddress$$

$$\#InChannels = 4$$
$$\#OutChannels = 4$$

`in` instruction

`IN` reads a sequence of bytes into memory starting at the address specified by *Creg*. *Areg* specifies the length of the sequence. *Breg* defines the channel address being used.

IN

$$\Phi SIMPLE\_OPERATION$$
$$\Xi ERROR$$
$$\Xi STATUS$$
$$Input? : \mathrm{seq}\, Byte$$
$$chan! : TwordAddress$$

$$\#Input? = val\, Areg$$
$$Breg \in InChannels$$
$$chan! = Breg$$
$$ByteMem'' = ByteMem \oplus \{ i : \mathbb{N} \mid i \in 1 \,..\, val\, Areg \bullet (ByteIndex\, Creg\, (i-1)) \mapsto (Input?\, i) \}$$
$$Cycles \geq 2 * ((\#Input? + WordLength - 1)\ \mathsf{div}\ WordLength) + 18$$
$$Opr = \texttt{in}$$

Note that *Areg*′, *Breg*′ and *Creg*′ are undefined.

`out` instruction

`OUT` writes a sequence of bytes from memory starting at the address specified by *Creg*. *Areg* specifies the length of the sequence. *Breg* defines the channel address being used.

OUT

$\Phi SIMPLE\_OPERATION$
$\Xi TRANS$
$Output! : \mathrm{seq}\, Byte$
$chan! : TwordAddress$

$Breg \in OutChannels$
$chan! = Breg$
$Output! =$
$\quad \{ i : \mathbb{N} \mid i \in 1 \,..\, valAreg \bullet i \mapsto ByteMem(ByteIndex\, Creg\, (i-1)) \}$
$Cycles \geq 2 * ((\#Output! + WordLength - 1)\ \mathsf{div}\ WordLength) + 20$
$Opr = \texttt{out}$

9.3.11 Miscellaneous instructions

`rev` operation

REV reverses the registers *Areg* and *Breg*.

REV

$\Phi SIMPLE\_OPERATION$
$\Xi TRANS$

$Areg' = Breg$
$Breg' = Areg$
$Creg' = Creg'$
$Cycles = 1$
$Opr = \texttt{rev}$

`stopp` operation

STOPP stops the processor. *Iptr* is saved in the work space so it can be analyzed later if required.

STOPP

$\Phi SIMPLE\_OPERATION$
$\Xi EVALUATION\_STACK$
$\Xi ERROR$

$Status' = stopped$
$WorkSpace'' = WorkSpace \oplus \{Twrd(-1) \mapsto Iptr'\}$
$Cycles = 11$
$Opr = \texttt{stopp}$

9.4 Power-up and Bootstrapping

The start of memory and the reset code are at standard addresses:

$$MemStart : TwordAddress$$
$$ResetCode : Address$$

These should normally be in RAM and ROM respectively for useful operation of the Transputer.

After start-up, the clock is initialized to zero for convenience and the processor is running.

InitTRANS

$$TRANS'$$

$$Oreg' = \mathbf{0}$$
$$Wptr' = MemStart$$
$$Iptr' = ResetCode$$
$$Clk' = 0$$
$$Status' = running$$

Note in particular that the *ErrorFlag* is not initialized.

It is assumed that a short bootstrap program is available in ROM to attain the above initialization conditions in an actual Transputer. Some instructions which are not in the instruction set defined here will be needed. For example, `clrhalterr` must be executed to disable halt-on-error mode. Further details of bootstrapping can be found in [224].

9.5 Combined Operations and Instructions

The Transputer *operations* defined here are:

$$\begin{aligned} \texttt{OPR} \mathrel{\widehat=} \; & \texttt{ADD} \lor \texttt{SUB} \lor \texttt{MUL} \lor \texttt{DIV} \lor \texttt{REM} \lor \texttt{SUM} \lor \texttt{DIFF} \lor \texttt{PROD} \lor \\ & \texttt{AND} \lor \texttt{OR} \lor \texttt{XOR} \lor \texttt{NOT} \lor \texttt{SHL} \lor \texttt{SHR} \lor \texttt{GT} \lor \texttt{REV} \lor \\ & \texttt{IN} \lor \texttt{OUT} \lor \texttt{SETERR} \lor \texttt{TESTERR} \lor \texttt{STOPERR} \lor \texttt{STOPP} \end{aligned}$$

The Transputer *instructions* included here are:

$$\begin{aligned} & INSTRUCTION \mathrel{\widehat=} \\ & \quad \texttt{PFIX} \lor \texttt{NFIX} \lor \texttt{ADC} \lor \texttt{LDL} \lor \texttt{STL} \lor \texttt{LDLP} \lor \texttt{J} \lor \texttt{CJ} \lor \texttt{EQC} \lor \texttt{OPR} \end{aligned}$$

When the state changes, only the main state components of the machine are of interest.

$$EXEC \mathrel{\widehat=} INSTRUCTION \upharpoonright (TRANS \land TRANS')$$

The operation of the Transputer consists of the initial state *InitTRANS* followed by a sequence of such instructions controlled by the contents of the memory currently pointed to by the *instruction pointer* before the execution of each instruction. The sequence continues until the processor is *stopped* by a `STOPP` or `STOPERR` instruction.

9.6 Conclusions

This specification is perhaps rather baroque for the number of processor instructions which it describes. This is because it has been produced from a more complete description of the actual Transputer instruction set [149]. The technique of factoring out common portions of specification becomes more effective for larger instruction sets.

Additionally the instructions are specified to the bit level. While this is useful for those wishing to refine this description further down into the levels of hardware, the compiler writer may prefer a more abstract description in a notation more akin to the description of the high-level programming language [62, 216].

Since arithmetic operations are defined in terms of words here, overflow conditions are easily detected. This is an aspect which is easily overlooked in more abstract specifications, but one which can often cause errors in practice. All resources in a real machine are finite whereas in a specification it is easy to include infinite objects (e.g., natural numbers).

V

Graphics

Some basic graphical concepts concerning pixels (‘picture elements’) are presented in Chapter 10. These are used to specify the raster-op function, an important graphical operation, in Chapter 11, and also a number of window systems later in the book.

Chapter 10

Basic Graphical Concepts

This chapter gives a formal framework to aid the description of pixels*, their organization into* pixel maps *and a number of* windows*. These definitions are used subsequently in Chapter 11 and Part VI.*

10.1 Background

The interest in formalizing aspects of computer graphics and interactive systems is gradually increasing [132]. For example, parts of the ISO/IEC GKS graphics standard have been formalized using Z [12, 13]. Early attempts have demonstrated the necessity of chosing an appropriate notation for the job [175]. Some formal notations have been designed with computer processing in mind [114] and readability can be a secondary consideration. However, in the case of standards, the latter becomes a much more important factor than the former.

The *Human-Computer Interface* (HCI) is an important part of most software systems, and window systems are currently the standard interface for the vast majority of workstations. Formalizing an HCI in a realistic and useful manner is a difficult task, but progress is being made in categorizing features of interfaces that may help to ensure their reliability in the future [128, 193]. There seems to be considerable scope for further work in this area.

10.2 Pixels

10.2.1 Pixel positions

A raster graphics display is made up of a set of pixels with positions or coordinates. These are normally defined in X–Y coordinate space. The display is a fixed size bounded rectangle in the X–Y plane.

$$Xsize, Ysize : \mathbb{N}_1$$

The offset in a particular direction is specified from zero up by convention. The position of a pixel may be specified by a pair of X–Y coordinates.

$$Xrange == 0 \,..\, Xsize - 1$$
$$Yrange == 0 \,..\, Ysize - 1$$
$$Pixel == Xrange \times Yrange$$

The pixel at $(0,0)$ is normally at the upper left-hand corner of the display and the pixel at $(Xsize-1, Ysize-1)$ is at the bottom right by convention on most graphical computer-based window systems.

Many operations are applied to pairs of pixels.

$$
\begin{array}{|l}
\textit{PixelPair} \\
\hline
pix_1, pix_2 : Pixel \\
x_1, x_2 : Xrange \\
y_1, y_2 : Yrange \\
\hline
pix_1 = (x_1, y_1) \\
pix_2 = (x_2, y_2) \\
\hline
\end{array}
$$

The '$+$', '$-$' and '$\leq$' operators may be overloaded to apply to pixel positions. '$+$' and '$-$' may be used for moving pixel areas around the display. '$\leq$' can be used to define pixel ordering from the top left to bottom right.

$$
\begin{array}{|l}
\_+\_, \\
\_-\_ : (Pixel \times Pixel) \pfun Pixel \\
\_\leq\_ : Pixel \rel Pixel \\
\hline
\forall\, PixelPair \bullet \\
\quad (x_1 + x_2 < Xsize \land y_1 + y_2 < Ysize) \Rightarrow \\
\qquad pix_1 + pix_2 = (x_1 + x_2, y_1 + y_2) \land \\
\quad (x_2 \leq x_1 \land y_2 \leq y_1) \Rightarrow \\
\qquad pix_1 - pix_2 = (x_1 - x_2, y_1 - y_2) \land \\
\quad pix_1 \leq pix_2 \Leftrightarrow x_1 \leq x_2 \land y_1 \leq y_2
\end{array}
$$

We can define the offset between any two pixel positions as a pixel offset. This is defined to wrap round the edge of the pixel area and thus is a total function.

$$
\begin{array}{|l}
offset : Pixel \to Pixel \tfun Pixel \\
\hline
\forall\, PixelPair \bullet \\
\quad offset\ pix_1\ pix_2 = ((x_1 + x_2) \bmod Xsize, (y_1 + y_2) \bmod Ysize)
\end{array}
$$

We can also overload the '$..$' operator to define a rectangular area of pixels.

$$
\begin{array}{|l}
\_..\_ : (Pixel \times Pixel) \to \mathbb{F}\, Pixel \\
\hline
\forall\, PixelPair \bullet \\
\quad pix_1 .. pix_2 = (x_1 .. x_2) \times (y_1 .. y_2)
\end{array}
$$

$$pix_1, pix_2 : Pixel \vdash pix_1 .. pix_2 = \{p : Pixel \mid pix_1 \leq p \land p \leq pix_2\}$$

A rectangular area of pixels can defined using any two opposing corners (e.g., returned using an attached mouse to sweep between the two). The following functions return the upper left and lower right pixel positions from two such pixel positions respectively.

$$
\begin{array}{|l}
\_ \underline{min} \_, \\
\_ \underline{max} \_ : (Pixel \times Pixel) \rightarrow Pixel \\
\hline
\forall\, PixelPair \bullet \\
\quad pix_1 \underline{min}\, pix_2 = (min\{x_1, x_2\}, min\{y_1, y_2\}) \land \\
\quad pix_1 \underline{max}\, pix_2 = (max\{x_1, x_2\}, max\{y_1, y_2\})
\end{array}
$$

10.2.2 Pixel maps

A raster graphics display has a number of bit-planes. This may be considered as the Z direction of the display.

$$
\begin{array}{|l}
Zsize : \mathbb{N}_1
\end{array}
$$

Each *bit* in a bit-plane has one of two values (cleared or set).

$$ClearVal == 0$$
$$SetVal == 1$$
$$BitVal == \{ClearVal, SetVal\}$$

The value of a pixel at a particular position may be modelled as a function from bit-plane number to bit value.

$$Zrange == 0 \,..\, Zsize - 1$$

$$Value == Zrange \rightarrow BitVal$$

If all the bits are clear the '*Value*' is considered '*Black*' and if they are all set it is considered '*White*'.

$$Black == (\, \mu\, val : Value \mid \operatorname{ran} val = \{ClearVal\}\,)$$

$$White == (\, \mu\, val : Value \mid \operatorname{ran} val = \{SetVal\}\,)$$

Note that if there is only one bit-plane (i.e., $Zsize = 1$) then pixel values can only be *Black* or *White*.

$$\vdash\ Zsize = 1 \Rightarrow Value = \{Black, White\}$$

A pixel map consists of a (partial) function from pixel positions to the value of the pixel contents. This can be used to describe part of a display, such as a window.

$$Pixmap == Pixel \pfun Value$$

Non-empty pixel maps may be of special interest.

$$Pixmap_1 == Pixmap \setminus \{\emptyset\}$$

Pixel maps are often rectangular in area. We can define such pixel maps using their bottom left and top right pixel positions.

$$Rectangle == \{map : Pixmap_1 \mid \exists_1\, p_1, p_2 : Pixel \bullet \operatorname{dom} map = p_1\,..\,p_2\}$$

Sometimes it is desirable to set all the range of a pixel map to a particular value, for

example when clearing a window down to the background colour. A function to set the range of a relation to a particular value is useful for this.

$$
\begin{array}{|l}
[P, V] \\
\hline
\mathit{setval} : V \to (P \pfun V) \to (P \pfun V) \\
\hline
\forall v : V;\ p : P \pfun V \bullet \\
\quad \mathit{setval}\ v\ p = (\ \mu\, m : P \pfun V \mid (\operatorname{dom} m = \operatorname{dom} p \land \operatorname{ran} m = \{v\})\ ) \\
\hline
\end{array}
$$

The following laws apply:

$$
\begin{array}{rcl}
p : \mathit{Pixmap};\ v : \mathit{Value} & \vdash & (\mathit{setval}\ v)^+\ p = \mathit{setval}\ v\ p \\
v : \mathit{Value} & \vdash & \mathit{setval}\ v\ \emptyset[\mathit{Pixel} \times \mathit{Value}] = \emptyset
\end{array}
$$

Two pixel maps may overlap. For example, one window may obscured by another. This can be captured as a relation between pixel maps:

$$
\begin{array}{|l}
[P, V] \\
\hline
\_\ \underline{\mathit{overlaps}}\ \_ : (P \pfun V) \leftrightarrow (P \pfun V) \\
\hline
\forall p_1, p_2 : P \pfun V \bullet \\
\quad p_1\ \underline{\mathit{overlaps}}\ p_2 \Leftrightarrow \operatorname{dom} p_1 \cap \operatorname{dom} p_2 \neq \emptyset \\
\hline
\end{array}
$$

A sequence of pixel maps may be overlaid in the order given by the sequence. It is convenient to define a distributed overriding operator for this.

$$
\begin{array}{|l}
[P, V] \\
\hline
\oplus/ : \operatorname{seq}(P \pfun V) \to (P \pfun V) \\
\hline
\oplus/\langle\rangle = \emptyset \\
\forall p : P \pfun V \bullet \oplus/\langle p \rangle = p \\
\forall s, t : \operatorname{seq}(P \pfun V) \bullet \oplus/(s \frown t) = (\oplus/s) \oplus (\oplus/t) \\
\hline
\end{array}
$$

Distributed overriding is particularly useful for defining the view on a screen of a display, given a sequence of possibly overlapping pixel maps.

The following laws apply for the distributed overriding operator:

$$
\begin{array}{rcl}
p_1, p_2 : \mathit{Pixmap} & \vdash & \oplus/\langle p_1, p_2 \rangle = p_1 \oplus p_2 \\
p : \mathit{Pixmap};\ s : \operatorname{seq} \mathit{Pixmap} & \vdash & \oplus/(s \frown \langle p \rangle) = (\oplus/s) \oplus p \\
p : \mathit{Pixmap};\ s : \operatorname{seq} \mathit{Pixmap} & \vdash & \oplus/(\langle p \rangle \frown s) = p \oplus (\oplus/s) \\
s : \operatorname{seq}_1 \mathit{Pixmap} & \vdash & \oplus/s = (\oplus/(\mathit{front}\ s)) \oplus (\mathit{last}\ s) \\
 & & \quad = (\mathit{head}\ s) \oplus (\oplus/(\mathit{tail}\ s)) \\
s : \operatorname{seq} \mathit{Pixmap} & \vdash & \operatorname{dom}(\oplus/s) = \operatorname{dom}(\bigcup(\operatorname{ran} s)) \\
s : \operatorname{seq} \mathit{Pixmap} & \vdash & \operatorname{ran}(\oplus/s) \subseteq \operatorname{ran}(\bigcup(\operatorname{ran} s)) \\
s : \operatorname{seq} \mathit{Pixmap} & \vdash & \bigcap(\operatorname{ran} s) = \emptyset \Rightarrow \oplus/s = \bigcup(\operatorname{ran} s)
\end{array}
$$

10.3 Windows

A series of windows on a display screen may be viewed as a sequence in which each window is laid on the screen in the order defined by the sequence (bottom first, top last). If some of the windows are removed from a sequence, it is sometimes desirable to 'squash' the remaining windows into a sequence again, keeping the windows in the same order, but ensuring that they are numbered sequentially from one upwards. The standard Z toolkit *squash* function (see page 118 of [381]) can be used to perform this operation.

If the windows in a sequence overlap, it is useful to be able to move selected windows so that their contents may be viewed (or hidden). This is analogous to shuffling a pile of sheets of paper (windows) on a desk (screen). Note that the sheets of paper may be of different sizes and in different positions on the desk.

We can define functions to 'select' and 'remove' a set of windows from a sequence using their identifiers rather than their position in the pile.

$$
\begin{array}{|l}
[W] \\
\_ \underline{select} \_, \\
\_ \underline{remove} \_ : \operatorname{seq} W \times \mathbb{P} W \rightarrow \operatorname{seq} W \\
\hline
\forall s : \operatorname{seq} W;\ w : \mathbb{P} W \bullet \\
\quad s \underline{select}\ w = squash(s \rhd w) \land \\
\quad s \underline{remove}\ w = squash(s \rhd w) \\
\hline
\end{array}
$$

We can then 'raise' or 'lower' these windows to the end or beginning of the sequence as required.

$$
\begin{array}{|l}
[W] \\
\_ \underline{raise} \_, \\
\_ \underline{lower} \_ : \operatorname{seq} W \times \mathbb{P} W \rightarrow \operatorname{seq} W \\
\hline
\forall s : \operatorname{seq} W;\ w : \mathbb{P} W \bullet \\
\quad s \underline{raise}\ w = (s \underline{remove}\ w) \frown (s \underline{select}\ w) \land \\
\quad s \underline{lower}\ w = (s \underline{select}\ w) \frown (s \underline{remove}\ w) \\
\hline
\end{array}
$$

Every window in a system usually has an identifier, denoted '*Window*', which allows it to be accessed uniquely.

$$[Window]$$

The generic functions defined in this section will normally be applied to such identifiers.

Windows often contain text. Thus, a text string is needed sometimes (e.g., for a title of a window). This is denoted as '*String*'. The string may be empty.

$$[String]$$

$$\mid\ '' : String$$

This concludes the basic definitions which will be used as required in Chapter 11 and Part VI.

Chapter 11

Raster-Op Functions

Raster-op *functions are useful when moving areas of pixels (e.g., parts of windows) around the screen on graphics display systems. Raster-op is now widely used in graphics systems, in particular for window systems. This chapter formally specifies raster-op functions and gives an example of its use. It makes use of some of the basic graphic concepts formalized in Chapter 10. Readers not familiar with 'raster-op' may prefer to do some background reading first (e.g., see [158, 312]).*

11.1 Pixel Operations

11.1.1 Pixel values

Given two pixel values, we may perform a bit-wise '*NAND*' ('not and') on the two values to produce a new value. If both bits are set then the result is clear; if either bit is clear then the result is set.

$$
\begin{array}{|l}
\_ \underline{NAND} \_ : (Value \times Value) \rightarrow Value \\
\hline
\forall\, val_1, val_2, val : Value \bullet \\
\quad val_1 \ \underline{NAND}\ val_2 = val \Leftrightarrow \\
\quad\quad (\forall\, n : Zrange \bullet \\
\quad\quad\quad (val_1\ n = SetVal \wedge val_2\ n = SetVal) \Rightarrow val\ n = ClearVal \wedge \\
\quad\quad\quad (val_1\ n = ClearVal \vee val_2\ n = ClearVal) \Rightarrow val\ n = SetVal\,)
\end{array}
$$

All the other binary and unary logical functions may be defined in terms of this function. For example, we can define a unary '*NOT*' function.

$$
\begin{array}{|l}
NOT : Value \rightarrow Value \\
\hline
\forall\, val : Value \bullet \\
\quad NOT\ val = val\ \underline{NAND}\ val
\end{array}
$$

We can also define binary '*AND*', '*OR*' and '*XOR*' functions.

$$
\begin{array}{|l}
\_ \underline{AND} \_, \\
\_ \underline{OR} \_, \\
\_ \underline{XOR} \_ : (Value \times Value) \rightarrow Value \\
\hline
\forall\, val_1, val_2 : Value \bullet \\
\quad val_1 \underline{AND}\, val_2 = NOT(val_1 \underline{NAND}\, val_2) \land \\
\quad val_1 \underline{OR}\, val_2 = (NOT\ val_1) \underline{NAND}\, (NOT\ val_2) \land \\
\quad val_1 \underline{XOR}\, val_2 = (val_1 \underline{OR}\, val_2) \underline{AND}\, (val_1 \underline{NAND}\, val_2)
\end{array}
$$

11.1.2 Pixel maps

A pixel map is considered to have the same shape as another if it can be 'moved' using a unique one-to-one function ($(offset\ pix_0)$ in the definition below) to give it the same domain. Intuitively this implies that each of the pixel maps could be moved *en masse* about the domain space so that its domain is exactly the same as the other map.

$$
\begin{array}{|l}
\_ \underline{sameshape} \_ : Pixmap \leftrightarrow Pixmap \\
\hline
\forall\, map_1, map_2 : Pixmap \bullet \\
\quad map_1 \underline{sameshape}\, map_2 \Leftrightarrow \\
\qquad (\exists\, pix_0 : Pixel \bullet \\
\qquad\quad \mathrm{dom}((offset\ pix_0) \fatsemi map_1) = \mathrm{dom}\, map_2\ )
\end{array}
$$

Consider a pair of (possibly overlapping) pixel maps which have the same shape.

$$
\begin{array}{|l}
\textit{MapPair} \\
\hline
map_1, \\
map_2 : Pixmap \\
\hline
map_1 \underline{sameshape}\, map_2 \\
\hline
\end{array}
$$

Two pixels in a pair of pixel maps with the same shape are considered to have the same position if the same offset function which relates the two maps also relates the two pixels. This is captured in '*samepos*' relation below. If the relation is true then it implies that the two pixels are in the same relative position within two pixel maps with the same shape. That is to say, if one of the pixel maps were to be moved on top of the other then one pixel would be exactly on top of the other.

$$
\begin{array}{|l}
\_ \underline{samepos} \_ : (Pixel \times Pixmap) \leftrightarrow (Pixel \times Pixmap) \\
\hline
\forall\, pix_1, pix_2 : Pixel;\ MapPair \mid \\
\quad pix_1 \in \mathrm{dom}\, map_1 \land pix_2 \in \mathrm{dom}\, map_2 \bullet \\
\qquad (pix_1, map_1) \underline{samepos}\, (pix_2, map_2) \Leftrightarrow \\
\qquad\quad (\exists\, pix_0 : Pixel \bullet \\
\qquad\qquad \mathrm{dom}((offset\ pix_0) \fatsemi map_1) = \mathrm{dom}\, map_2 \land \\
\qquad\qquad pix_1 \mapsto pix_2 \in offset\ pix_0\ )
\end{array}
$$

We may define operations similar to the bit-wise operations on values to apply to pixel maps. In this case by convention, the last operand of the function has the same domain as the result of the function. We start by defining the '*nand*' function.

$$\_ \underline{nand} \_ : (Pixmap \times Pixmap) \pfun Pixmap$$

$$\begin{aligned}
&\forall MapPair;\ map : Pixmap \bullet \\
&\quad map_1 \underline{nand}\ map_2 = map \Leftrightarrow \\
&\qquad \dom map = \dom map_2 \land \\
&\qquad (\forall pix_1 : \dom map_1;\ pix_2 : \dom map_2 \mid \\
&\qquad\quad (pix_1, map_1) \underline{samepos}\ (pix_2, map_2) \bullet \\
&\qquad\qquad map_1(pix_1)\ \underline{NAND}\ map_2(pix_2) = map\ pix_2\ )
\end{aligned}$$

Note that this operation is not commutative, unlike '*NAND*' since the second operand defines the domain of the result of the function. Specifically, if the domains of the pixel maps differ, the ordering is important. If the domains are the same then the ordering is unimportant.

$$\begin{aligned}
MapPair \vdash\ & \dom map_1 = \dom map_2 \Rightarrow \\
& map_1 \underline{nand}\ map_2 = map_2 \underline{nand}\ map_1
\end{aligned}$$

Using the basic '*nand*' function, we may define fifteen further operations. Four of these degenerate to monadic functions. '*noop*' leaves a pixel map unaffected, '*not*' inverts all bits, '*clear*' clears all the bits and '*set*' sets all the bits.

$$noop, \quad not, \quad clear, \quad set : Pixmap \to Pixmap$$

$$\begin{aligned}
&\forall map : Pixmap \bullet \\
&\quad noop\ map = map \land \\
&\quad not\ map = map \underline{nand}\ map \land \\
&\quad clear\ map = map \underline{nand}\ (not\ map) \land \\
&\quad set\ map = not(clear\ map)
\end{aligned}$$

There are five more important operators. These correspond to standard logical operations except '*copy*' which is extremely useful for moving pixel maps.

$$\_ \underline{and} \_, \quad \_ \underline{or} \_, \quad \_ \underline{xor} \_, \quad \_ \underline{nor} \_, \quad \_ \underline{copy} \_ : (Pixmap \times Pixmap) \pfun Pixmap$$

$$\begin{aligned}
&\forall MapPair \bullet \\
&\quad map_1 \underline{and}\ map_2 = not(map_1 \underline{nand}\ map_2) \land \\
&\quad map_1 \underline{or}\ map_2 = (not\ map_1) \underline{nand}\ (not\ map_2) \land \\
&\quad map_1 \underline{xor}\ map_2 = (map_1 \underline{or}\ map_2) \underline{and}\ (map_1 \underline{nand}\ map_2) \land \\
&\quad map_1 \underline{nor}\ map_2 = not(map_1 \underline{or}\ map_2) \land \\
&\quad map_1 \underline{copy}\ map_2 = map_1 \underline{or}\ (clear\ map_2)
\end{aligned}$$

The rest of the operations are used less often but are detailed here for completeness.

$$
\begin{array}{|l}
\_ \underline{copyInverted} \_, \\
\_ \underline{andReverse} \_, \\
\_ \underline{andInverted} \_, \\
\_ \underline{orReverse} \_, \\
\_ \underline{orInverted} \_, \\
\_ \underline{equiv} \_ : (Pixmap \times Pixmap) \pfun Pixmap \\
\hline
\forall MapPair \bullet \\
\quad map_1 \ \underline{copyInverted} \ map_2 = (not\ map_1) \ \underline{or} \ (clear\ map_2) \land \\
\quad map_1 \ \underline{andReverse} \ map_2 = map_1 \ \underline{and} \ (not\ map_2) \land \\
\quad map_1 \ \underline{andInverted} \ map_2 = (not\ map_1) \ \underline{and} \ map_2 \land \\
\quad map_1 \ \underline{orReverse} \ map_2 = map_1 \ \underline{or} \ (not\ map_2) \land \\
\quad map_1 \ \underline{orInverted} \ map_2 = (not\ map_1) \ \underline{or} \ map_2 \land \\
\quad map_1 \ \underline{equiv} \ map_2 = (not\ map_1) \ \underline{xor} \ map_2
\end{array}
$$

This covers the sixteen possible raster-op Boolean functions on two values.

11.2 Display Operations

A display screen consists of a pixel map.

$$
\begin{array}{|l}
\hline
Display \\
\hline
screen : Pixmap \\
\hline
\end{array}
$$

During changes to the screen, its size does not change although the pixel values displayed on the screen may be updated.

$$
\begin{array}{|l}
\hline
\Delta Display \\
\hline
Display \\
Display' \\
\hline
\dom screen' = \dom screen \\
\hline
\end{array}
$$

The screen may be updated using one of the raster-op functions previously defined. Some functions require a source and destination area while others degenerate into a single area.

$$
\begin{array}{|l}
\hline
RasterOp_1 \\
\hline
\Delta Display \\
area? : \power Pixel \\
op? : Pixmap \fun Pixmap \\
\hline
screen' = screen \oplus op?(area? \rhd screen) \\
\hline
\end{array}
$$

RasterOp$_2$

$\Delta Display$
$area? : \mathbb{P}\, Pixel$
$from? : Pixel$
$op? : (Pixmap \times Pixmap) \pfun Pixmap$

$screen' = screen \oplus$
$\quad op?\,(\,((offset\, from?)(\!|\, area? \,|\!)) \rhd screen\,,\, area? \rhd screen\,)$

11.3 An Example – Swapping Pixel Maps

Consider two separate non-overlapping pixel maps with the same shape.

SepPair

MapPair

$\neg\, map_1\ \underline{overlaps}\ map_2$

$\Delta SepPair \mathrel{\widehat=} SepPair \land SepPair'$

We may swap a pair of pixel maps with the same shape using the '*copy*' operation.

CopySwap

$\Delta SepPair$

$map'_1 = map_2\ \underline{copy}\ map_1$

$map'_2 = map_1\ \underline{copy}\ map_2$

In practice, these two '*copy*' operations cannot be carried out simultaneously. A third copy is necessary and additionally a temporary pixel map area is required. This can easily be expressed by using three schemas, one for each operation, and then combining them using the schema composition operator ('$\fatsemi$'). This is left as a exercise for the reader.

Alternatively, the '*xor*' raster-op function may be used. Three sequential operations are still necessary, but the use of a temporary buffer area is eliminated. The following two schemas '*xor*' one or other of a pair of pixel maps with its opposite number.

Xor$_1$

$\Delta SepPair$

$map'_1 = map_2\ \underline{xor}\ map_1$

$map'_2 = map_2$

Xor$_2$

$\Delta SepPair$

$map'_1 = map_1$

$map'_2 = map_1\ \underline{xor}\ map_2$

A swap may be achieved between the two pixel maps by applying '*xor*' three times sequentially as follows.

$$XorSwap \mathrel{\widehat=} Xor_1 \fatsemi Xor_2 \fatsemi Xor_1$$

By symmetry, the following is also true.

$$\Delta SepPair \vdash XorSwap \Leftrightarrow (Xor_2 \fatsemi Xor_1 \fatsemi Xor_2)$$

The *XorSwap* operation has exactly the same effect as using the '*copy*' operator twice simultaneously, as in the *CopySwap* operation.

$$\Delta SepPair \vdash CopySwap \Leftrightarrow XorSwap$$

11.4 Conclusion

In this chapter we have formally specified the raster-op function and how this may be applied to a graphics display. We have given an example of its use for swapping areas of a display. By defining operations performed on rectangular areas, we have specified some of the underlying operations necessary for a window system. Three different window systems are formalized in Part VI.

VI

Window Systems

A number of operations for three window systems are formally specified using Z, namely WM in Chapter 12, the Blit in Chapter 13, and the widely used X window system in Chapter 14. The specifications make use of some of the graphical definitions in Part V. Finally, Chapter 15 briefly compares the three window systems and draws some general conclusions about the use of Z for specifying realistically sized systems.

Chapter 12

The ITC 'WM' Window Manager

WM, part of the 'Andrew' distributed system, is a window manager developed at the Information Technology Center (ITC) at Carnegie-Mellon University (CMU) [356, 415]. This runs on UNIX *workstations designed to be networked on a very large scale (c5,000–10,000 nodes for the entire campus at CMU). Because of the distributed file system, any authorized person may also use any other workstation on the network, and indeed create windows on other workstations remotely. Subsequently this was implemented as the Andrew Toolkit under the X window system [325]. This chapter, and the following two chapters on the Blit and X window systems respectively, make use of graphical concepts defined in Part V.*

12.1 System State

The state of the system is introduced in stages. In this model we consider a single machine for simplicity since we are concerned with how the window manager works rather than how the network operates. Operations over the network will be detailed later.

Each window has a number of pieces of information associated with it. These include a header area for titles and other information, and a separate body area to hold the actual contents of the window. These do not overlap and together they make up the pixel map of the displayed window. In practice, the header is a thin rectangular area just above its associated body.

Map

$$header, body, map : Pixmap$$
$$area : \mathbb{P}\, Pixel$$

$$\langle header, body \rangle \text{ partition } map$$
$$area = \operatorname{dom} map$$

The user can request a window to lie within a specified range of dimensions and can also explicitly ask for a window body to be hidden from view or exposed on the screen. Each window has a title which can be set by the user. This information is used by the window manager to lay out the window on the screen, although there is no guarantee that what the user asks for is what the user gets!

$$HideExpose ::= Hide \mid Expose$$

Control

$$title : String$$
$$control : HideExpose$$
$$xylimits : Pixel \times Pixel$$
$$(first\ xylimits) \leq (second\ xylimits)$$

Together these make up the information describing a particular window.

$$Info \mathrel{\widehat=} Map \land Control$$

There are a finite number of windows on a particular screen. One of these is considered to be the currently selected window. This may be 'undefined' sometimes. Most *WM* library functions take effect on the currently selected window. Each window has information, including a pixel map, associated with it.

$$Undefined : Window$$

WM$_0$

$$windows : \mathbb{F}\ Window$$
$$current : Window$$
$$contents : Window \pfun Info$$
$$Undefined \notin windows$$
$$windows = \mathrm{dom}\ contents$$
$$current \in windows \cup \{Undefined\}$$

The display screen consists of the background overlaid with windows. The window pixel maps do not overlap. All windows are contained within the background area.

WM$_1$

$$WM_0$$
$$maps : Window \pfun Pixmap$$
$$areas : Window \pfun (\mathbb{P}\ Pixel)$$
$$screen, background : Pixmap$$
$$maps = contents \fatsemi (\lambda\ Info \bullet map)$$
$$areas = contents \fatsemi (\lambda\ Info \bullet area)$$
$$\mathsf{disjoint}\ areas$$
$$\bigcup(\mathrm{ran}\ areas) \subseteq \mathrm{dom}\ background$$
$$screen = background \oplus \bigcup(\mathrm{ran}\ maps)$$

You can have as many windows as you need, subject to the restriction that the *WM* process can handle at most 20 windows, including hidden windows and windows requested by other programs, at one time.

$$MaxWindows : \mathbb{N}$$
$$MaxWindows = 20$$

We can include this limitation in our model of the state.

$$WM_2 \mathrel{\widehat=} [\, WM_1 \mid \#windows \leq MaxWindows \,]$$

The size of a window on the user's display is one of the resources that the *Window Manager* allocates. A program can request a given size, and *WM* will take the requested size into account when making decisions, but it does not guarantee a particular size. This process is modelled as a function of the system. The number of windows is not changed by this function. Additionally, control information supplied by the user is left unchanged.

$$WINDOWS == Window \pfun Info$$

WM
$$WM_2$$
$$adjust : WINDOWS \to WINDOWS$$

$$\forall w, w' : WINDOWS \mid w' = adjust\ w \bullet$$
$$\quad \#w' = \#w \land$$
$$\quad w' \fatsemi (\lambda Info \bullet \theta Control) = w \fatsemi (\lambda Info \bullet \theta Control)$$

Initially there are no windows and the current window is undefined.

InitWM
$$WM'$$

$$windows' = \emptyset$$
$$current' = Undefined$$

Operations change the state of the system. However the background and hence the size of the screen remains constant. Additionally the algorithm to adjust the size of windows does not change.

ΔWM
$$WM$$
$$WM'$$

$$background' = background$$
$$adjust' = adjust$$

Sometimes the state of the system is unaffected during an operation.

$$\Xi WM \mathrel{\widehat=} [\, \Delta WM \mid \theta WM' = \theta WM \,]$$

Many operations are concerned with the current window. Hence we define a schema giving a partial specification covering all common aspects of such operations. This

can be used to shorten subsequence specifications of these operations and reduce repetition. The names of such schemas are prepended with 'Φ' to distinguish these from actual operations.

$$
\begin{array}{|l}
\hline
\Phi Current \\
\Delta WM \\
Info \\
Info' \\
\hline
current \in windows \\
current' = current \\
\theta Info = contents\ current \\
contents' = adjust\ (contents \oplus \{current \mapsto \theta Info'\}) \\
\hline
\end{array}
$$

This leaves a valid current window the same, but updates the information associated with it in some (as yet unspecified) way.

12.2 Window Operations

12.2.1 Creation and deletion

When a window is created, the system adjusts all the windows in the system appropriately. The window body is exposed when it is created. In practice, the operation also takes the name of a host as input since a window may be created anywhere on the network of workstations. However this is detailed later.

$$
\begin{array}{|l}
\hline
NewWindow \\
\Delta WM \\
w! : Window \\
Info \\
\hline
\#windows < MaxWindows \\
w! \notin windows \cup \{Undefined\} \\
current' = w! \\
control = Expose \\
contents' = adjust\ (contents \cup \{w! \mapsto \theta Info\}) \\
\hline
\end{array}
$$

The currently selected window can be deleted.

$$
\begin{array}{|l}
\hline
DeleteWindow \\
\Delta WM \\
\hline
current \in windows \\
current' = Undefined \\
contents' = adjust\ (\{current\} \lhd contents) \\
\hline
\end{array}
$$

12.2.2 Window size

A program can request a given size range, and *WM* will take the requested size into account when making decisions, but it does not guarantee a particular size. The rest of the window information is unaffected. The windows will be adjusted by the system as necessary (i.e., any or all of the displayed windows may change shape as a result of setting the size of one particular window).

SetDimensions

$\Phi Current$
$minxy?, maxxy? : Pixel$

$map' = map$

$title' = title$

$control' = control$

$xylimits' = (minxy?, maxxy?)$

The size of the body of the currently selected window can be returned. If the window is actually hidden (i.e., *WM* has adjusted the window to display the header only), then the returned size is empty.

GetDimensions

$\Phi Current$
$wh! : Pixel$
$xy_1, xy_2 : Pixel$

$\theta Info' = \theta Info$

$\mathrm{dom}\, body = xy_1 \mathinner{..} xy_2$

$wh! = xy_2 - xy_1$

12.2.3 Windows visibility

A window is considered 'visible' when both its header and its body are displayed and 'hidden' when only its header is displayed. Windows are visible when they are first created and remain so unless the user hides them. A program can also control window visibility. A visible window may be hidden.

HideMe

$\Phi Current$

$map' = map$

$title' = title$

$control' = Hide$

$xylimits' = xylimits$

Similarly, a hidden window may be exposed.

ExposeMe

$\Phi Current$

$map' = map$

$title' = title$

$control' = Expose$

$xylimits' = xylimits$

Note that the state of '*control*' before the operation has not been checked above, so $HideMe_0$ will leave a hidden window hidden and $ExposeMe_0$ will leave a visible window exposed.

12.2.4 Other operations

A window may be explicitly selected as the current window, until another window is selected or created. All output will be sent to the selected window.

SelectWindow

ΔWM

$w? : Window$

$w? \in windows$

$current' = w?$

$contents' = contents$

The title of a window may be set. This involves placing a text string in the header section of the window contents.

SetTitle

$\Phi Current$

$s? : String$

$map' = map$

$title' = s?$

$control' = control$

$xylimits' = xylimits$

The body of the currently selected window may be set to white.

ClearWindow

$\Phi Current$

$header' = header$

$body' = setval\ White\ body$

$title' = title$

$control' = control$

$xylimits' = xylimits$

Other operations supplied by the *WM* library include line, text and string drawing, raster operations, operations to save and restore parts of the picture, input handling, menus, mouse input, etc. In addition, new operations may be added; at the time that this specification was originally formulated *WM* was still under development.

12.3 Errors

There is a *Null* window identifier which is never a valid window.

$$Null : Window$$

$$WM_{err} \mathrel{\widehat=} [\, WM \mid Null \notin windows \,]$$

ΔWM and ΞWM are redefined appropriately.

Some operations return a window identifier. If this is non-null then the operation is successful.

Success$_{WM}$

ΔWM_{err}

$w! : Window$

$w! \neq Null$

Alternatively a error may occur. There is a limit on the number of windows which *WM* can handle. This could cause an error when creating a new window.

TooManyWindows

ΞWM_{err}

$w! : Window$

$\#windows \geq MaxWindows$

$w! = Null$

We can now make the operation to create a new window total.

$$NewWindow_1 \mathrel{\widehat=} (NewWindow \wedge Success_{WM}) \\ \vee\ TooManyWindows$$

The current window may be undefined when one is required:

NoCurrentWindow

$$\Xi WM_{err}$$

$$current = Undefined$$

Many operations may return with this error:

$$DeleteWindow_1 \mathrel{\widehat=} DeleteWindow \lor NoCurrentWindow$$

$$SetDimensions_1 \mathrel{\widehat=} SetDimensions \lor NoCurrentWindow$$

$$GetDimensions_1 \mathrel{\widehat=} GetDimensions \lor NoCurrentWindow$$

$$SetTitle_1 \mathrel{\widehat=} SetTitle \lor NoCurrentWindow$$

$$ClearWindow_1 \mathrel{\widehat=} ClearWindow \lor NoCurrentWindow$$

An invalid window may be selected:

InvalidWindow

$$\Xi WM_{err}$$
$$w? : Window$$

$$w? \notin windows$$

$$SelectWindow_1 \mathrel{\widehat=} SelectWindow \lor InvalidWindow$$

A window may always be hidden or exposed, even if this does not affect its state, so no error schemas are required for the *HideMe* and *ExposeMe* operations.

12.4 The ITC Network

In practice, as mentioned previously, there are many window managers, each running on a host workstation on a large network. Some hosts are running *WM*. All workstations have unique host names and all windows have unique identifiers across the network.

ITC

$$hosts : \mathbb{P}\, String$$
$$wms : String \pfun WM$$

$$\mathrm{dom}\, wms \subseteq hosts$$

$$\mathsf{disjoint}\ (wms \fatsemi (\lambda\, WM \bullet windows))$$

Initially there are no hosts (and hence no window managers) on the network.

$$InitITC \mathrel{\widehat=} [\, ITC' \mid hosts' = \emptyset \,]$$

Operations cause changes on the network.

$$\Delta ITC \mathrel{\widehat=} ITC \land ITC'$$

Hosts can be added to the system (e.g., booting up) and removed (e.g., crashing or powering down).

AddHost

ΔITC
$host? : String$

$host? \notin hosts \cup \{``\,''\}$

$hosts' = hosts \cup \{host?\}$

$wms' = wms$

RemoveHost

ΔITC
$host? : String$

$host? \in hosts$

$hosts' = hosts \setminus \{host?\}$

$wms' = \{host?\} \lhd wms$

Operations can be initiated on a particular 'local' host. These do not affect the host names on the network.

ΦHost

ΔITC
$localhost : String$

$hosts' = hosts$

$localhost \in hosts$

For example, *WM* may be executed on a host, and may be subsequently killed.

ExecWM

$\Phi Host$
$initwm : WM$

$localhost \notin \mathrm{dom}\, wms$

$wms' = wms \cup \{localhost \mapsto initwm\}$

KillWM

$\Phi Host$

$localhost \in \mathrm{dom}\, wms$

$wms' = \{localhost\} \lhd wms$

WM operations can be modelled in the global context of the network by updating the state of *WM* on a particular local host which is already running *WM*.

$$
\begin{array}{|l}
\hline
\Phi WM \\
\Phi Host \\
\Delta WM \\
host : String \\
\hline
\theta WM = wms\ host \\
\theta WM' = wms'\ host \\
\hline
\end{array}
$$

The *Window Manager* on the local host can be requested to create new windows on any machine on the ITC network that is running a *WM* process by supplying the appropriate host name. Alternatively, specifying a null host parameter results in a request for a window on the local machine. This is the normal mode of operation.

$$
\begin{array}{|l}
\hline
NewWindow_{ITC} \\
NewWindow_1 \\
\Phi WM \\
host? : String \\
\hline
host? = `' \Rightarrow host = localhost \\
host? \neq `' \Rightarrow host = host? \\
\hline
\end{array}
$$

Other window operations discussed previously may be described in the global network context. For example:

$$DeleteWindow_{ITC} \mathrel{\widehat=} DeleteWindow_1 \land \Phi WM$$

If the current window is on the local machine, then the operation is executed locally, otherwise it is carried out over the network.

12.5 Simplifications and Assumptions

For the purposes of brevity, the pop-up menus supplied by *WM* have been ignored in the description given. These could easily be added to the state specification by including them as extra window information in a schema called *Menu*.

$$InfoMenu \mathrel{\widehat=} Info \land Menu$$

In practice, windows are not adjusted immediately, but when the *Window Manager* next makes a size decision (e.g., when the user requests *WM* to proportion the windows). This is not modelled here. In addition, the way in which the windows are proportioned is not specified since this is not covered in the original documentation used to formulate this specification.

12.6 Comments

The *WM* window manager provides a simple system with non-overlapping windows. Hence no notion of window ordering is required. The idea of a 'current' window for each process using *WM* means that this information is held as part of the state of the system and need not be specified as input to many window operations. Windows are

automatically reduced in size by the system when there is not enough space on the screen. This simplifies the task of organizing windows for the user.

The next chapter considers another window system, the Blit.

Chapter 13

Blit Windows

The Blit [16, 329], developed at Bell Labs, Murray Hill, is more like an intelligent terminal than a workstation. It is diskless and, at the time of its design, interacted with a remote host running the Bell Labs Eighth Edition UNIX *via a 9600 baud (slow) RS-232 serial line. It has its own simple process scheduler and a bit-mapped display. Programs can be run on the Blit (downloaded from the remote host), on the remote host itself using a standard window terminal emulator process on the Blit, or on both using two special purpose programs which interact with each other over the serial line. Deciding how to split a program between the Blit and remote host is a tricky but interesting problem.*

13.1 System State

The Blit contains 'layers' which are analogous to windows on most other systems. However there is no protection between layers. Each layer has an rectangular region on the screen associated with it. Here we model this simply as a partial function from pixel points to values.

$$Layer == Window$$
$$Point == Pixel$$

Each pixel point is two-valued – i.e., each window is a simple bit map.

$$Zsize = 1$$

Several layers (with associated rectangular windows) may exist simultaneously. The layers are ordered as a sequence for reasons that will be seen in a moment. There is an invalid null layer for error returns (see later).

$$NullLayer : Layer$$

$$
\begin{array}{|l}
\textit{Blit}_0 \\
\hline
\textit{layers} : \operatorname{seq} \textit{Layer} \\
\textit{windows} : \textit{Layer} \pfun \textit{Rectangle} \\
\textit{rects} : \operatorname{seq} \textit{Rectangle} \\
\hline
\operatorname{ran} \textit{layers} = \operatorname{dom} \textit{windows} \\
\textit{NullLayer} \notin \operatorname{ran} \textit{layers} \\
\textit{rects} = \textit{layers} \fatsemi \textit{windows} \\
\hline
\end{array}
$$

Several processes may also exist simultaneously in the Blit. Each process has an associated program and state. A process may be disabled or enabled. However the rest of this specification is not concerned with the state of processes, but it is included here for completeness. Each process is normally associated with a layer. Creating a process without a layer or vice versa is dangerous.

$$[\textit{Program}, \textit{StateInfo}]$$

$$
\begin{array}{|l}
\textit{Proc} \\
\hline
\textit{prog} : \textit{Program} \\
\textit{state} : \textit{StateInfo} \\
\textit{l} : \textit{Layer} \\
\hline
\end{array}
$$

The Blit includes no window 'manager' as such since any process has access to the entire screen. There are a series of processes in the system each identified uniquely by a process id. There is a null invalid id which is returned by operations to indicate an error. One of the processes may be assigned to receive mouse and keyboard events.

$$[\textit{Id}]$$

$$\textit{NullId} : \textit{Id}$$

$$
\begin{array}{|l}
\textit{Blit}_1 \\
\hline
\textit{Blit}_0 \\
\textit{procs} : \textit{Id} \pfun \textit{Proc} \\
\textit{receiver} : \textit{Id} \\
\hline
\textit{NullId} \notin \operatorname{dom} \textit{procs} \\
\textit{receiver} \in \operatorname{dom} \textit{procs} \cup \{\textit{NullId}\} \\
\forall \textit{proc} : \operatorname{ran} \textit{procs} \bullet \textit{proc}.\textit{l} \in \operatorname{ran} \textit{layers} \cup \{\textit{NullLayer}\} \\
\hline
\end{array}
$$

A process may be associated with a default terminal emulation program if desired.

$$\textit{Blit}_2 \mathrel{\widehat=} [\, \textit{Blit}_1;\ \textit{default} : \textit{Program} \,]$$

The background may be considered as another layer which defines the size of the screen. The display consists of all the layers overlaid on top of the background. All layers are contained within the background. The order is determined from the position in the sequence (first at the bottom, last on top).

Blit

$$Blit_2$$
$$screen, background : Rectangle$$

$$\mathrm{dom}(\oplus/rects) \subseteq \mathrm{dom}\, background$$
$$screen = background \oplus (\oplus/rects)$$

Initially there are no layers or processes in the system.

InitBlit

$$Blit'$$

$$layers' = \langle\rangle$$
$$procs' = \emptyset$$
$$receiver' = NullId$$

Operations do not change the default terminal program or the background of the display.

Δ*Blit*

$$Blit$$
$$Blit'$$

$$default' = default$$
$$background' = background$$

Some operations do not affect the state of the Blit.

$$\Xi Blit \mathrel{\widehat=} [\, \Delta Blit \mid \theta Blit' = \theta Blit \,]$$

Often operations only affect layers and all the processes in the system are left unaffected.

Φ*Layer*

$$\Delta Blit$$

$$procs' = procs$$
$$receiver' = receiver$$

Similarly, processes are often changed while leaving all the layers in the system unaffected.

Φ*Proc*

$$\Delta Blit$$

$$layers' = layers$$
$$windows' = windows$$

13.2 System Operations

13.2.1 New layers

A layer may be created in a specified rectangle in the physical display bit-map. The address of the layer is returned.

NewLayer

$\Phi Layer$
$r? : Rectangle$
$l! : Layer$

$\mathrm{dom}\, r? \subseteq \mathrm{dom}\, background$

$l! \notin \mathrm{ran}\, layers$

$layers' = layers \frown \langle l! \rangle$

$windows' = windows \cup \{l! \mapsto r?\}$

A layer may also be de-allocated. The associated process should also be freed for safety, but this is a separate operation.

DelLayer

$\Phi Layer$
$l? : Layer$

$l? \in \mathrm{ran}\, layers$

$layers' = layers \setminus \{l?\}$

$windows' = \{l?\} \lhd windows$

13.2.2 New processes

A new process can be allocated. A handle on the process is returned. Note that the associated layer is undefined. The process program is often the default terminal emulation program and in practice this is specified using a null argument.

NewProc

$\Phi Proc$
$f? : Program$
$id! : Id$
$proc : Proc$

$proc.prog = f?$

$procs' = procs \cup \{id! \mapsto proc\}$

$receiver' = receiver$

A process may be created using the standard user interface to select the rectangle for the process's layer. This associates the process with the layer.

$NewWindow_0 \mathrel{\widehat=} NewProc \fatsemi NewLayer$

$NewWindow_{Blit} \mathrel{\widehat=} [\, NewWindow_0 \mid l! = proc.l \,]$

A layer may be selected so that the process in that layer becomes the receiver of mouse and keyboard events.

ToLayer

$\Phi Proc$
$l? : Layer$
$proc : Proc$

$proc \in \operatorname{ran} procs$

$l? = proc.l$

$procs' = procs$

$receiver' = procs^{\sim}\ proc$

The process whose layer is indicated by the mouse may be returned.

GetProc

$\Xi Blit$
$l? : Layer$
$proc! : Proc$

$proc! \in \operatorname{ran} procs$

$proc!.l = l?$

Alternatively, a handle on all the processes in the system may be returned.

GetProcTab

$\Xi Blit$
$procs! : Id \pfun Proc$

$procs! = procs$

13.2.3 Mouse operations

The Blit includes a mouse. This controls the position of a cursor on the screen. Additionally, any combination of the three buttons on the mouse may be pressed at any time.

$Buttons ::= Button1 \mid Button2 \mid Button3$

Mouse

$xy : Point$
$buttons : \mathbb{P}\, Buttons$

Some pixel positions on the display screen may be associated with a particular layer.

This is the layer which is visible at that particular pixel. Otherwise the background is visible at that point. The gun-sight cursor is used to find a particular layer.

Gunsight

$$\Xi Blit$$
$$pos? : Mouse$$
$$l! : Layer$$

$$pos?.xy \in \mathrm{dom}(\oplus / rects) \Rightarrow$$
$$\quad l! = layers(max\{n : \mathrm{dom}\, rects \mid pos?.xy \in \mathrm{dom}(rects\, n)\})$$

$$pos?.xy \notin \mathrm{dom}(\oplus / rects) \Rightarrow$$
$$\quad l! = NullLayer$$

We can now give a more complete definition for *GetProc*.

$$GetProc_1 \mathrel{\widehat=} Gunsight \gg GetProc$$

The box cursor is used to pick out a rectangular area. This is done by sweeping out a rectangle while button 3 is depressed.

Box

$$\Xi Blit$$
$$pos1?, pos2? : Mouse$$
$$r! : Rectangle$$

$$pos1?.buttons = \{Button3\}$$

$$pos2?.buttons = \emptyset$$

$$\mathrm{dom}\, r! = (\mathrm{dom}\, background) \cap$$
$$\qquad ((pos1?.xy \mathbin{\underline{min}} pos2?.xy)..(pos1?.xy \mathbin{\underline{max}} pos2?.xy))$$

13.2.4 The 'mux' multiplexer

The underlying library routines available for the Blit do not include all the basic operations necessary for a complete window manager. However a program called *mux* may be downloaded from the host system. This manages asynchronous windows, or layers, on the Blit terminal. Each layer is essentially a separate terminal. Layers are created, deleted, and rearranged using the mouse. Depressing mouse button 3 activates a menu of layer operations and releasing the button selects an operation. Some of these operations are covered here.

A new layer containing a terminal emulator process may be created by sweeping out a rectangle with the mouse while button 3 is depressed.

$$New \mathrel{\widehat=} Box \gg NewWindow_{Blit}$$

The size and location of a layer on the screen may be changed. A gun-sight cursor to select the layer and a box cursor to select the new position are presented to the user. The domain of the layer's rectangular area is updated.

$$Reshape \mathrel{\widehat=} (Gunsight \gg DelLayer) \fatsemi (Box \gg NewLayer)$$

A non-current layer may be selected using button 1. The layer is pulled to the front of the screen and made the current layer for keyboard and mouse input.

Top

$$\Phi Layer$$
$$l? : Layer$$

$$layers' = layers \underline{raise} \{l?\}$$
$$windows' = windows$$

$$Current \& Top \mathrel{\widehat=} Gunsight \gg (ToLayer \fatsemi Top)$$

13.3 Errors

Successful operations can be reported.

Success$_{Blit}$

$$\Delta Blit$$
$$l! : Layer$$

$$l! \neq NullLayer$$

Similarly, failures can also be reported. This could be because there is not enough memory for example.

Failure

$$\Xi Blit$$
$$l! : Layer$$

$$l! = NullLayer$$

A rectangle not within the background area could be given in error.

InvalidRect

$$\Xi Blit$$
$$r? : Rectangle$$

$$\neg\, (\mathrm{dom}\, r? \subseteq \mathrm{dom}\, background)$$

For example, the operations to create a new layer or process may fail because of an invalid rectangle or lack of memory.

$$NewLayerone_1 \mathrel{\widehat=} (NewLayer \land Success_{Blit}) \lor (InvalidRect \land Failure) \lor Failure$$

$$NewProcone_1 \mathrel{\widehat=} (NewProc \land Success_{Blit}) \lor Failure$$

$$NewWindow_{Blit1} \mathrel{\widehat=} (NewWindow_{Blit} \land Success_{Blit}) \lor Failure$$

Sometimes an invalid layer may be specified as input.

InvalidLayer

$\Xi Blit$

$l? : Layer$

$l? \notin \operatorname{ran} layers$

A layer must exist to delete it or make it the current receiver.

$DelLayer_1 \mathrel{\widehat=} DelLayer \lor InvalidLayer$

$ToLayer_1 \mathrel{\widehat=} ToLayer \lor InvalidLayer$

Some operations return no errors.

$GetProcTab_1 \mathrel{\widehat=} GetProcTab$

13.4 Simplifications, Assumptions and Comments

Many cursor and mouse operations and other graphics operations have been ignored for brevity.

The documentation [16] states that the associated process must be freed when a layer is de-allocated. However it does not make it clear how to do this so this has not been specified.

The Blit is different from most other window systems in that it is a diskless intelligent terminal which interacts with a remote host in normal operation. In addition there is no protection between processes and layers within the Blit. Hence care must be exercised when programming it, but in return this allows greater flexibility and versatility.

While the Blit was an interesting research exercise, it never achieved widespread use. In the next chapter, we consider another window system that has come to dominate the UNIX-based workstation market, namely X.

Chapter 14

The X Window System

X [357, 358, 359] is a network transparent windowing system developed at the Massachusetts Institute of Technology (MIT) and designed to run under UNIX. *The X display server distributes user input to, and accepts output requests from various client programs either on the same machine or over a network. This chapter documents (an early version of) some of the C library calls available to X users [360]. Some of the graphical operations available under X have been documented elsewhere [144]. X has subsequently become an important window system in the* UNIX *workstation market, and is being developed further by the X Consortium, an independent not-for-profit company formed in 1993 as a successor to the MIT X Consortium.*

14.1 System State

The state of the X system is introduced in simple stages in order to build up the concepts involved. This is done by redefining a state schema called X in terms of itself and a series of manageably sized state definition fragments.

All the windows in an X server are arranged in a strict hierarchy. At the top of the hierarchy is the 'root' window. Each window has a parent except the root window. Child windows may in turn have their own children. Each window, including the root window, may be considered to consist of a pixel map in this simple description.

$$
\begin{array}{|l}
\hline
X_0 \\
root : Window \\
children : \mathbb{P}\, Window \\
parents : Window \pfun Window \\
subwindows : Window \rel Window \\
windows : Window \pfun Pixmap \\
\hline
root \notin children \\
children = \mathrm{dom}\, parents \\
subwindows = parents^{\sim} \\
\mathrm{dom}\, windows = children \cup \{root\} \\
\hline
\end{array}
$$

Subwindows are displayed in a particular order within their parent window. This may

be modelled as a sequence of uniquely identified windows in ascending order of display. These consist of all the child windows.

$$
\begin{array}{|l}
\hline
X_1 \\
X_0 \\
order : Window \pfun \operatorname{iseq} Window \\
\hline
\operatorname{dom} order = \operatorname{dom} windows \\
(\forall w_1, w_2 : \operatorname{dom} order \mid w_1 \neq w_2 \bullet \\
\quad \operatorname{ran}(order\ w_1) \cap \operatorname{ran}(order\ w_2) = \emptyset) \\
\bigcup \{w : \operatorname{dom} order \bullet \operatorname{ran}(order\ w)\} = children \\
(\forall w : \operatorname{dom} windows \bullet \operatorname{ran}(order\ w) = subwindows(\!| \{w\} |\!)) \\
\hline
\end{array}
$$

Windows must be 'mapped' before they can be displayed. The root window is always mapped. All of a window's ancestors (if any) must also be mapped for it to be viewable on the display. Unviewable windows are mapped but have some ancestor which is unmapped.

$$
\begin{array}{|l}
\hline
X_2 \\
X_1 \\
mapped, viewable, unviewable : \mathbb{P}\, Window \\
\hline
mapped \subseteq \operatorname{dom} windows \\
root \in mapped \\
viewable = \{c : children \mid parents^*(\!| \{c\} |\!) \subseteq mapped\} \cup \{root\} \\
unviewable = \{c : children \mid c \in (mapped \setminus viewable)\} \\
\hline
\end{array}
$$

Each viewable window has an associated visible pixel map which consists of the pixel map of the window overlaid with its subwindows (in order) if any. These are 'clipped' to the size of the parent window.

The root window covers the entire background of the display screen. The screen displays the pixel map visible from the root window.

$$
\begin{array}{|l}
\hline
X \\
X_2 \\
visible : Window \pfun Pixmap \\
screen, background : Pixmap \\
\hline
\operatorname{dom} visible = viewable \\
(\forall w : viewable \bullet \\
\quad visible\ w = (windows\ w) \oplus (\operatorname{dom}(windows\ w) \lhd \\
\qquad\qquad \oplus / (squash((order\ w) \fatsemi visible)))) \\
background = windows\ root \\
screen = visible\ root \\
\hline
\end{array}
$$

Initially there are no children and only the root window is mapped. Hence only the background is displayed.

InitX

$$X'$$

$$windows' = \{root' \mapsto background'\}$$

$$order' = \{root' \mapsto \langle\rangle\}$$

$$mapped' = \{root'\}$$

Thus there are no child windows:

$$InitX \vdash children' = \emptyset$$

Proof:

$$\begin{array}{l} InitX \\ \{\text{predicate in } InitX\} \\ \Rightarrow windows' = \{root' \mapsto background'\} \\ \{\text{law of `dom'}\} \\ \Rightarrow \mathrm{dom}\, windows' = \{root'\} \\ \{\text{predicate in } X'\} \\ \Rightarrow children' \cup \{root'\} = \{root'\} \\ \{\text{since } root' \notin children' \text{ from } X'\} \\ \Rightarrow children' = \emptyset \end{array}$$

Such reasoning is normally only done informally in a designer's head; confirming such properties serves to increase the confidence that a specification does actually describe what is wanted.

Consider changes in the window system. The root window identifier and the background of the screen do not change.

ΔX

$$X$$
$$X'$$

$$root' = root$$

$$background' = background$$

Sometimes the state of the system is unaffected during an operation.

$$\Xi X \mathrel{\widehat{=}} [\, \Delta X \mid \theta X' = \theta X \,]$$

We can now consider operations on the state of the system; initially, error-free operations will be presented for simplicity. Error conditions are covered later.

14.2 Window Operations

14.2.1 Creating and destroying windows

Firstly, we wish to be able to create windows. For these operations we have to supply the parent window under which the new window is to reside in the window hierarchy. The position, size and background of the window must also be specified. Here these are defined by '*bgnd*?' for simplicity. Note that the created window will not actually be displayed until it is 'mapped' (see later).

CreateWindow

ΔX
$parent? : Window$
$bgnd? : Pixmap$
$w! : Window$

$parent? \in \mathrm{dom}\, windows$

$w! \notin \mathrm{dom}\, windows$

$windows' = windows \cup \{w! \mapsto bgnd?\}$

$order' = order \oplus \{parent? \mapsto (order\ parent?) \frown \langle w! \rangle\}$

$mapped' = mapped$

Note that the predicates in the schema above fully define the state after the operation since all the other state components may be derived from those given above. The other components are included in the state definition to allow us to have different views of the system, depending on the manner in which we wish to access the state.

Sometimes it is convenient to create several windows at once under a single parent window. Note that not all the windows requested may be created, but this is indicated by the information returned. This consists of a partial injection obtained from the sequence numbers of the windows which are actually created to the window identifiers which they are allocated.

CreateWindows

ΔX
$parent? : Window$
$defs? : \mathrm{seq}\, Pixmap$
$defs! : \mathbb{N} \rightarrowtail Window$

$parent? \in \mathrm{dom}\, windows$

$\mathrm{dom}\, defs! \subseteq \mathrm{dom}\, defs?$

$\mathrm{ran}\, defs! \cap \mathrm{dom}\, windows = \emptyset$

$windows' = windows \cup (defs!^{\sim} \fatsemi defs?)$

$order' = order \oplus \{parent? \mapsto (order\ parent?) \frown (squash\ defs!)\}$

$mapped' = mapped$

We also wish to destroy windows. Given a particular window, we may wish to destroy a set of windows which are associated with it. We can define a partial specification to do this as a schema. Exactly which windows are to be destroyed is not specified for the present.

$\Phi Destroy$

ΔX

$w? : Window$

$destroy : \mathbb{P}\, Window$

$w? \in children$

$windows' = destroy \lhd windows$

$(\forall w : \mathrm{dom}\, windows \bullet order'\ w = (order\ w)\ \underline{remove}\ destroy)$

$mapped' = mapped \setminus destroy$

We may wish all subwindows, as well as the window itself, to be destroyed.

$$DestroyWindow \mathrel{\widehat=} [\ \Phi Destroy \mid destroy = subwindows^*(\!|\ \{w?\}\ |\!)\ ]$$

Alternatively, we may wish to just destroy the subwindows under the specified window.

$$DestroySubwindows \mathrel{\widehat=} [\ \Phi Destroy \mid destroy = subwindows^+(\!|\ \{w?\}\ |\!)\ ]$$

Note that the 'root' background window cannot be destroyed using these operations. Only child windows may be destroyed.

$$DestroyWindow \vdash root' \in \mathrm{dom}\, windows'$$

Proof: directly from the invariant $\mathrm{dom}\, windows = children \cup \{root\}$ in the schema X_0. However, we may wish to investigate this further, to see what preconditions are introduced by such a constraint.

14.2.2 Manipulating windows

A window, and all its ancestors, must be 'mapped' to be visible on the screen. However a mapped window may still be invisible if it is obscured by a sibling window.

Mapping operations require a child window to be specified. The hierarchical relationships between windows and the contents of the windows are left unaffected.

ΦMap

ΔX

$w? : Window$

$w? \in children$

$windows' = windows$

Mapping a window raises the window and all its subwindows which have had map requests. Mapping a window which is already mapped has no effect on the screen – it does *not* raise it.

MapWindow

$$\Phi Map$$
$$parent : Window$$

$$parent = parents\ w?$$
$$w? \notin mapped \Rightarrow order' = order \oplus \{parent \mapsto ((order\ parent)\ \underline{raise}\ \{w?\})\}$$
$$w? \in mapped \Rightarrow order' = order$$
$$mapped' = mapped \cup \{w?\}$$

All the unmapped subwindows of a given window can be mapped together. The order in which they are mapped is chosen by the system rather than the caller.

MapSubwindows

$$\Phi Map$$
$$neworder : \mathrm{iseq}\, Window$$
$$newmapped : \mathbb{P}\, Window$$

$$newmapped = subwindows(\!|\ \{w?\}\ |\!) \setminus mapped$$
$$\mathrm{ran}\, neworder = newmapped$$
$$order' = order \oplus \{w? \mapsto ((order\ w?)\ \underline{remove}\ newmapped) \frown neworder\}$$
$$mapped' = mapped \cup newmapped$$

A window can be unmapped. The window will disappear from view if it was visible.

UnmapWindow

$$\Phi Map$$

$$order' = order$$
$$mapped' = mapped \setminus \{w?\}$$

All subwindows of a specified window can be unmapped.

UnmapSubwindows

$$\Phi Map$$

$$order' = order$$
$$mapped' = mapped \setminus subwindows(\!|\ \{w?\}\ |\!)$$

Windows may be manipulated in various ways. Given a window, its pixel map may be updated. It is also raised to the top of the display. We can define a general schema to simplify the definition of such operations.

ΦWindow

ΔX
$w? : Window$
$map : Pixmap$
$parent : Window$

$parent = parents\ w?$

$windows' = windows \oplus \{w? \mapsto map\}$

$order' = order \oplus \{parent \mapsto ((order\ parent)\ \underline{raise}\ \{w?\})\}$

$mapped' = mapped$

(Note that $parent = parents\ w?$ implies that $w? \in children$.)

A window may be moved and raised without changing its size. Moving a mapped window may or may not lose its contents, depending on various circumstances.

MoveWindow

$\Phi Window$
$xy? : Pixel$

$\mathrm{dom}\ map = \mathrm{dom}((offset\ xy?) \fatsemi (windows\ w?))$

The size of a window may be changed without changing its upper left coordinate. A new width and height are given. The window is always raised. Changing the size of a mapped window loses its contents.

ChangeWindow

$\Phi Window$
$wdht? : Pixel$
$pix_1, pix_2 : Pixel$

$\mathrm{dom}(windows\ w?) = pix_1 . . pix_2$

$\mathrm{dom}\ map = pix_1 . . (pix_1 + wdht?)$

The size and location of a window may be configured together by combining the last two operations. The window is raised and the contents are lost.

$$ConfigureWindow \mathrel{\widehat=} (MoveWindow \upharpoonright \Delta X) \fatsemi (ChangeWindow \upharpoonright \Delta X)$$

Some operations explicitly affect the order in which the windows are displayed. A child window is specified, and window relationships, the windows themselves, and the set of mapped windows remain unchanged.

$$
\begin{array}{|l}
\hline
\Phi Order \\
\Delta X \\
w? : Window \\
parent : Window \\
suborder, suborder' : \operatorname{seq} Window \\
\hline
parent = parents\ w? \\
windows' = windows \\
suborder = order\ parent \\
order' = order \oplus \{parent \mapsto suborder'\} \\
mapped' = mapped \\
\hline
\end{array}
$$

(Note that as for the $\Phi Window$ schema, $parent = parents\ w?$ implies that $w? \in children$.)

A window may be 'raised' so that no sibling window obscures it. If the windows are regarded as overlapping sheets of paper stacked on a desk, then raising a window is analogous to moving the sheet to the top of the stack, while leaving its position on the desk the same.

$$RaiseWindow \mathrel{\widehat=} [\ \Phi Order \mid suborder' = suborder\ \underline{raise}\ \{w?\}\ ]$$

A window may also be 'lowered' in a complementary fashion. If the windows are regarded as overlapping sheets of paper stacked on a desk, then lowering a window is analogous to moving the sheet to the bottom of the stack, while leaving its position on the desk the same.

$$LowerWindow \mathrel{\widehat=} [\ \Phi Order \mid suborder' = suborder\ \underline{lower}\ \{w?\}\ ]$$

Overlapping mapped subwindows of a particular window may be raised or lowered in a circular manner. The set of these windows is identified. If it is non-empty, the ordering of the window's children is updated; otherwise it is left unchanged.

$$
\begin{array}{|l}
\hline
\Phi Circ \\
\Delta X \\
w? : Window \\
submapped, circ : \mathbb{P}\, Window \\
suborder, suborder' : \mathrm{seq}\, Window \\
\hline
w? \in children \\
submapped = subwindows(\!| \{w?\} |\!) \cap mapped \\
circ = \{w : submapped \mid \\
\quad (\exists w_2 : submapped \bullet w_2 \neq w \land (visible\ w_2)\ \underline{overlaps}\ (visible\ w))\} \\
windows' = windows \\
suborder = order\ w? \\
circ \neq \emptyset \Rightarrow order' = order \oplus \{w? \mapsto suborder'\} \\
circ = \emptyset \Rightarrow order' = order \\
mapped' = mapped \\
\hline
\end{array}
$$

For a particular window, the lowest mapped child that is partially obscured by another child may be raised. Repeated executions lead to round robin raising.

$$
\begin{array}{|l}
\hline
CircWindowUp \\
\Phi Circ \\
\hline
circ \neq \emptyset \Rightarrow \\
\quad suborder' = suborder\ \underline{raise}\ \{suborder(min(\mathrm{dom}(suborder \upharpoonright circ)))\} \\
\hline
\end{array}
$$

Similarly, the highest mapped child of a particular window that (partially) obscures another child may be lowered. Repeated executions lead to round robin lowering.

$$
\begin{array}{|l}
\hline
CircWindowDown \\
\Phi Circ \\
\hline
circ \neq \emptyset \Rightarrow \\
\quad suborder' = suborder\ \underline{lower} \\
\qquad\qquad \{suborder(max(\mathrm{dom}(suborder \upharpoonright circ)))\} \\
\hline
\end{array}
$$

14.2.3 Other operations

We can ask for information about a particular window. As well as the size, position, etc., of the window, details about the mapped state of the window are returned. '*IsUnmapped*' indicates that the window is unmapped, '*IsMapped*' indicates that it is mapped and displayed (i.e., all of its ancestors are also mapped), and '*IsInvisible*' implies that it is mapped but some ancestor is not mapped.

$$MappedState ::= IsUnmapped \mid IsMapped \mid IsInvisible$$

QueryWindow

$$
\begin{array}{l}
\Xi X \\
w? : Window \\
info! : Pixmap \\
mapped! : MappedState \\
\hline
w? \in children \\
info! = windows\ w? \\
w? \notin mapped \Rightarrow \\
\quad mapped! = IsUnmapped \\
parents^{*}(\!|\ \{w?\}\ |\!) \subseteq mapped \Rightarrow \\
\quad mapped! = IsMapped \\
w? \in mapped \land \neg\ (parents^{+}(\!|\ \{w?\}\ |\!) \subseteq mapped) \Rightarrow \\
\quad mapped! = IsInvisible
\end{array}
$$

We can also find out the window identifiers of the parent and all the children (and hence the number of children) for a particular window. The children are listed in current stacking order, from bottommost (first) to topmost (last).

QueryTree

$$
\begin{array}{l}
\Xi X \\
w? : Window \\
parent! : Window \\
children! : \mathrm{seq}\ Window \\
\hline
parent! = parents\ w? \\
children! = order\ w?
\end{array}
$$

The X system includes many other operations. These include more detailed window operations, mouse operations, graphics for line drawing and fill operations, screen raster operations, moving bits and pixels to and from the screen, storing and freeing bit maps and pixel maps, cursor definition, colour operations, font manipulation routines, text output to a window, and so on. However the operations covered give an indication of the basic windowing facilities available under the X system.

14.3 Errors

Many operations return a status report signalling success or failure of the operation. Let this be denoted '*Status*'. Often a '*NULL*' status indicates success and a non-*NULL* status indicates failure.

$$[Status]$$

$$NULL : Status$$

The operations covered so far detail what should happen in the event of no errors. In this case we also wish to report the fact that the operation was successful.

$Success_X$

$status! : Status$

$status! = NULL$

If errors do occur, then these need to be reported as well. For example, an invalid parent window may be specified.

InvalidParent

ΞX

$parent? : Window$

$status! : Status$

$parent? \notin \mathrm{dom}\, windows$

$status! \neq NULL$

Alternatively, an invalid child window could be given as input.

$InvalidWindow_X$

ΞX

$w? : Window$

$status! : Status$

$w? \notin children$

$status! \neq NULL$

We may include these errors with the previously defined operations which ignored error conditions, to produce total operations.

$$CreateWindow_1 \mathrel{\widehat=} (CreateWindow \land Success_X) \lor InvalidParent$$

All the other operations covered take the following form:

$$DestroyWindow_1 \mathrel{\widehat=} (DestroyWindow \land Success_X) \lor InvalidWindow_X$$

14.4 Simplifications and Assumptions

In the description given, only 'opaque' windows have been considered. The actual X system includes 'transparent' windows, mainly used for menus, and 'icon' windows which may be associated with opaque windows, but these have been ignored in this description for simplicity. These could be included in the state of the system. The operation specifications would need to be updated appropriately.

X_{ext}

X

$transparent, opaque : \mathbb{P}\, Window$

$icon : Window \rightarrowtail\!\!\!\!\!\!\rightarrow Window$

$\langle transparent, opaque, \mathrm{ran}\, icon \rangle$ partition $children$

$\mathrm{dom}\, icon \subseteq opaque$

Windows have other information associated with them besides their pixel maps and their mapping status, such as border information. However this is not covered here. Exposure events that result from window operations are also ignored.

The informal description used to formulate this specification was not completely clear on a number of points. For example, the exact ordering of windows and their subwindows is not made explicitly clear after operations which affect this. In particular, it has been assumed here that raising and lowering a window implies that all its subwindows are also raised or lowered. Where necessary, an educated guess has been made as to the behaviour of the system.

14.5 Comments and Inconsistencies

The X Window System is relatively complicated. It includes a number of basic concepts, several of which could not be included here fully because of lack of space. The hierarchical structure makes it very versatile.

Perhaps surprisingly, X has no notion of a 'current' window. Hence a large number of the library routines need a window identifier as input (including all those covered here). This is rather cumbersome and could introduce some unnecessary overhead in application programs using the system. However this is advantageous if a number of windows are being updated simultaneously since then there are effectively several current windows.

An earlier version of this specification was sent to MIT with annotations, raising questions about areas which were not well understood from the original documentation. A number of inconsistencies in the formal specification (compared to the implementation of X) were discovered from the feedback obtained. The major errors were as follows:

- Children are always on top of their parent, and the hierarchies of two siblings never interleave. In the original specification, an overall order ($order : \mathrm{seq}\, Window$) was included as part of the state; it did not preclude the above. Here the ordering is defined on a per window basis, for just the immediate children.
- The contents of unmapped and invisible parts of windows are lost. For example, in the schema ΦMap, the predicate '$windows' = windows$' is actually incorrect since the contents of the window $w?$ will be lost if it is unmapped. However the specification has not been changed in this respect since exposure events are ignored here, and these would typically restore the contents of re-exposed windows. If exposure events were added to this specification then this should be changed.

These points were missed from the original documentation. They would probably have been discovered if an implementation of X had been available for 'testing' purposes. The documentation could be improved in these areas to avoid misunderstanding.

This concludes the descriptions of window systems in Z included in this book. The next chapter compares the WM, Blit and X window systems, based on the experience of formalizing each of them.

Chapter 15

Formal Specification of Existing Systems

A high level description of three existing window systems has been presented in chapters 12, 13 and 14. Only a few operations for each system have been covered. A complete description would require a manual for each of the systems; a formal specification does not necessarily reduce the size of a description using informal methods. However it does make it much more precise. Because of this, it is possible to reason about a system and detect inconsistencies in it far more easily than the case where only an informal specification is available. Even if formal specification is not used in the final documentation, its use will clarify points which can then be described informally to the user.

15.1 Comparison of Window Systems

Of the three window systems investigated in chapters 12 to 14, X provides the most comprehensive features. *WM* is a much simpler system with no overlapping windows or hierarchical structure. However it does automatically adjust the size of windows when necessary. The Blit is a 'raw' machine onto which window management functions can be loaded if desired. The following table gives a comparison of the features available on each system.

Window system	Overlapping windows	Hierarchical structure	Automatic sizing	Current window
WM	$\times$	$\times$	$\surd$	$\surd$
Blit	$\surd$	$\times$	$\times$	$\surd$
X	$\surd$	$\surd$	$\times$	$\times$

The X window system was originally investigated first. It turned out to be the most complicated system and took a significant amount of time to formalize. Subsequently, the specification of *WM* and the Blit system were comparatively easy.

Since the original specification, Version 11 of X (or X11 as it is normally known) has become an industry standard and is available on many workstations. The other two systems are not so widely used. X now includes a library interface built on top of the main X interface that implements almost all of *WM*. Hence most *WM* applications will run under X without source modification.

15.2 Case Study Experience

The specifications of the window systems presented here were originally undertaken as part of the Distributed Computing Software (DCS) Project at the Programming Research Group within the Oxford University Computing Laboratory [43, 47]. As well as designing and documenting network services using a formal notation, part of the brief of the DCS project was to undertake case studies of existing systems and to formally specify parts of them in Z to gain a greater understanding of their operation.

Originally it had been hoped to compare parts of a number of distributed systems using Z. However, the authors of potential systems for investigation could only supply academic papers (not enough information) or the source code (too much information). What was required was some form of informal documentation for the system. Because window systems are used directly by users, there seems to be more readable documentation for such systems.

In each case, omissions and ambiguities in the documentation were discovered by attempting to formalize the system. Where necessary, intelligent guesses were made about the actual operation. These were usually correct, but not always.

Subsequently, the formal specifications could be used to update the existing documentation, or even rewrite it from scratch. Although Z has been developed as a design tool, it is also well suited for *post hoc* specifications of existing systems, and for detecting and correcting errors and anomalies in the documentation of such systems [41].

The most important stage of formalizing a system is selecting the right level of abstraction for modelling its state. This is normally an iterative process. On attempting to specify an operation one often needs to backtrack to change the abstract state of the system. In particular, extra state components can be convenient to provide different views of the system depending on the operation involved.

There are likely to be some inconsistencies between the specifications given in chapters 12 to 14 and the actual operation of the systems described. This is due to impreciseness and misunderstanding of the informal documentation used to formulate these specifications. This illustrates one of the reasons for using formal specification techniques – to avoid ambiguity or vagueness and to aid precise communication of ideas. Because of this, formal notation forces issues to the surface much earlier in the design process than when using informal description techniques such as natural language and diagrams. Difficult areas of the design cannot be ignored until the implementation stage. This reduces the number of backtracks necessary round the design/implementation cycle.

Additionally, using formal specification techniques should reduce maintenance costs since more of the errors in a system will be discovered before it is released into the field. Although specification and design costs will be increased, implementation and maintenance costs should be lower, reducing overall costs.

Formally specifying an existing system could be particularly useful if it is to be re-engineered to comply with modern software engineering standards. In such cases there could be costs benefits by taking such an approach [319].

15.3 General Conclusions

The Z notation can be used to succinctly specify real systems. The examples given here and other case studies undertaken in academia and industry lend support to this assertion.

Z may be used to produce readable specifications. It has been designed to be read by humans rather than computers. Thus it can form the basis for documentation.

Large specifications are manageable in Z, using the schema notation for structuring. It is possible to produce hierarchical specifications. A part of a system may be specified in isolation, and then this may be put into a global context.

The appendices provide further general information on the Z notation. Appendix A gives pointers to further sources of information on Z, especially available on-line. A glossary of Z notation is given in Appendix B. A comprehensive literature guide in included in Appendix C, together with a substantial bibliography and index.

Acknowledgements

The work for the book was largely supported by the UK Science and Engineering Research Council (SERC) and its successor, the Engineering and Physical Sciences Research Council (EPSRC). The author was funded by the SERC on the Distributed Computing Software project (grant number GR/C/9786.6) from 1985 to 1987, on the Software Engineering project (on an SERC rolling grant) from 1987 to 1989, on the collaborative Information Engineering Directorate safemos project (GR/F36491) from 1989 to 1993 and is currently funded on EPSRC grant number GR/J15186 investigating *Provably Correct Hardware/Software Co-design*.

Many people have provided inspiration for and have contributed further over the years to the material presented here. These include: Rosalind Barden, Peter Breuer, Stephen Brien, David Brown, Jim Davies, Neil Dunlop, Jim Farr, Roger Gimson, Tim Gleeson, Mike Gordon, Anthony Hall, Ian Hayes, He Jifeng, Mike Hinchey, Tony Hoare, Darrel Ince, Steve King, Kevin Lano, Carroll Morgan, John Nicholls, Paritosh Pandya, Philip Rose, Jeff Sanders, Jane Sinclair, Ib Sørensen, Mike Spivey, Victoria Stavridou, Susan Stepney, Bernard Sufrin, Stig Topp-Jørgensen and Jim Woodcock. Please forgive me if I have inadvertently omitted anyone; I am extremely grateful to the above for providing such a rich environment at Oxford and elsewhere.

Mike Gordon helped in particular with the section on schemas in Chapter 3. Carroll Morgan and Roger Gimson were the original team on the Distributed Computing Software Project, as reported in Chapter 4. David Brown designed the example in Chapter 7. Jim Farr undertook an MSc. project under my supervision, on which the specification of the Transputer instruction set presented in Chapter 9 is partially based. Rob Pike (AT&T), Jim Gettys (DEC), C. Neuwirth (CMU), Dave Presotto (AT&T), David Rosenthal (CMU), Mahadev Satyanarayanan (CMU), Bob Scheifler (MIT) and Bill Weihl (MIT) helped provide material, information and access to the window systems described in Part VI. Susan Stepney and Rosalind Barden of Logica Cambridge Limited initiated the literature guide in Appendix C as part of the collaborative ZIP project. Jane Sinclair read the manuscript as it neared completion and gave some valuable comments.

The book has been formatted using Leslie Lamport's LaTeX document preparation system [251]. The Z notation has been formatted and type-checked throughout, using Mike Spivey's *f*UZZ package [380].

Finally, thank you to my family, Jane, Alice and Emma, for being so patient and providing a wonderful environment at home.

Appendices

Appendix A gives some general pointers for further information on Z, especially that which is available on-line. A glossary of the Z notation is provided in Appendix B. Finally, Appendix C surveys the generally available literature on Z, in conjunction with a comprehensive bibliography.

Appendix A

Information on Z

This appendix provides some details on how to access information on Z, particularly electronically. It has been adapted from a message that is updated and sent out monthly on international computer networks.

A more recent version of this information is available on-line on the following World Wide Web (WWW) hypertext page where it is split into convenient sections:

`http://www.faqs.org/faqs/z-faq/`

A.1 Electronic Newsgroup

The `comp.specification.z` electronic USENET newsgroup was established in June 1991 and is intended to handle messages concerned with the formal specification notation Z (pronounced "zed"). It has an estimated readership of around 30,000 people worldwide. Z, based on set theory and first order predicate logic, has been developed by members of the Programming Research Group at the Oxford University Computing Laboratory (OUCL) and elsewhere since the late 1970s. It is now used by industry as part of the software (and hardware) development process in both the UK and the US. It is undergoing international ISO standardization. `Comp.specification.z` provides a convenient forum for messages concerned with recent developments and the use of Z. Pointers to and reviews of recent books and articles are particularly encouraged. These will be included in the Z bibliography (see Section A.8) if they appear in `comp.specification.z`.

A.2 Electronic Mailing List

There is an associated Z FORUM electronic mailing list that was initiated in January 1986 by Ruaridh Macdonald, RSRE (now the Defence Research Agency, DRA), UK. Articles are now automatically cross-posted between `comp.specification.z` and the mailing list for those whose do not have access to USENET news. This may apply especially to industrial Z users who are particularly encouraged to subscribe and post their experiences, comments and requests to the list. Please contact the email address `zforum-request@comlab.ox.ac.uk` with your name, address and email address to join the mailing list (or if you change your email address or wish to be removed from the list).

Readers are strongly urged to read the `comp.specification.z` newsgroup rather than the Z FORUM mailing list if possible. Messages for submission to the Z FORUM mailing list and the `comp.specification.z` newsgroup may be emailed to `zforum@comlab.ox.ac.uk`. This method of posting is particularly recommended for important messages like announcements of meetings since not all messages posted on `comp.specification.z` reach the OUCL.

A mailing list for the Z User Meeting educational issues session has been set by Neville Dean, Anglia Polytechnic University, UK. Anyone interested may join by emailing `zugeis-request@comlab.ox.ac.uk` with your contact details.

A.3 Postal Mailing List

If you wish to join the postal Z mailing list, please send your address to Amanda Kingscote, Praxis plc, 20 Manvers Street, Bath BA1 1PX, UK (tel +44-1225-444700, fax +44-1225-465205, email `ark@praxis.co.uk`). This will ensure you receive details of Z meetings, etc., particularly for people without access to electronic mail.

A.4 Subscribing to the Newsgroup and Mailing List

If you use Z, you are welcome to introduce yourself to the newsgroup and Z FORUM list by describing your work with Z or raising any questions you might have about Z which are not answered here. You may also advertize publications concerning Z which you or your colleagues produce. These may then be added to the master Z bibliography maintained at the OUCL (see Section A.8).

A.5 Electronic Z Archive

Information on the World Wide Web (WWW) is available under the following page:

```
http://www.zuser.org/z/
```

See also the following page on formal methods in general:

```
http://www.afm.sbu.ac.uk/
```

The WWW global hypertext system is accessible using the *netscape*, *mosaic* or *lynx* programs for example. Contact your system manager if WWW access is not available on your system.

Some of the archive is also available via anonymous FTP on the Internet under the `ftp://ftp.comlab.ox.ac.uk/pub/Zforum` directory. Type the command '`ftp ftp.comlab.ox.ac.uk`' (or alternatively '`ftp 163.1.27.2`' if this does not work) and use '`anonymous`' as the login id and your email address as the password when prompted. The FTP command '`cd pub/Zforum`' will get you into the Z archive directory.

In `ftp://ftp.comlab.ox.ac.uk/pub/Zforum/README` there is some general information and `00index` in the same directory gives a list of the files. Retrieve these using the FTP command '`get README`', for example.

There is an automatic electronic mail-based electronic archive server which allows access to some of the archive such as most messages on `comp.specification.z`

and Z FORUM, as well as a selection of other Z-related text files. Send an email message containing the command '`help`' to `archive-server@comlab.ox.ac.uk` for further information on how to use the server. A command of '`index z`' will list the Z-related files. If you have serious trouble accessing the archive server, please contact the address `archive-management@comlab.ox.ac.uk`.

A.6 Z Tools

Various tools for formatting, type-checking and aiding proofs in Z are available. A free LaTeX style file and documentation can be obtained from the OUCL archive server. Access `ftp://ftp.comlab.ox.ac.uk/pub/Zforum/zed.sty` and the `zguide.tex` file in the same directory via anonymous FTP. The newer styles '`csp_zed.sty`' which uses the new font selection scheme and '`zed-csp.sty`' which supports LaTeX2e handle CSP and Z symbols, and are available in the same location. A style for Object-Z '`oz.sty`' with a guide '`oz.tex`' is also accessible.

The *f*UZZ package [380], a syntax and type-checker with a LaTeX style option and fonts, is available from the Spivey Partnership, 10 Warneford Road, Oxford OX4 1LU, UK. It is compatible with the second edition of Spivey's Z Reference Manual [381]. Contact Mike Spivey (email `Mike.Spivey@comlab.oxford.ac.uk`) for further information. Alternatively send the command '`send z fuzz`' to the OUCL archive server or access `ftp://ftp.comlab.ox.ac.uk/pub/Zforum/fuzz` for brief information and an order form.

CADiℤ [237], a UNIX-based suite of tools for checking and typesetting Z specifications. CADiℤ also supports previewing and interactive investigation of specifications. It is available from York Software Engineering, University of York, York YO1 5DD, UK (tel +44-1904-433741, fax +44-1904-433744). CADiℤ supports a language like that of the Z Base Standard (Version 1.0). A particular extension allows one specification document to import another, including the mathematical toolkit as one such document. Typesetting support is available for both *troff* and for LaTeX. Browsing operations include display of information deduced by the type-checker (e.g. types of expressions and uses of variables), expansion of schemas, pre- and postcondition calculation, and simplification by the one-point rule. Work is on-going to provide support for refinement of Z specifications to Ada programs through a literate program development method and integrated proof facilities. Further information is available from David Jordan at York on `yse@minster.york.ac.uk`.

ProofPower [236] is a suite of tools supporting specification and proof in Higher Order Logic (HOL) and in Z. Short courses on ProofPower-Z are available as demand arises. Information about ProofPower can be obtained automatically from `ProofPower-server@win.icl.co.uk`. Contact Roger Jones, International Computers Ltd, Eskdale Road, Winnersh, Wokingham, Berkshire RG11 5TT, UK (tel +44-1734-693131 ext 6536, fax +44-1734-697636, email `rbj@win.icl.co.uk`) for further details.

Zola is a tool that supports the production and typesetting of Z specifications, including a type-checker and a Tactical Proof System. The tool is sold commercially and available to academic users at a special discount. For further information, contact K. Ashoo, Imperial Software Technology, 62–74 Burleigh Street, Cambridge CB1 1DJ, UK (tel +44-1223-462400, fax +44-1223-462500, email `ka@ist.co.uk`).

ZTC [444] is a Z type-checker available free of charge for educational and non-

profit uses. It is intended to be compliant with the 2nd edition of Spivey's Z Reference Manual [381]. It accepts LaTeX with `zed` or `oz` styles, and ZSL – an ASCII version of Z. ZANS is a research prototype Z animator. Both ZTC and ZANS run on Linux, SunOS 4.x, Solaris 2.x, HP-UX 9.0, DOS, and extended DOS. They are available via anonymous FTP under `ftp://ise.cs.depaul.edu/pub` in the directories `ZANS-`*x.xx* and `ZTC-`*x.xx*, where *x.xx* are version numbers. Contact Xiaoping Jia `jia@cs.depaul.edu`. for further information.

Formaliser is a syntax-directed WYSIWYG Z editor and interactive type-checker, running under Microsoft Windows, available from Logica. Contact Susan Stepney, Logica UK Limited, Cambridge Division, Betjeman House, 104 Hills Road, Cambridge CB2 1LQ, UK (email `susan@logcam.co.uk`, tel +44-1223-366343, fax +44-1223-251001) for further information.

DST-*f*UZZ is a set of tools based on the *f*UZZ package by Mike Spivey, supplying a Motif based user interface for LaTeX based pretty printing, syntax and type checking. A CASE tool interface allows basic functionality for combined application of Z together with structured specifications. The tools are integrated into SoftBench. For further information contact Hans-Martin Hoercher, DST Deutsche System-Techik GmbH, Edisonstrasse 3, D-24145 Kiel, Germany (tel +49-431-7109-478, fax +49-431-7109-503, email `hmh@informatik.uni-kiel.d400.de`).

The B-Tool can be used to check proofs concerning parts of Z specifications. The B-Toolkit is a set of integrated tools which fully supports the B-Method for formal software development and is available from B-Core (UK) Limited, Magdalen Centre, The Oxford Science Park, Oxford OX4 4GA, UK. For further details, contact Ib Sørensen (email `Ib.Sorensen@comlab.ox.ac.uk`, tel +44-1865-784520, fax +44-1865-784518).

Z fonts for Microsoft Windows and Macintosh are available on-line. For hyperlinks to these and other Z tool resources see the WWW Z page tools section:

`http://www.zuser.org/z/#tools`

A.7 Courses on Z

There are a number of courses on Z run by industry and academia. Oxford University offers industrial short courses in the use Z. As well as introductory courses, recent newly developed material includes advanced Z-based courses on proof and refinement, partly based around the B-Tool. Courses are held in Oxford, or elsewhere (e.g., on a company's premises) if there is enough demand. For further information, contact Jim Woodcock (tel +44-1865-283514, fax +44-1865-273839, email `Jim.Woodcock@comlab.ox.ac.uk`).

Logica offer a five day course on Z at company sites. Contact Rosalind Barden (email `rosalind@logcam.co.uk`, tel +44-1223-366343 ext 4860, fax +44-1223-322315) at Logica UK Limited, Betjeman House, 104 Hills Road, Cambridge CB2 1LQ, UK.

Praxis Systems plc runs a range of Z (and other formal methods) courses. For details contact Anthony Hall on +44-1225-444700 or `jah@praxis.co.uk`.

Formal Systems (Europe) Limited run a range of Z, CSP and other formal methods courses, primarily in the US and with such lecturers as Jim Woodcock and Bill Roscoe (both lecturers at the OUCL). For dates and prices contact Kate Pearson (tel +44-1865-

728460, fax +44-1865-201114) at Formal Systems (Europe) Limited, 3 Alfred Street, Oxford OX1 4EH, UK.

DST Deutsche System-Technik runs a collection of courses for either Z or CSP, mainly in Germany. These courses range from half day introductions to formal methods and Z to one week introductory or advanced courses, held either at DST, or elsewhere. For further information contact Hans-Martin Hoercher, DST Deutsche System-Techik GmbH, Edisonstrasse 3, D-24145 Kiel, Germany (tel +49-431-7109-478, fax +49-431-7109-503, email `hmh@informatik.uni-kiel.d400.de`).

A.8 Publications

A searchable on-line Z bibliography is available on the World Wide Web under

`http://www.zuser.org/z/bib.html`

in BIBTEX format. For those without WWW access, an older compressed version is available via anonymous FTP, together with a formatted compressed POSTSCRIPT version:

```
ftp://ftp.comlab.ox.ac.uk/pub/Zforum/z.bib.Z
ftp://ftp.comlab.ox.ac.uk/pub/Zforum/z.ps.Z
```

Information on Oxford University Programming Research Group (PRG) Technical Monographs and Reports, including many on Z, is available from the librarian (tel +44-1865-273837, fax +44-1865-273839, email `library@comlab.ox.ac.uk`).

Formal Methods: A Survey by S. Austin & G. I. Parkin, March 1993 [18] includes information on the use and teaching of Z in industry and academia. Contact DITC Office, Formal Methods Survey, National Laboratory, Teddington, Middlesex TW11 0LW, UK (tel +44-181-943-7002, fax +44-181-977-7091) for a copy.

The following books largely concerning Z have been or are due to be published (in approximate chronological order):

- I. Hayes (ed.), Specification Case Studies, Prentice Hall International Series in Computer Science, 1987. (2nd ed., 1993)
- J. M. Spivey, Understanding Z: A specification language and its formal semantics, Cambridge University Press, 1988.
- D. Ince, An Introduction to Discrete Mathematics, Formal System Specification and Z, Oxford University Press, 1988. (2nd ed., 1993)
- J. C. P. Woodcock & M. Loomes, Software Engineering Mathematics, Addison-Wesley, 1989.
- J. M. Spivey, The Z Notation: A reference manual, Prentice Hall International Series in Computer Science, 1989. (2nd ed., 1992)
  Widely used as the current de facto standard for Z.
- A. Diller, Z: An introduction to formal methods, Wiley, 1990.
- J. E. Nicholls (ed.), Z user workshop, Oxford 1989, Springer-Verlag, Workshops in Computing, 1990.
- B. Potter, J. Sinclair & D. Till, An Introduction to Formal Specification and Z, Prentice Hall International Series in Computer Science, 1991.
- D. Lightfoot, Formal Specification using Z, MacMillan, 1991.

- A. Norcliffe & G. Slater, Mathematics for Software Construction, Ellis Horwood, 1991.
- J. E. Nicholls (ed.), Z User Workshop, Oxford 1990, Springer-Verlag, Workshops in Computing, 1991.
- I. Craig, The Formal Specification of Advanced AI Architectures, Ellis Horwood, 1991.
- M. Imperato, An Introduction to Z, Chartwell-Bratt, 1991.
- J. B. Wordsworth, Software Development with Z, Addison-Wesley, 1992.
- S. Stepney, R. Barden & D. Cooper (eds.), Object Orientation in Z, Springer-Verlag, Workshops in Computing, August 1992.
- J. E. Nicholls (ed.), Z User Workshop, York 1991, Springer-Verlag, Workshops in Computing, 1992.
- D. Edmond, Information Modeling: Specification and implementation, Prentice Hall, 1992.
- J. P. Bowen & J. E. Nicholls (eds.), Z User Workshop, London 1992, Springer-Verlag, Workshops in Computing, 1993.
- S. Stepney, High Integrity Compilation: A case study, Prentice Hall, 1993.
- M. McMorran & S. Powell, Z Guide for Beginners, Blackwell Scientific, 1993.
- K. C. Lano & H. Haughton (eds.), Object-oriented Specification Case Studies, Prentice Hall International Object-Oriented Series, 1993.
- B. Ratcliff, Introducing Specification using Z: A practical case study approach, McGraw-Hill, 1994.
- A. Diller, Z: An introduction to formal methods, 2nd ed., Wiley, 1994.
- J. P. Bowen & J. A. Hall (eds.), Z User Workshop, Cambridge 1994, Springer-Verlag, Workshops in Computing, 1994.
- R. Barden, S. Stepney & D. Cooper, Z in Practice, Prentice Hall BCS Practitioner Series, 1994.
- D. Rann, J. Turner & J. Whitworth, Z: A beginner's guide. Chapman & Hall, 1994.
- D. Heath, D. Allum & L. Dunckley, Introductory Logic and Formal Methods. A. Waller, Henley-on-Thames, 1994.
- L. Bottaci and J. Jones, Formal Specification using Z: A modelling approach. International Thomson Publishing, London, 1995.
- D. Sheppard, An Introduction to Formal Specification with Z and VDM. McGraw Hill International Series in Software Engineering, 1995.
- J. P. Bowen & M. G. Hinchey (eds.), ZUM'95: The Z formal specification notation, 9th International Conference of Z Users, Springer-Verlag, Lecture Notes in Computer Science, 1995.

A.9 Object-oriented Z

Several object-oriented extensions to or versions of Z have been proposed. The book *Object Orientation in Z* [387], is a collection of papers describing various OOZ approaches – Hall, ZERO, MooZ, Object-Z, OOZE, Schuman & Pitt, Z^{++}, ZEST and Fresco (an OO VDM method) – in the main written by the methods' inventors, and all specifying the same two examples. A more recent book entitled *Object Oriented Specification Case Studies* [258] surveys the principal methods and languages for formal object-oriented specification, including Z-based approaches. For a fuller list of relevant publications, see page 249.

A.10 Executable Z

Z is a (non-executable in general) specification language, so there is no such thing as a Z compiler/linker/etc. as you would expect for a programming language. Some people have looked at animating subsets of Z for rapid prototyping purposes, using logic and functional programming for example, but this work is preliminary and is not really the major point of Z, which is to increase human understandability of the specified system and allow the possibility of formal reasoning and development. However, Prolog seems to be the main favoured language for Z prototyping and some references may be found in the Z bibliography (see Section A.8) and on page 246.

A.11 Meetings

Regular Z User Meetings (ZUM), now known as the International Conference of Z Users, have been held for a number of years. Details are issued on the newsgroup `comp.specification.z` and sent out on the Z User Group mailing list mentioned in Section A.12. Information on Z User Meetings is available via WWW under:

```
http://www.zuser.org/zum/
```

The proceedings for Z User Meetings have been published in the Springer-Verlag *Workshops in Computing* series from the 4th meeting in 1989 until the 8th meeting in 1994. The proceedings are now published in the *Lecture Notes in Computer Science* series by the same publisher. See page 248 for further information on published proceedings.

The Refinement Workshop is another relevant series, organized by BCS-FACS in the UK. The proceedings for these workshops are mainly published in the Springer-Verlag *Workshops in Computing* series.

The FME Symposium, the successor to the VDM-Europe series of conferences, is organized by Formal Methods Europe. The proceedings are published in the Springer-Verlag *Lecture Notes in Computer Science* series. The chairman of Formal Methods Europe is Prof. Peter Lucas, TU Graz, Austria (email `lucas@ist.tu-graz.ac.at`).

The IFIP WG6.1 International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols (FORTE) addresses formal techniques and testing methodologies applicable to distributed systems such as Estelle, LOTOS, SDL, ASN.1, Z, etc.

WIFT (Workshop on Industrial-strength Formal specification Techniques) is a new workshop started in the US in 1994. The proceedings are published by the IEEE Computer Society [159].

ICECCS (IEEE International Conference on Engineering of Complex Computer Systems) includes a formal methods track and is organized from the US.

Details of Z-related meetings can be advertized in the `comp.specification.z` newsgroup if desired. All the above meetings are likely to be repeated in some form.

A.12 Z User Group

The Z User Group (ZUG) was set up in 1992 to oversee Z-related activities, and the Z User Meetings in particular. As a subscriber to `comp.specification.z`, Z FORUM or the postal mailing list, you may consider yourself a member of the Z User Group. There are currently no charges for membership, although this is subject to review if necessary. Contact `zforum-request@comlab.ox.ac.uk` for further information.

A.13 Draft Z Standard

The proposed Z standard under ISO/IEC JTC1/SC22 is available electronically via anonymous FTP *only* (not via the mail server since it is too large) from the Z archive at Oxford in compressed POSTSCRIPT format. Version 1.0 of the draft standard is accessible as a main file and an annex file:

```
ftp://ftp.comlab.ox.ac.uk/pub/Zforum/zstandard1.0.ps.Z
ftp://ftp.comlab.ox.ac.uk/pub/Zforum/zstandard-annex1.0.ps.Z
```

It is also available in printed form from the Oxford University Computing Laboratory librarian (email `library@comlab.ox.ac.uk`, tel +44-1865-273837, fax +44-1865-273839) by requesting Technical Monograph number PRG-107.

A.14 Related Organizations

The BCS-FACS (British Computer Society Formal Aspects of Computer Science special interest group) and FME (Formal Methods Europe) are two organizations interested in formal methods in general. Contact BCS-FACS, Dept. of Computer Studies, Loughborough University of Technology, Loughborough, Leicester LE11 3TU, UK (tel +44-1509-222676, fax +44-1509-211586, email `FACS@lut.ac.uk`) for further information. A *FACS Europe* newsletter is issued to members of FACS and FME. Please send suitable Z-related material to the Z column editor, David Till, Dept. of Computer Science, City University, Northampton Square, London, EC1V 0HB, UK (tel +44-171-477-8552, email `till@cs.city.ac.uk`) for possible publication. Material from articles appearing on the `comp.specification.z` newsgroup may be included if considered of sufficient interest (with permission from the originator if possible). It would be helpful for posters of articles on `comp.specification.z` to indicate if they do not want further distribution for any reason.

A.15 Comparisons of VDM and Z

See page 244 for a list of publications on this subject. In particular, see [205], available as an on-line Technical Report from the University of Manchester [202]. VDM is discussed on the (unmoderated) VDM FORUM mailing list. To contact the list administrator, send electronic mail to `vdm-forum-request@jiscmail.ac.uk`.

A.16 Corrections

Please send corrections or new relevant information about meetings, books, tools, etc., to `bowenjp@sbu.ac.uk`. New questions and model answers are also gratefully received!

Appendix B

Z Glossary

Names

a,b	identifiers
d,e	declarations (e.g., $a : A;\ b, \ldots : B$...)
f,g	functions
m,n	numbers
p,q	predicates
s,t	sequences
x,y	expressions
A,B	sets
C,D	bags
Q,R	relations
S,T	schemas
X	schema text (e.g., d, $d \mid p$ or S)

Definitions

$a == x$	Abbreviated definition
$a ::= b \mid \ldots$	Data type definition (or $a ::= b\langle\langle x \rangle\rangle \mid \ldots$)
$[a]$	Introduction of a given set or basic type (or $[a, \ldots]$)
$a\_$	Prefix operator
$\_a$	Postfix operator
$\_a\_$	Infix operator

Logic

true	Logical true constant
false	Logical false constant
$\neg\ p$	Logical negation
$p \land q$	Logical conjunction
$p \lor q$	Logical disjunction
$p \Rightarrow q$	Logical implication ($\neg\ p \lor q$)
$p \Leftrightarrow q$	Logical equivalence ($p \Rightarrow q\ \land\ q \Rightarrow p$)

$\forall X \bullet q$	Universal quantification
$\exists X \bullet q$	Existential quantification
$\exists_1 X \bullet q$	Unique existential quantification
let $a == x;\ \ldots \bullet p$	Local definition

Sets and expressions

$x = y$	Equality of expressions
$x \neq y$	Inequality ($\neg\ (x = y)$)
$x \in A$	Set membership
$x \notin A$	Non-membership ($\neg\ (x \in A)$)
$\emptyset$	Empty set (also $\{\}$)
$A \subseteq B$	Set inclusion or subset
$A \subset B$	Strict set inclusion or subset ($A \subseteq B \land A \neq B$)
$\{x, y, \ldots\}$	Set of elements
$\{X \bullet x\}$	Set comprehension
$\lambda X \bullet x$	Lambda-expression – function
$\mu X \bullet x$	Mu-expression – unique value
let $a == x;\ \ldots \bullet y$	Local definition
if p **then** x **else** y	Conditional expression
$(x, y, \ldots)$	Ordered tuple
$A \times B \times \ldots$	Cartesian product
$\mathbb{P} A$	Power set (set of subsets)
$\mathbb{P}_1 A$	Non-empty power set
$\mathbb{F} A$	Set of finite subsets
$\mathbb{F}_1 A$	Non-empty set of finite subsets
$A \cap B$	Set intersection
$A \cup B$	Set union
$A \setminus B$	Set difference
$\bigcup A$	Generalized union of a set of sets
$\bigcap A$	Generalized intersection of a set of sets
$\mathit{first}\, x$	First element of an ordered pair
$\mathit{second}\, x$	Second element of an ordered pair
$\#A$	Size or cardinality of a finite set

Relations

$A \leftrightarrow B$	Relation ($\mathbb{P}(A \times B)$)
$a \mapsto b$	Maplet ((a, b))
$\mathrm{dom}\, R$	Domain of a relation
$\mathrm{ran}\, R$	Range of a relation
$\mathrm{id}\, A$	Identity relation
$Q \fatsemi R$	Forward relational composition
$Q \circ R$	Backward relational composition ($R \fatsemi Q$)
$A \lhd R$	Domain restriction
$A \lhd\!\!\!\!- R$	Domain anti-restriction

$A \rhd R$	Range restriction
$A \rhd\!\!= R$	Range anti-restriction
$R(\!\mid A \mid\!)$	Relational image
iter n R	Relation composed *n* times
R^n	Same as *iter n R*
$R^\sim$	Inverse of relation
R^*	Reflexive-transitive closure
R^+	Irreflexive-transitive closure
$Q \oplus R$	Relational overriding ($(\mathrm{dom}\, R \lhd Q) \cup R$)
$a \underline{R} b$	Infix relation

Functions

$A \pfun B$	Partial functions
$A \fun B$	Total functions
$A \pinj B$	Partial injections
$A \inj B$	Total injections
$A \psurj B$	Partial surjections
$A \surj B$	Total surjections
$A \bij B$	Bijective functions
$A \ffun B$	Finite partial functions
$A \finj B$	Finite partial injections
$f\, x$	Function application (or $f(x)$)

Numbers

$\mathbb{Z}$	Set of integers
$\mathbb{N}$	Set of natural numbers $\{0, 1, 2, ...\}$
$\mathbb{N}_1$	Set of non-zero natural numbers ($\mathbb{N} \setminus \{0\}$)
$m + n$	Addition
$m - n$	Subtraction
$m * n$	Multiplication
m div n	Division
m mod n	Modulo arithmetic
$m \leq n$	Less than or equal
$m < n$	Less than
$m \geq n$	Greater than or equal
$m > n$	Greater than
succ n	Successor function $\{0 \mapsto 1, 1 \mapsto 2, ...\}$
$m \,..\, n$	Number range
min A	Minimum of a set of numbers
max A	Maximum of a set of numbers

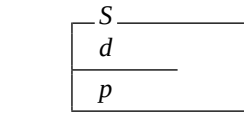
Sequences

$\mathrm{seq}\, A$	Set of finite sequences
$\mathrm{seq}_1\, A$	Set of non-empty finite sequences
$\mathrm{iseq}\, A$	Set of finite injective sequences
$\langle\rangle$	Empty sequence
$\langle x, y, \ldots\rangle$	Sequence $\{1 \mapsto x, 2 \mapsto y, \ldots\}$
$s \frown t$	Sequence concatenation
$\frown / s$	Distributed sequence concatenation
head s	First element of sequence ($s(1)$)
tail s	All but the head element of a sequence
last s	Last element of sequence ($s(\#s)$)
front s	All but the last element of a sequence
rev s	Reverse a sequence
squash f	Compact a function to a sequence
$A \upharpoonright s$	Sequence extraction ($squash(A \lhd s)$)
$s \upharpoonright A$	Sequence filtering ($squash(s \rhd A)$)
s prefix t	Sequence prefix relation ($s \frown v = t$)
s suffix t	Sequence suffix relation ($u \frown s = t$)
s in t	Sequence segment relation ($u \frown s \frown v = t$)
disjoint A	Disjointness of an indexed family of sets
A partition B	Partition an indexed family of sets

Bags

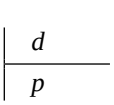
$\mathrm{bag}\, A$	Set of bags or multisets ($A \nrightarrow \mathbb{N}_1$)
$[\![]\!]$	Empty bag
$[\![x, y, \ldots]\!]$	Bag $\{x \mapsto 1, y \mapsto 1, \ldots\}$
count $C\, x$	Multiplicity of an element in a bag
$C \sharp x$	Same as *count* $C\, x$
$n \otimes C$	Bag scaling of multiplicity
x in C	Bag membership
$C \sqsubseteq D$	Sub-bag relation
$C \uplus D$	Bag union
$C \uplus D$	Bag difference
items s	Bag of elements in a sequence

Schema notation



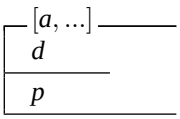
Vertical schema.

New lines denote ‘ ;’ and ‘$\wedge$’. The schema name and predicate part are optional. The schema may subsequently be referenced by name in the document.



Axiomatic description.

The description may be non-unique. The predicate part is optional. The definitions apply globally in the document subsequently.



Generic construction.

The generic parameters are optional. The definitions must be unique. The definitions apply globally in the document subsequently.

$S \mathrel{\widehat=} [X]$	Horizontal schema
$[T; ... \mid ...]$	Schema inclusion
$z.a$	Component selection (given $z : S$)
θS	Tuple of components
$\neg\, S$	Schema negation
$S \land T$	Schema conjunction
$S \lor T$	Schema disjunction
$S \Rightarrow T$	Schema implication
$S \Leftrightarrow T$	Schema equivalence
$S \setminus (a, ...)$	Hiding of component(s)
$S \upharpoonright T$	Projection of components
$\mathrm{pre}\ S$	Schema precondition
$S \fatsemi T$	Schema composition (S then T)
$S \gg T$	Schema piping (S outputs to T inputs)
$S[a/b, ...]$	Schema component renaming (b becomes a, etc.)
$\forall X \bullet S$	Schema universal quantification
$\exists X \bullet S$	Schema existential quantification
$\exists_1 X \bullet S$	Schema unique existential quantification

Conventions

$a?$	Input to an operation
$a!$	Output from an operation
a	State component before an operation
a'	State component after an operation
S	State schema before an operation
S'	State schema after an operation
ΔS	Change of state (normally $S \land S'$)
ΞS	No change of state (normally $[S \land S' \mid \theta S = \theta S']$)
ΦS	Partial specification of an operation
$\vdash p$	Theorem
$d \vdash p$	Theorem with declarations ($\vdash \forall d \bullet p$)

Appendix C

Literature Guide

C.1 Introduction

This literature guide contains a selected list of some pertinent publications for Z users. Most of those included are readily available, either as books or in journals. A few unpublished items have been included, where they are particularly relevant and can be obtained reasonably easily.

Some references in the bibliography at the back of the book are accompanied by an annotation. This may include a contents list (of a book), a list of the titles of Z related papers (in a collection) with cross references to the full details, or a summary of the work. Papers are arranged by subject (with authors and brief details of the subject matter), together with cross references to the full details in the bibliography.

C.2 Management, Style, and Method

C.2.1 Justification and introduction

For justifications for using formality, and quick introductions to Z, see:

[92, 283] Cohen/McDermid. Justification of formal methods and notations
[288] Meyer. On formalism in specifications
[377] Spivey. Introduction to Z
[421, 422] Wing. General introduction to formal methods including Z
[428] Woodcock. Structuring specifications

For discussion about using formal methods in practice, see:

[23] Barden *et al*. Z in practice
[26, 72, 73, 169, 285] Barroca/McDermid, Bowen/Stavridou and Gerhart *et al*. Formal methods and safety-critical systems
[167, 213] Gerhart and Hinchey/Bowen. Applications of formal methods
[184, 64, 68, 67] Hall and Bowen/Hinchey. Myths and guidelines about formal methods
[439] Worden. Fermenting and distilling ideas about formal and object-oriented methods in industry

C.2.2 Educational issues

Educational issues are presented and discussed in:

[101] Cooper. Educating management
[162, 163] Garlan. Effective integration of formal methods into a professional master of software engineering course
[353] Saiedian. The mathematics of computing
[401] Swatman. Educating information systems professionals

Various papers describing good specification style are:

[6] Ainsworth *et al*. The use of viewpoint specifications, a technique with concentrates on making large specifications more understandable
[134] Duke. Enhancing structure
[180, 181] Gravell. Minimization in specification/design and what makes a good specification
[269] Macdonald. Usage and abusage

C.2.3 Method integration

Much work has been done on attempting to integrate Z with traditional structured analysis methods. Some of this is described in:

[17] Aujla *et al*. A rigorous review technique
[82] Bryant *et al*. Structured methodologies and formal notations
[131] Draper. Z and SSADM
[123] Giovanni and Iachini. HOOD and Z
[274, 332, 333, 334] Polack, Mander, *et al*. SAZ Method – Structured Analysis and Z
[240, 241] Josephs and Redmond-Pyle. Entity-relationship models, structured methods, and Z
[344, 345] Randell. Data Flow Diagrams and Z
[363] Semmens and Allen. Yourdon and Z
[364] Semmens *et al*. Integrated structured analysis and formal specification techniques
[413] van Hee *et al*. Petri nets and Z

C.2.4 Z methodology

Other work towards the development of a 'method' for Z itself include:

[23] Barden *et al*. Z in practice
[187] Hall and McDermid. Towards a Z method using axiomatic specification in Z (using order sorted algebra and OBJ3 [176] in particular)
[309] Neilson. A rigorous development method from Z to C [244]
[424] Wood. A practical approach using Z and the refinement calculus
[443] Wordsworth. Software development with Z

The application of metrics to formal specifications has been studied:

[418, 20] Whitty and Bainbridge *et al*. Structural metrics for Z specifications

A formal specification in Z can be useful for deciding test cases, etc. Work on testing is reported in:

[8, 9] Ammann and Offutt. Functional and test specifications based on the category-partition method
[88] Carrington and Stocks. Formal methods and software testing
[113] Cusack and Wezeman. Deriving tests for objects specified in Z
[188] Hall. Testing with respect to formal specification
[197] Hayes. Specification directed module testing
[390] Stocks. Applying formal methods to software testing
[391] Stocks and Carrington. Deriving software test cases from formal specifications
[392, 393] Stocks and Carrington. Test templates: a specification-based testing framework and case study

C.3 Application Areas

C.3.1 Surveys

Surveys of formal methods, including Z users, are reported in:

[18] Austin and Parkin. Formal methods: a survey
[22] Barden *et al*. Use of Z (in the UK)
[107, 105, 106, 109, 108, 168, 169] Craigen *et al*. An international survey of major industrial formal methods applications, including a number using Z

C.3.2 CICS transaction processing system

One of the high profile users of Z is IBM UK Laboratories at Hursley for specification and development of the CICS transaction processing system. General descriptions of the CICS project include:

[94] Collins *et al*. Introducing formal methods: the CICS experience with Z
[218] Houston and King. CICS project report
[319] Nix and Collins. Use of software engineering and Z in the development of CICS
[328] Phillips. CICS experience throughout the life-cycle
[442] Wordsworth. The CICS Application Programming Interface (API) definition

Specifying secure systems is discussed in:

[235] Jones. Verification of critical properties
[375] Smith and Keighley. A secure transaction mechanism (SWORD secure DBMS)

C.3.3 Specification of hardware

Not all Z specifications are of software systems. Much interesting and important work has been done on formally specifying hardware, including microprocessors. The Inmos T800 Transputer Floating Point Unit (FPU)microcode development is a major real example where formal methods have saved time by reducing the amount of testing needed.

[280, 282, 281, 368, 367] May, Shepherd *et al.* T800 transputer FPU development

More technical papers on hardware applications (including embedded software) are:

[24] Barrett. A floating-point number system (IEEE-754 standard)
[38, 39, 45, 61] Bowen. Microprocessor instruction sets (Motorola 6800 and Transputer)
[119, 120, 164, 165, 199] Delisle/Garlan and Hayes. Oscilloscopes, including reuse of specifications
[242, 243] Kemp. Viper microprocessor
[374] Smith and Duke. Cache coherence protocol (in Object-Z)
[379] Spivey. Real-time kernel

Communications systems and protocols are specified in:

[38] Bowen *et al.* Network services via remote procedure calls (RPC)
[84] Butler. Service extension (PABX)
[139] Duke *et al.* Protocol specification and verification using Z
[142] Duke *et al.* Object-oriented protocol specification (mobile phone system, in Object-Z)
[179] Gotzhein. Open distributed systems
[196] Haughton. Safety and liveness properties of communication protocols
[276, 278] Mataga and Zave. Formal specification of telephone features
[330] Pilling *et al.* Inheritance protocols for real-time scheduling
[337] Potter and Till. Gateway functions within a communications network
[445] Zave and Jackson. Specification of switching systems (PBX)

C.3.4 Graphics and HCI

The following papers describe the use of Z for various graphics applications, standards (especially GKS), and Human-Computer Interfaces (HCI):

[2] Abowd *et al.* A survey of user interface languages including Z
[10, 11] Arnold *et al.* Configurable models of graphics systems (GKS)
[43, 47] Bowen. Formal specification of window systems (X in particular)
[80] Brown and Bowen. An extensible input system for UNIX
[129] Dix *et al.* Human-Computer Interaction (HCI)
[133] Duce *et al.* Formal specification of *Presentation Environments for Multimedia Objects* (PREMO)
[136, 137] Duke and Harrison. Event model of human-system interaction and mapping user requirements to implementations
[192] Harrison. Engineering human-error tolerant software
[303, 304] Narayana and Dharap. Formal specification of a Look Manager and a dialog system
[307] Nehlig and Duce. Formal specification of the GKS design primitive
[398] Sufrin. Formal specification of a display-oriented editor
[397, 400] Sufrin and He. Effective user interfaces and interactive processes
[406] Took. A formal design for an autonomous display manager

C.3.5 Safety-critical systems

An important application area for formal methods is safety-critical systems where human lives may depend on correctness of the system.

[26, 72, 73, 169, 285] Barroca/McDermid, Bowen/Stavridou and Gerhart *et al*. Surveys covering formal methods and safety-critical systems
[48, 63, 73] Bowen *et al*. Safety-critical systems and standards
[97] Coombes *et al*. Formal specification of an aerospace system: the attitude monitor
[190] Hamilton. The industrial use of Z within a safety-critical software system
[189] Hamer and Peleska. Z applied to the A330/340 cabin communication system
[231] Johnson. Using Z to support the design of interactive safety-critical systems
[250] Knight and Littlewood. Special issue of *IEEE Software* on *Safety-Critical Systems*
[331] Place and Kang. Safety-critical software: status report and annotated bibliography

Some examples of the application of Z to safety-critical systems are:

[25] Barroca *et al*. Architectural specification of an avionic subsystem
[226, 227, 228, 229] Jacky. Formal specifications for a clinical cyclotron
[249] Knight and Kienzle. Using Z to specify a safety-critical system in the medical sector
[351] Ruddle. Specification of real-time safety-critical control systems

C.3.6 Other Z applications

Other papers describing a variety of applications using Z include:

[1] Abowd *et al*. Software architectures
[35] Boswell. Specification and validation of a security policy model for the NATO Air Command and Control System (ACCS)
[44] Bowen. A text formatting tool
[53, 255] Bowen, Lano and Breuer. Reverse engineering
[81] Brownbridge. CASE toolset (for SSADM)
[83] Butcher. A behavioural semantics for Linda-2
[104] Craig. Specification of advanced Artificial Intelligence (AI) architectures
[115, 116, 117] de Barros and Harper. Formal specification and derivation of relational database applications
[151] Fenton and Mole. Flow-graph transformation
[298] Morgan and Sufrin. Specification of the UNIX file system
[305] Nash. Large systems
[348] Reizer *et al*. Requirements specification of a proposed POSIX standard
[385] Stepney. High integrity compilation
[399] Sufrin. A Z model of the UNIX *make* utility
[434] Woodcock *et al*. Formal specification of the UK Interim Defence Standard 00-56
[447] Zhang and Hitchcock. Designing knowledge-based systems and information systems

C.4 Textbooks on Z

[36] Bottaci and Jones. Formal specification using Z: a modelling approach
[126] Diller. Z: an introduction to formal methods (2nd edition)
[203] Hayes *et al*. Specification case studies (the first book on Z, now in its 2nd edition, containing an excellent selection of example Z specifications)
[209] Heath, Allum and Dunckley. Introductory logic and formal methods
[222] Imperato. An introduction to Z
[223] Ince. An introduction to discrete mathematics and formal system specification (2nd edition)
[262] Lightfoot. Formal specification using Z
[286] McMorran and Powell. Z guide for beginners
[320] Norcliffe and Slater. Mathematics of software construction
[336] Potter, Sinclair and Till. An introduction to formal specification and Z (a popular first textbook on Z)
[346] Rann, Turner and Whitworth. Z: a beginner's guide
[347] Ratcliff. Introducing specification using Z
[369] Sheppard. An introduction to formal specification with Z and VDM
[437] Woodcock and Loomes. Software engineering mathematics
[443] Wordsworth. Software development with Z

A video course is also available [321, 322].

C.5 Language Details

C.5.1 Syntax and semantics

Z's syntax, semantics and mathematical toolkit are being internationally standardized under ISO/IEC JTC1/SC22. A draft version of the standard is available:

[79] Brien and Nicholls. Z Base Standard, version 1.0

The definition of the Z syntax and mathematical toolkit used by many practitioners is:

[381] Spivey. Z Reference Manual ('ZRM', 2nd edition)

More technical works describing Z's formal semantics are:

[412] Diepen and van Hee. The link between Z and the relational algebra
[161] Gardiner *et al*. A simpler semantics
[376] Spivey. Understanding Z
[382] Spivey and Sufrin. Type inference

C.5.2 Z and VDM

Z is often compared and contrasted with VDM (Vienna Development Method). The following papers show the cross-fertilization and comparisons between the two:

[30] Bera. Structuring for the VDM specification language, in response to the Z schema notation
[171] Gilmore. Correctness-oriented approaches to software development in which the Z, VDM and algebraic styles are compared

[202] Hayes. A comparative case study of VDM and Z
[205] Hayes *et al*. Understanding the differences between VDM and Z
[263, 264] Lindsay. A VDM perspective on reasoning about Z specifications and transferring VDM verification techniques to Z
[265] Lindsay and van Keulen. Case studies in the verification of specifications in VDM and Z
[293] Monahan and Shaw. Model-based specifications, including a discussion of the respective trade-offs in specification between Z and VDM
[369] Sheppard. A book introducing formal specification with Z and VDM

C.5.3 Reasoning about specifications

Reasoning about Z specifications is addressed in:

[297] Morgan and Sanders. Laws of the Logical Calculi
[425] Woodcock. Calculating properties (preconditions)
[433, 275] Woodcock/Brien and Martin. $\mathcal{W}$, a logic for Z.

C.5.4 Refinement

Work on refining Z-like specifications towards an implementation (see also Section C.6.1) includes:

[19] Bailes and Duke. Class refinement
[24] Barrett. Refinement from Z to microcode via Occam
[27] Baumann. Z and natural semantics programming language theory for algorithm refinement
[125, 126] Diller. Hoare logic and Part II: *Methods of Reasoning*
[154, 155, 156] Fidge. Real-time refinement and program development
[171] Gilmore. Correctness-oriented approaches to software development (Z, VDM and algebraic styles are compared)
[208] He *et al*. Foundations for data refinement
[230] Jacob. Varieties of refinement
[238] Josephs. Data refinement calculation for Z specifications
[248] King and Sørensen. Specification and design of a library system
[254, 257] Lano and Haughton. Reasoning and refinement in object-oriented specification languages
[271, 272, 273] Mahoney/Hayes *et al*. Timed refinement
[308, 309] Neilson. Hierarchical refinement of Z specifications and a rigorous development method from Z to C [244]
[365] Sennett. Using refinement to convince (pattern matching in ML)
[366] Sennett. Demonstrating the compliance of Ada programs [341] with Z specifications
[400] Sufrin and He. Specification, analysis and refinement of interactive processes
[420] Whysall and McDermid. Object-oriented specification and refinement
[423] Wood. Software refinery
[430] Woodcock. Implementing promoted operations in Z
[438] Woodcock and Morgan. Refinement of state-based concurrent systems

[443] Wordsworth. Software development with Z

C.5.5 Refinement Calculus

The 'refinement calculus' approach to refinement is espoused in:

[246] King. Z and the refinement calculus
[295] Morgan. A standard student textbook (2nd edition)
[299] Morgan and Vickers. Collected research papers
[424] Wood. A practical approach using Z and the refinement calculus

The related B-Method, with associated B-Tool, B-Toolkit and Abstract Machine Notation (AMN), have been developed by Abrial *et al*., also the progenitor of Z:

[3, 4, 5] Abrial. The B-Tool, B-Method and forthcoming B-Book
[118] Dehbonei and Mejia. Use of B in the railways signalling industry
[127] Diller and Docherty. A comparison of Z and Abstract Machine Notation
[260] Lano and Haughton. Formal development in B AMN (a tutorial paper)
[310, 311] Neilson and Prasad. ZedB (a prototype B-based proof tool)
[349] Ritchie *et al*. Experiences in using the Abstract Machine Notation in a GKS graphics standard case study
[395] Storey and Haughton. A strategy for the production of verifiable code using the B-Method

C.5.6 Executable specifications

Execution of formal specifications is a subject of perennial debate. See:

[204] Hayes and Jones. Specifications are not (necessarily) executable

A retort may be found in:

[160] Fuchs. Specifications are (preferably) executable

Animating Z specifications is discussed in:

[77] Breuer and Bowen. Correct executable semantics for Z using abstract interpretation, including an informal taxonomy of approaches
[124] Dick *et al*. Computer-aided transformation of Z into the logic programming language Prolog
[126] Diller. Part IV: *Specification Animation* (using the functional programming language Miranda)
[130] Doma and Nicholl. EZ: automatic prototyping
[177] Goodman. Animating Z specifications in Haskell using a monad
[194, 195] Hasselbring. Animation of Object-Z specifications with a set-oriented prototyping language
[232] Johnson and Sanders. Functional implementations (Z to Miranda)
[266] Love. Animating Z specifications in SQL
[289] Minkowitz *et al*. C++ [396] library for implementing specifications
[389] Stepney and Lord. An access control system (Z to Prolog)
[410, 409] Valentine. Z^{--}, an executable subset of Z
[416] West and Eaglestone. Two approaches to animation (Z to Prolog)

Further information on executing Z may be found on page 229.

C.5.7 Language features

Specific language features are addressed in:

[15, 371] Arthan and Smith. Free types in Z (including recursion)
[198] Hayes. A generalization of bags
[200] Hayes. Interpretations of schema operators
[201] Hayes. Multi-relations in Z (a cross between multisets and binary relations)
[267] Lupton. Promotion and forward simulation
[298] Morgan and Sufrin. Schema framing
[411] Valentine. Adding real numbers to the Z mathematical toolkit
[426, 430] Woodcock. Proof rules for promotion and implementing promoted operations

C.5.8 Concurrent systems

Some research has been undertaken in using and adapting Z to model concurrent systems:

[99] Coombes and McDermid. Specifying distributed real-time systems
[147, 148] Evans. Visualizing, specifying and verifying concurrent systems using Z
[252] Lamport. TLZ: Temporal Logic of Actions (TLA) and Z
[304] Narayana and Dharap. Invariant properties in a dialog system
[362] Schuman *et al*. Object-oriented process specification

C.5.9 Z and CSP

In particular, there has been some work on combining Z and CSP (Communicating Sequential Processes), a formal process model with associated algebra for concurrent systems:

[28] Benjamin. A message passing system: an example of combining CSP and Z
[239] Josephs. Theoretical work on a state-based approach to communicating processes
[438] Woodcock and Morgan. Refinement of state-based concurrent systems

C.5.10 Real-time systems

Researchers have also considered modelling and reasoning about real-time systems, for example, by combining temporal logic with Z.

[99] Coombes and McDermid. Specifying temporal requirements for distributed real-time systems
[143] Duke and Smith. Temporal logic and Z specifications
[146] Engel. Specifying real-time systems with Z and the Duration Calculus
[152] Fergus and Ince. Modal logic and Z specifications item[[153]] Fidge. Specification and verification of real-time behaviour using Z and RTL
[154, 155, 156] Fidge. Real-time refinement and program development

[207] He Jifeng *et al*. Provably correct systems, including the use of Duration Calculus with schemas for structuring
[252] Lamport. TLZ: Temporal Logic of Actions (TLA) and Z
[271, 272, 273] Mahoney/Hayes *et al*. Timed refinement
[304] Narayana and Dharap. Invariant properties in a dialog system using Z and temporal logic
[330] Pilling *et al*. Inheritance protocols for real-time scheduling
[351] Ruddle. Specification of real-time safety-critical control systems
[372] Smith. An object-oriented approach including a formalization of temporal logic history invariants

C.6 Collections of papers

C.6.1 Conference proceedings

Regular Z User Meetings (ZUM) are organized by the Z User Group (ZUG) and have had full formally published proceedings since the 4th meeting:

[40] Bowen. 2nd Z User Meeting, Oxford, 1987
[42] Bowen. 3rd Z User Meeting, Oxford, 1988
[313] Nicholls. 4th Z User Meeting, Oxford, 1989
[315] Nicholls. 5th Z User Meeting, Oxford, 1990
[317] Nicholls. 6th Z User Meeting, York, 1991
[70] Bowen and Nicholls. 7th Z User Meeting, London, 1992
[60] Bowen and Hall. 8th Z User Meeting, Cambridge, 1994
[69] Bowen and Hinchey. 9th International Conference of Z Users, Limerick, 1995

The annual Refinement Workshop is organized by the BCS-FACS Special Interest Group. Papers cover a variety of refinement techniques from specification to code, and include some Z examples.

[284] McDermid. 1st Refinement Workshop, York, 1988
[300] Morgan and Woodcock. 3rd Refinement Workshop, Hursley, 1990
[301] Morris and Shaw. 4th Refinement Workshop, Cambridge, 1991
[235] Jones *et al*. 5th refinement Workshop, London, 1992
[405] Till. 6th refinement Workshop, London, 1994

FME Symposia are held every 18 months, organized by Formal Methods Europe. These grew out of the the later VDM-Europe conferences which included papers on Z:

[33] Bloomfield *et al*. VDM'88, Dublin
[32] Bjørner *et al*. VDM'90, Kiel
[338, 339] Prehn and Toetenel. VDM'91, Noordwijkerhout
[435] Woodcock and Larsen. FME'93, Odense
[302] Naftalin *et al*. FME'94, Barcelona

WIFT (Workshop on Industrial-strength Formal Specification Techniques) is a US workshop that held its first meeting in 1994 [159], and is likely to be repeated.

C.6.2 Journal special issues

The following special issues of journals and magazines, either dedicated to Z, or to formal methods with some Z content, have appeared:

[394] Blair. *Computer Communications*, March 1992. The practical use of FDTs (Formal Description Techniques) in communications and distributed systems, including a paper on Z

[65] Bowen and Hinchey. *Information and Software Technology*, May/June 1995. Contains a selection of Z papers, mainly revised from ZUM'94 [60]

[95] Cooke. *The Computer Journal*, October/December 1992. Two special issues on formal methods, including Z

[167] Gerhart. *IEEE Software*, September 1990. Formal methods with an emphasis on Z in particular, published in conjunction with special Formal Methods issues of *IEEE Transactions on Software Engineering* and *IEEE Computer*, which also include papers on Z

[250] Knight and Littlewood. *IEEE Software*, January 1994. Safety-critical systems, including several papers mentioning formal methods and Z

[283] McDermid. *Software Engineering Journal*, January 1989. Special section on Z

[436] Woodcock and Larsen. *IEEE Transactions on Software Engineering*, February 1995. Contains the best papers from FME'93 [435])

[45, 367] Zedan. *Microprocessors and Microsystems*, December 1990. Special feature on *Formal aspects of microprocessor design*

C.7 Tools

The ZIP Project tools catalogue lists some tools that support formatting, checking and proof of Z specifications:

[326] Parker. Z tools catalogue

Details of individual tools may be found in:

[14] Arthan. A proof tool based on HOL which grew into ProofPower (see below)

[58, 59] Bowen and Gordon. Z and HOL (a tool based on higher order logic)

[157] Flynn *et al*. Formaliser (editor and type-checker)

[236] Jones. ICL ProofPower (a commercial tool based on HOL)

[237, 408] Jordan *et al*. CADiℤ (formatter and type-checker)

[310, 311] Neilson and Prasad. ZedB (a prototype B-based schema expansion and precondition calculator tool)

[352] Saaltink. Z and EVES (a tool based on ZF set theory)

[380] Spivey. *f*UZZ (a commercial LaTeX formatter and type-checker, 2nd edition)

[444] Xiaoping Jia. ZTC (a freely available type-checker)

C.8 Object-Oriented Approaches

There has been much work recently to enhance Z with some of the structuring ideas from object-orientation. Overviews and comparisons can be found in:

[86] Carrington. ZOOM workshop report
[258] Lano and Haughton. Object-oriented specification case studies, many using extensions to Z
[387, 388] Stepney *et al*. Collected papers and a survey on object-orientation in Z

Object-Z is the best-documented and probably most widely used object-oriented extension to Z. The definitive description of the language is:

[141] Duke *et al*. Version 1 of Object-Z

Other Object-Z papers include:

[87] Carrington *et al*. Object-Z: an object-oriented extension to Z
[133] Duce *et al*. Formal specification of *Presentation Environments for Multimedia Objects* (PREMO)
[135] Duke and Duke. Towards a semantics
[138] Duke and Duke. Aspects of object-oriented specification (card game example)
[142] Duke *et al*. Object-oriented protocol specification (mobile phone system)
[194] Hasselbring. Animation with a set-oriented prototyping language
[342] Rafsanjani and Colwill. From Object-Z to C++ [396]
[374] Smith and Duke. Cache coherence protocol

Descriptions of other object-oriented approaches in conjunction with Z may be found in:

[7] Alencar and Goguen. OOZE: an object-oriented Z environment
[90] Chan and Trinder. An object-oriented data model supporting multi-methods, multiple inheritance, and static type-checking
[111] Cusack. Inheritance in object-oriented Z
[185, 186] Hall. A specification calculus for object-oriented systems and class hierarchies in Z
[191] Hammond. Producing Z specifications from object-oriented analysis
[253, 256, 53] Lano/Haughton *et al*. Z^{++}: an object-oriented extension to Z
[279] Maung and Howse. Hyper-Z: a new approach to object-orientation
[287] Meira and Cavalcanti. MooZ: Modular object-oriented Z specifications
[361, 362] Schuman, Pitt *et al*. Object-oriented subsystem and process specification
[417] Wezeman and Judge. Z for managed objects
[419, 420] Whysall and McDermid. Object-oriented specification and refinement

C.9 On-line Information

The BibTeX source for the bibliography at the end of the book is available on-line via the World Wide Web (WWW) under the following URL (Uniform Resource Locator):

```
http://www.zuser.org/z/bib.html
```

The bibliography is searchable. The user may provide a regular expression or select from a number of predefined keywords. Hyperlinks are included to some of the documents and other relevant details, such is book information, that can be accessed on-line.

Acknowledgements

Thanks are due to all those who suggested references for inclusion in the Z bibliography. This guide has been adapted from the collaborative ZIP project bibliography [386], originally produced by Susan Stepney and Rosalind Barden of Logic Cambridge Limited, together with the on-line Z bibliography held at the Oxford University Computing Laboratory [49]. Their original contribution is gratefully acknowledged.

Bibliography

[1] G. D. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architectures. *ACM Software Engineering Notes*, 18(5):9–20, December 1993.

[2] G. D. Abowd, J. P. Bowen, A. J. Dix, M. D. Harrison, and R. Took. User interface languages: A survey of existing methods. Technical Report PRG-TR-5-89, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK, October 1989.

[3] J.-R. Abrial. The B tool. In Bloomfield et al. [33], pages 86–87.

[4] J.-R. Abrial. The B method for large software, specification, design and coding (abstract). In Prehn and Toetenel [339], pages 398–405.

[5] J.-R. Abrial. *The B-Book*. Cambridge University Press, To appear.

Contents: Mathematical reasoning; Set notation; Mathematical objects; Introduction to abstract machines; Formal definition of abstract machines; Theory of abstract machines; Constructing large abstract machines; Example of abstract machines; Sequencing and loop; Programming examples; Refinement; Constructing large software systems; Example of refinement;

Appendices: Summary of the most current notations; Syntax; Definitions; Visibility rules; Rules and axioms; Proof obligations.

[6] M. Ainsworth, A. H. Cruikchank, P. J. L. Wallis, and L. J. Groves. Viewpoint specification and Z. *Information and Software Technology*, 36(1):43–51, 1994.

[7] A. J. Alencar and J. A. Goguen. OOZE: An object-oriented Z environment. In P. America, editor, *Proc. ECOOP'91 European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 180–199. Springer-Verlag, 1991.

[8] P. Ammann and J. Offutt. Functional and test specifications for the Mistix file system. Technical Report ISSE-TR-93-100, Department of Information & Software Systems Engineering, George Mason University, USA, January 1993.

[9] P. Ammann and J. Offutt. Using formal methods to mechanize category-partition testing. Technical Report ISSE-TR-93-105, Department of Information & Software Systems Engineering, George Mason University, USA, September 1993.

[10] D. B. Arnold, D. A. Duce, and G. J. Reynolds. An approach to the formal specification of configurable models of graphics systems. In G. Maréchal, editor, *Proc. EUROGRAPHICS'87, European Computer Graphics Conference and Exhibition*, pages 439–463. Elsevier Science Publishers (North-Holland), 1987.

The paper describes a general framework for the formal specification of modular graphics systems, illustrated by an example taken from the Graphical Kernel System (GKS) standard.

[11] D. B. Arnold and G. J. Reynolds. Configuring graphics systems components. *IEE/BCS Software Engineering Journal*, 3(6):248–256, November 1988.

[12] D. J. Arnold, D. A. Duce, and G. J. Reynolds. An approach to the formal specification of configurable models of graphics systems. In G. Maréchal, editor, *Proc. EUROGRAPHICS'87*, pages 439–463. Elsevier Science Publishers (North-Holland), 1987.

[13] D. J. Arnold and G. J. Reynolds. Configuring graphics systems components. *IEE/BCS Software Engineering Journal*, 3(6):248–256, November 1988.

[14] R. D. Arthan. Formal specification of a proof tool. In Prehn and Toetenel [338], pages 356–370.

[15] R. D. Arthan. On free type definitions in Z. In Nicholls [317], pages 40–58.

[16] AT&T Bell Laboratories, Murray Hill, New Jersey, USA. *UNIX™ Timesharing System, Programmer's Manual*, eighth edition, 1985. Volume 1.

[17] S. Aujla, A. Bryant, and L. Semmens. A rigorous review technique: Using formal notations within conventional development methods. In *Proc. 1993 Software Engineering Standards Symposium*, pages 247–255. IEEE Computer Society Press, 1993.

[18] S. Austin and G. I. Parkin. Formal methods: A survey. Technical report, National Physical Laboratory, Queens Road, Teddington, Middlesex, TW11 0LW, UK, March 1993.

[19] C. Bailes and R. Duke. The ecology of class refinement. In Morris and Shaw [301], pages 185–196.

[20] J. Bainbridge, R. W. Whitty, and J. B. Wordsworth. Obtaining structural metrics of Z specifications for systems development. In Nicholls [315], pages 269–281.

[21] R. Barden and S. Stepney. Support for using Z. In Bowen and Nicholls [70], pages 255–280.

[22] R. Barden, S. Stepney, and D. Cooper. The use of Z. In Nicholls [317], pages 99–124.

[23] R. Barden, S. Stepney, and D. Cooper. *Z in Practice*. BCS Practitioner Series. Prentice Hall, 1994.

[24] G. Barrett. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, 15(5):611–621, May 1989.

This paper presents a formalization of the IEEE standard for binary floating-point arithmetic in Z. The formal specification is refined into four components. The procedures presented form the basis for the floating-point unit of the Inmos IMS T800 transputer. This work resulted in a joint UK Queen's Award for Technological Achievement for Inmos Ltd and the Oxford University Computing Laboratory in 1990. It was estimated that the approach saved a year in development time compared to traditional methods.

[25] L. M. Barroca, J. S. Fitzgerald, and L. Spencer. The architectural specification of an avionic subsystem. In France and Gerhart [159], pages 17–29.

[26] L. M. Barroca and J. A. McDermid. Formal methods: Use and relevance for the development of safety-critical systems. *The Computer Journal*, 35(6):579–599, December 1992.

[27] P. Baumann. Z and natural semantics. In Bowen and Hall [60], pages 168–184.

[28] M. Benjamin. A message passing system: An example of combining CSP and Z. In Nicholls [313], pages 221–228.

[29] M. Benveniste. Writing operational semantics in Z: A structural approach. In Prehn and Toetenel [338], pages 164–188.

[30] S. Bera. Structuring for the VDM specification language. In Bloomfield et al. [33], pages 2–25.

[31] J. Bicarregui and B. Ritchie. Invariants, frames and postconditions: A comparison of the VDM and B notations. *IEEE Transactions on Software Engineering*, 21(2):79–89, 1995.

[32] D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors. *VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*. VDM-Europe, Springer-Verlag, 1990.

The 3rd VDM-Europe Symposium was held at Kiel, Germany, 17–21 April 1990. A significant number of papers concerned with Z were presented [89, 135, 164, 123, 179, 185, 246, 354, 383, 412, 438].

[33] R. Bloomfield, L. Marshall, and R. Jones, editors. *VDM – The Way Ahead*, volume 328 of *Lecture Notes in Computer Science*. VDM-Europe, Springer-Verlag, 1988.

The 2nd VDM-Europe Symposium was held at Dublin, Ireland, 11–16 September 1988. See [3, 30].

[34] D. Blyth, C. Boldyreff, C. Ruggles, and N. Tetteh-Lartey. The case for formal methods in standards. *IEEE Software*, pages 65–67, September 1990.

[35] A. Boswell. Specification and validation of a security policy model. *IEEE Transactions on Software Engineering*, 21(2):99–106, 1995.

This paper describes the development of a formal security model in Z for the NATO Air Command and Control System (ACCS): a large, distributed, multilevel-secure system. The model was subject to manual validation, and some of the issues and lessons in both writing and validating the model are discussed.

[36] L. Bottaci and J. Jones. *Formal Specification Using Z: A Modelling Approach*. International Thomson Publishing, London, 1995.

[37] S. R. Bourne. *The UNIX System*. International Computer Science Series. Addison-Wesley, 1982.

[38] J. P. Bowen. Formal specification and documentation of microprocessor instruction sets. *Microprocessing and Microprogramming*, 21(1–5):223–230, August 1987.

[39] J. P. Bowen. The formal specification of a microprocessor instruction set. Technical Monograph PRG-60, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK, January 1987.

The Z notation is used to define the Motorola M6800 8-bit microprocessor instruction set.

[40] J. P. Bowen, editor. *Proc. Z Users Meeting, 1 Wellington Square, Oxford*, Wolfson Building, Parks Road, Oxford, UK, December 1987. Oxford University Computing Laboratory.

The 1987 Z Users Meeting was held on Friday 8 December at the Department of External Studies, Rewley House, 1 Wellington Square, Oxford, UK.

[41] J. P. Bowen. Formal specification in Z as a design and documentation tool. In *Proc. Second IEE/BCS Conference on Software Engineering*, number 290 in Conference Publication, pages 164–168. IEE/BCS, July 1988.

[42] J. P. Bowen, editor. *Proc. Third Annual Z Users Meeting*, Wolfson Building, Parks Road, Oxford, UK, December 1988. Oxford University Computing Laboratory.

The 1988 Z Users Meeting was held on Friday 16 December at the Department of External Studies, Rewley House, 1 Wellington Square, Oxford, UK. Issued with *A Miscellany of Handy Techniques* by R. Macdonald, RSRE, *Practical Experience of Formal Specification: A programming interface for communications* by J. B. Wordsworth, IBM, and a number of posters.

[43] J. P. Bowen. Formal specification of window systems. Technical Monograph PRG-74, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK, June 1989.

Three window systems, X from MIT, WM from Carnegie-Mellon University and the Blit from AT&T Bell Laboratories are covered.

[44] J. P. Bowen. POS: Formal specification of a UNIX tool. *IEE/BCS Software Engineering Journal*, 4(1):67–72, January 1989.

[45] J. P. Bowen. Formal specification of the ProCoS/safemos instruction set. *Microprocessors and Microsystems*, 14(10):631–643, December 1990.

This article is part of a special feature on *Formal aspects of microprocessor design*, edited by H. S. M. Zedan. See also [367].

[46] J. P. Bowen. Z bibliography. Oxford University Computing Laboratory, 1990 onwards.

This bibliography is maintained in BIBTEX database format at the Oxford University Computing Laboratory. To add entries, please send as complete information as possible to `zforum-request@comlab.ox.ac.uk`.

[47] J. P. Bowen. X: Why Z? *Computer Graphics Forum*, 11(4):221–234, October 1992.

This paper asks whether window management systems would not be better specified through a formal methodology and gives examples in Z of the X Window System.

[48] J. P. Bowen. Formal methods in safety-critical standards. In *Proc. 1993 Software Engineering Standards Symposium*, pages 168–177. IEEE Computer Society Press, 1993.

[49] J. P. Bowen. Comp.specification.z and Z FORUM frequently asked questions. In Bowen and Hall [60], pages 397–404.

[50] J. P. Bowen. Select Z bibliography. In Bowen and Hall [60], pages 359–396.

[51] J. P. Bowen, editor. *Towards Verified Systems*, volume 2 of *Real-Time Safety Critical Systems series*. Elsevier Science Publishers, 1994.

[52] J. P. Bowen. Z glossary. *Information and Software Technology*, 37(5-6):333–334, May–June 1995.

[53] J. P. Bowen, P. T. Breuer, and K. C. Lano. Formal specifications in software maintenance: From code to Z^{++} and back again. *Information and Software Technology*, 35(11/12):679–690, November/December 1993.

[54] J. P. Bowen, M. Fränzle, E.-R. Olderog, and A. P. Ravn. Developing correct

systems. In *Proc. 5th Euromicro Workshop on Real-Time Systems*, pages 176–187. IEEE Computer Society Press, 1993.

[55] J. P. Bowen, R. B. Gimson, and S. Topp-Jørgensen. The specification of network services. Technical Monograph PRG-61, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK, August 1987.

[56] J. P. Bowen, R. B. Gimson, and S. Topp-Jørgensen. Specifying system implementations in Z. Technical Monograph PRG-63, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK, February 1988.

[57] J. P. Bowen and T. J. Gleeson. Distributed operating systems. In H. S. M. Zedan, editor, *Distributed Computer Systems: Theory and Practice*, chapter 1, pages 3–28. Butterworth Scientific Ltd., 1990.

[58] J. P. Bowen and M. J. C. Gordon. Z and HOL. In Bowen and Hall [60], pages 141–167.

[59] J. P. Bowen and M. J. C. Gordon. A shallow embedding of Z in HOL. *Information and Software Technology*, 37(5-6):269–276, May–June 1995.

Revised version of [58].

[60] J. P. Bowen and J. A. Hall, editors. *Z User Workshop, Cambridge 1994*, Workshops in Computing. Springer-Verlag, 1994.

Proceedings of the Eigth Annual Z User Meeting, St. John's College, Cambridge, UK. Published in collaboration with the British Computer Society. For individual papers, see [27, 58, 50, 49, 77, 88, 90, 127, 146, 148, 162, 186, 187, 191, 194, 252, 276, 332, 373, 417, 434, 439]. The proceedings also includes an *Introduction and Opening Remarks*, a *Select Z Bibliography* [50] and a section answering *Frequently Asked Questions* [49].

[61] J. P. Bowen, He Jifeng, R. W. S. Hale, and J. M. J. Herbert. Towards verified systems: The SAFEMOS project. In C. J. Mitchell and V. Stavridou, editors, *The Mathematics of Dependable Systems*, volume 55 of *The Institute of Mathematics and its Applications Conference Series*, pages 23–48. Oxford University Press, 1995.

[62] J. P. Bowen, He Jifeng, and P. K. Pandya. An approach to verifiable compiling specification and prototyping. In P. Deransart and J. Małuszyński, editors, *Programming Language Implementation and Logic Programming*, volume 456 of *Lecture Notes in Computer Science*, pages 45–59. Springer-Verlag, 1990.

[63] J. P. Bowen and M. G. Hinchey. Formal methods and safety-critical standards. *IEEE Computer*, 27(8):68–71, August 1994.

[64] J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods: Dispelling industrial prejudices. In Naftalin et al. [302], pages 105–117.

[65] J. P. Bowen and M. G. Hinchey. Editorial. *Information and Software Technology*, 37(5-6):258–259, May–June 1995.

A special issue on Z. See [52, 66, 59, 74, 163, 260, 274, 278, 409].

[66] J. P. Bowen and M. G. Hinchey. Report on Z User Meeting (ZUM'94). *Information and Software Technology*, 37(5-6):335–336, May–June 1995.

[67] J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, July 1995.

This article deals with further myths in addition to those presented in [184]. Previous versions issued as:

- Technical Report PRG-TR-7-94, Oxford University Computing Laboratory, June 1994.
- Technical Report 357, University of Cambridge, Computer Laboratory, January 1995.

[68] J. P. Bowen and M. G. Hinchey. Ten commandments of formal methods. *IEEE Computer*, 28(4):56–63, April 1995.

Previously issued as: Technical Report 350, University of Cambridge, Computer Laboratory, September 1994.

[69] J. P. Bowen and M. G. Hinchey, editors. *ZUM'95: The Z Formal Specification Notation, 9th International Conference of Z Users, Limerick, Ireland, September 7-9, 1995, Proceedings*, volume 967 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[70] J. P. Bowen and J. E. Nicholls, editors. *Z User Workshop, London 1992*, Workshops in Computing. Springer-Verlag, 1993.

Proceedings of the Seventh Annual Z User Meeting, DTI Offices, London, UK. Published in collaboration with the British Computer Society. For individual papers, see [21, 75, 100, 107, 113, 112, 131, 206, 227, 249, 259, 266, 279, 318, 323, 342, 351, 401, 411]. The proceedings also includes an *Introduction and Opening Remarks*, a *Select Z Bibliography* and a section answering *Frequently Asked Questions*.

[71] J. P. Bowen and V. Stavridou. Formal methods and software safety. In H. H. Frey, editor, *Safety of computer control systems 1992 (SAFECOMP'92), Computer Systems in Safety-critical Applications*, pages 93–98. Pergamon Press, 1992.

[72] J. P. Bowen and V. Stavridou. The industrial take-up of formal methods in safety-critical and other areas: A perspective. In Woodcock and Larsen [435], pages 183–195.

[73] J. P. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *IEE/BCS Software Engineering Journal*, 8(4):189–209, July 1993.

A survey on the use of formal methods, including B and Z, for safety-critical systems. Winner of the 1994 IEE Charles Babbage Premium award. A previous version is also available as Oxford University Computing Laboratory Technical Report PRG-TR-5-92.

[74] J. P. Bowen, S. Stepney, and R. Barden. Annotated Z bibliography. *Information and Software Technology*, 37(5-6):317–332, May–June 1995.

Revised version of [386].

[75] A. Bradley. Requirements for Defence Standard 00-55. In Bowen and Nicholls [70], pages 93–94.

[76] P. T. Breuer. Z! in progress: Maintaining Z specifications. In Nicholls [315], pages 295–318.

[77] P. T. Breuer and J. P. Bowen. Towards correct executable semantics for Z. In Bowen and Hall [60], pages 185–209.

[78] S. M. Brien. The development of Z. In D. J. Andrews, J. F. Groote, and C. A. Middelburg, editors, *Semantics of Specification Languages (SoSL)*, Workshops in Computing, pages 1–14. Springer-Verlag, 1994.

[79] S. M. Brien and J. E. Nicholls. Z base standard. Technical Monograph PRG-107, Oxford University Computing Laboratory, Wolfson Building, Parks Road,

Oxford, UK, November 1992. Accepted for standardization under ISO/IEC JTC1/SC22.

This is the first publicly available version of the proposed ISO Z Standard. See also [381] for the current most widely available Z reference manual.

[80] D. J. Brown and J. P. Bowen. The Event Queue: An extensible input system for UNIX workstations. In *Proc. European Unix Users Group Conference*, pages 29–52. EUUG, May 1987.

Available from EUUG Secretariat, Owles Hall, Buntingford, Hertfordshire SG9 9PL, UK.

[81] D. Brownbridge. Using Z to develop a CASE toolset. In Nicholls [313], pages 142–149.

[82] A. Bryant. Structured methodologies and formal notations: Developing a framework for synthesis and investigation. In Nicholls [313], pages 229–241.

[83] P. Butcher. A behavioural semantics for Linda-2. *IEE/BCS Software Engineering Journal*, 6(4):196–204, July 1991.

[84] M. J. Butler. Service extension at the specification level. In Nicholls [315], pages 319–336.

[85] J. N. Buxton and R. Malcolm. Software technology transfer. *Software Engineering Journal*, 6(1):17–23, January 1991.

[86] D. Carrington. ZOOM workshop report. In Nicholls [317], pages 352–364.

This paper records the activities of a workshop on Z and object-oriented methods held in August 1992 at Oxford. A comprehensive bibliography is included.

[87] D. Carrington, D. J. Duke, R. Duke, P. King, G. A. Rose, and G. Smith. Object-Z: An object-oriented extension to Z. In S. Vuong, editor, *Formal Description Techniques, II (FORTE'89)*, pages 281–296. Elsevier Science Publishers (North-Holland), 1990.

[88] D. Carrington and P. Stocks. A tale of two paradigms: Formal methods and software testing. In Bowen and Hall [60], pages 51–68.

Also available as Technical Report 94-4, Department of Computer Science, University of Queensland, 1994.

[89] P. Chalin and P. Grogono. Z specification of an object manager. In Bjørner et al. [32], pages 41–71.

[90] D. K. C. Chan and P. W. Trinder. An object-oriented data model supporting multi-methods, multiple inheritance, and static type checking: A specification in Z. In Bowen and Hall [60], pages 297–315.

[91] P. Chapront. Vital coded processor and safety related software design. In H. H. Frey, editor, *Safety of computer control systems 1992 (SAFECOMP'92), Computer Systems in Safety-critical Applications, Proc. IFAC Symposium*, pages 141–145. Pergamon Press, 1992.

[92] B. Cohen. Justification of formal methods for system specifications & A rejustification of formal notations. *IEE/BCS Software Engineering Journal*, 4(1):26–38, January 1989.

[93] D. Coleman. The technology transfer of formal methods: what's going wrong? In *Proc. 12th ICSE Workshop on Industrial Use of Formal Methods, Nice, France*, March 1990.

[94] B. P. Collins, J. E. Nicholls, and I. H. Sørensen. Introducing formal methods:

The CICS experience with Z. In B. Neumann et al., editors, *Mathematical Structures for Software Engineering*. Oxford University Press, 1991.

[95] J. Cooke. Editorial – formal methods: What? why? and when? *The Computer Journal*, 35(5):417–418, October 1992.

An editorial introduction to two special issues on *Formal Methods*. See also [26, 96, 290, 364, 432] for papers relevant to Z.

[96] J. Cooke. Formal methods – mathematics, theory, recipes or what? *The Computer Journal*, 35(5):419–423, October 1992.

[97] A. C. Coombes, L. Barroca, J. S. Fitzgerald, J. A. McDermid, L. Spencer, and A. Saeed. Formal specification of an aerospace system: The attitude monitor. In Hinchey and Bowen [213], pages 307–332.

[98] A. C. Coombes and J. A. McDermid. A tool for defining the architecture of Z specifications. In Nicholls [315], pages 77–92.

[99] A. C. Coombes and J. A. McDermid. Specifying temporal requirements for distributed real-time systems in Z. Computer Science Report YCS176, University of York, Heslington, York YO1 5DD, UK, 1992.

[100] A. C. Coombes and J. A. McDermid. Using diagrams to give a formal specification of timing constraints in Z. In Bowen and Nicholls [70], pages 119–130.

[101] D. Cooper. Educating management in Z. In Nicholls [313], pages 192–194.

[102] Coopers & Lybrand. *Safety related computer controlled systems market study*. HMSO, London, UK, 1992. Review for the Department of Trade and Industry.

[103] V. A. O. Cordeiro, A. C. A. Sampaio, and S. L. Meira. From MooZ to Eiffel – a rigorous approach to system development. In Naftalin et al. [302], pages 306–325.

[104] I. Craig. *The Formal Specification of Advanced AI Architectures*. AI Series. Ellis Horwood, September 1991.

This book contains two rather large (and relatively complete) specifications of Artificial Intelligence (AI) systems using Z. The architectures are the blackboard and Cassandra architectures. As well as showing that formal specification *can* be used in AI at the architecture level, the book is intended as a case-studies book, and also contains introductory material on Z (for AI people). The book assumes a knowledge of Z, so for non-AI people its primary use is for the presentation of the large specifications. The blackboard specification, with explanatory text, is around 100 pages.

[105] D. Craigen, S. L. Gerhart, and T. J. Ralston. Formal methods reality check: Industrial usage. In Woodcock and Larsen [435], pages 250–267.

Revised version in [108].

[106] D. Craigen, S. L. Gerhart, and T. J. Ralston. An international survey of industrial applications of formal methods. Technical Report NIST GCR 93/626-V1 & 2, Atomic Energy Control Board of Canada, US National Institute of Standards and Technology, and US Naval Research Laboratories, 1993.

Volume 1: Purpose, Approach, Analysis and Conclusions; Volume 2: Case Studies. Order numbers: PB93-178556/AS & PB93-178564/AS; National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161, USA.

[107] D. Craigen, S. L. Gerhart, and T. J. Ralston. An international survey of industrial applications of formal methods. In Bowen and Nicholls [70], pages 1–5.

[108] D. Craigen, S. L. Gerhart, and T. J. Ralston. Formal methods reality check: Industrial usage. *IEEE Transactions on Software Engineering*, 21(2):90–98, 1995.

Revised version of [228].

[109] D. Craigen, S. L. Gerhart, and T. J. Ralston. Formal methods technology transfer: Impediments and innovation. In Hinchey and Bowen [213], pages 399–419.

[110] D. Craigen, S. Kromodimoeljo, I. Meisels, W. Pase, and M. Saaltink. EVES: An overview. In Prehn and Toetenel [338], pages 389–405.

[111] E. Cusack. Inheritance in object oriented Z. In P. America, editor, *Proc. ECOOP'91 European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 167–179. Springer-Verlag, 1991.

[112] E. Cusack. Using Z in communications engineering. In Bowen and Nicholls [70], pages 196–202.

[113] E. Cusack and C. Wezeman. Deriving tests for objects specified in Z. In Bowen and Nicholls [70], pages 180–195.

[114] L. Damnjanovic. The formal specification of level 1a of GKS. *Computer Graphics Forum*, 10:11–25, 1991.

[115] R. S. M. de Barros. Deriving relational database programs from formal specifications. In Naftalin et al. [302], pages 703–723.

[116] R. S. M. de Barros and D. J. Harper. Formal development of relational database applications. In D. J. Harper and M. C. Norrie, editors, *Specifications of Database Systems, Glasgow 1991*, Workshops in Computing, pages 21–43. Springer-Verlag, 1992.

Zc, a Z-like formalism, is used.

[117] R. S. M. de Barros and D. J. Harper. A method for the specification of relational database applications. In Nicholls [317], pages 261–286.

[118] B. Dehbonei and F. Mejia. Formal methods in the railways signalling industry. In Naftalin et al. [302], pages 26–34.

[119] N. Delisle and D. Garlan. Formally specifying electronic instruments. In *Proc. Fifth International Workshop on Software Specification and Design*. IEEE Computer Society, May 1989. Also published in ACM SIGSOFT Software Engineering Notes 14(3).

[120] N. Delisle and D. Garlan. A formal specification of an oscilloscope. *IEEE Software*, 7(5):29–36, September 1990.

Unlike most work on the application of formal methods, this research uses formal methods to gain insight into system architecture. The context for this case study is electronic instrument design.

[121] Department of Trade and Industry. *Product Standards: Machinery*. HMSO Publications Centre, PO Box 276, London SW8 5DT, UK, April 1993. Covers *The Supply of Machinery (Safety) Regulations* 1992 (S.I. 1992/3073).

[122] P. Deransart. Prolog standardisation: the usefulness of a formal specification. Posted on `comp.lang.prolog`, `comp.specification` and `comp.software-eng` electronic USENET newsgroups, October 1992.

[123] R. Di Giovanni and P. L. Iachini. HOOD and Z for the development of complex systems. In Bjørner et al. [32], pages 262–289.

[124] A. J. J. Dick, P. J. Krause, and J. Cozens. Computer aided transformation of Z into Prolog. In Nicholls [313], pages 71–85.

[125] A. Diller. Z and Hoare logics. In Nicholls [317], pages 59–76.

[126] A. Diller. *Z: An Introduction to Formal Methods*. John Wiley & Sons, Chichester, UK, 2nd edition, 1994.

This book offers a comprehensive tutorial to Z from the practical viewpoint. Many natural deduction style proofs are presented and exercises are included. Z as defined in the 2nd edition of *The Z Notation* [381] is used throughout.

Contents: Tutorial introduction; Methods of reasoning; Case studies; Specification animation; Reference manual; Answers to exercises; Glossaries of terms and symbols; Bibliography.

[127] A. Diller and R. Docherty. Z and abstract machine notation: A comparison. In Bowen and Hall [60], pages 250–263.

[128] A. J. Dix. *Formal Methods for Interactive Systems*. Computers and People Series. Academic Press, 1991.

[129] A. J. Dix, J. Finlay, G. D. Abowd, and R. Beale. *Human-Computer Interaction*. Prentice Hall International, 1993.

[130] V. Doma and R. Nicholl. EZ: A system for automatic prototyping of Z specifications. In Prehn and Toetenel [338], pages 189–203.

[131] C. Draper. Practical experiences of Z and SSADM. In Bowen and Nicholls [70], pages 240–251.

[132] D. A. Duce. Formal methods in computer graphics. *Computer Graphics Forum*, 10:359–361, 1989.

[133] D. A. Duce, D. J. Duke, P. J. W. ten Hagen, and G. J. Reynolds. PREMO - an initial approach to a formal definition. *Computer Graphics Forum*, 13(3):C–393–C–406, 1994.

PREMO (Presentation Environments for Multimedia Objects) is a work item proposal by the ISO/IEC JTC11/SC24 committee, which is responsible for international standardization in the area of computer graphics and image processing.

[134] D. J. Duke. Enhancing the structures of Z specifications. In Nicholls [317], pages 329–351.

[135] D. J. Duke and R. Duke. Towards a semantics for Object-Z. In Bjørner et al. [32], pages 244–261.

[136] D. J. Duke and M. D. Harrison. Event model of human-system interaction. *IEE/BCS Software Engineering Journal*, 10(1):3–12, January 1995.

[137] D. J. Duke and M. D. Harrison. Mapping user requirements to implementations. *IEE/BCS Software Engineering Journal*, 10(1):13–20, January 1995.

[138] R. Duke and D. J. Duke. Aspects of object-oriented formal specification. In *Proc. 5th Australian Software Engineering Conference (ASWEC'90)*, pages 21–26, 1990.

[139] R. Duke, I. J. Hayes, P. King, and G. A. Rose. Protocol specification and verification using Z. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification VIII*, pages 33–46. Elsevier Science Publishers (North-Holland), 1988.

[140] R. Duke, P. King, G. A. Rose, and G. Smith. The Object-Z specification language. In T. Korson, V. Vaishnavi, and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 5*, pages 465–483. Prentice Hall, 1991.

[141] R. Duke, P. King, G. A. Rose, and G. Smith. The Object-Z specification language: Version 1. Technical Report 91-1, Department of Computer Science, University of Queensland, St. Lucia 4072, Australia, April 1991.

The most complete (and currently the standard) reference on Object-Z. It has been reprinted by ISO JTC1 WG7 as document number 372. A condensed version of this report was published as [140].

[142] R. Duke, G. A. Rose, and A. Lee. Object-oriented protocol specification. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Protocol Specification, Testing, and Verification X*, pages 325–338. Elsevier Science Publishers (North-Holland), 1990.

[143] R. Duke and G. Smith. Temporal logic and Z specifications. *Australian Computer Journal*, 21(2):62–69, May 1989.

[144] N. J. Dunlop. Formal specification of a windowing system. Msc thesis, Oxford University Computing Laboratory, UK, 1988.

[145] M. Dyer. *The Cleanroom Approach to Quality Software Development*. Series in Software Engineering Practice. John Wiley & Sons, 1992.

[146] M. Engel. Specifying real-time systems with Z and the Duration Calculus. In Bowen and Hall [60], pages 282–294.

[147] A. S. Evans. Specifying & verifying concurrent systems using Z. In Naftalin et al. [302], pages 366–400.

[148] A. S. Evans. Visualising concurrent Z specifications. In Bowen and Hall [60], pages 269–281.

[149] J. R. Farr. A formal specification of the Transputer instruction set. Master's thesis, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK, September 1987.

A partial specification of the Inmos Transputer instruction set.

[150] P. C. Fencott, A. J. Galloway, M. A. Lockyer, S. J. O'Brien, and S. Pearson. Formalising the semantics of Ward/Mellor SA/RT essential models using a process algebra. In Naftalin et al. [302], pages 681–702.

[151] N. E. Fenton and D. Mole. A note on the use of Z for flowgraph transformation. *Information and Software Technology*, 30(7):432–437, 1988.

[152] E. Fergus and D. C. Ince. Z specifications and modal logic. In P. A. V. Hall, editor, *Proc. Software Engineering 90*, volume 1 of *British Computer Society Conference Series*. Cambridge University Press, 1990.

[153] C. J. Fidge. Specification and verification of real-time behaviour using Z and RTL. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science, pages 393–410. Springer-Verlag, 1992.

[154] C. J. Fidge. Real-time refinement. In Woodcock and Larsen [435], pages 314–331.

[155] C. J. Fidge. Adding real time to formal program development. In Naftalin et al. [302], pages 618–638.

[156] C. J. Fidge. Proof obligations for real-time refinement. In Till [405], pages 279–305.

[157] M. Flynn, T. Hoverd, and D. Brazier. Formaliser – an interactive support tool for Z. In Nicholls [313], pages 128–141.

[158] J. D. Foley et al. *Computer Graphics: Principles and Practice*. Addison-Wesley, 1990.

[159] R. B. France and S. L. Gerhart, editors. *Proc. Workshop on Industrial-strength Formal Specification Techniques*. IEEE Computer Society Press, 1995.

[160] N. E. Fuchs. Specifications are (preferably) executable. *IEE/BCS Software Engineering Journal*, 7(5):323–334, September 1992.

[161] P. H. B. Gardiner, P. J. Lupton, and J. C. P. Woodcock. A simpler semantics for Z. In Nicholls [315], pages 3–11.

[162] D. Garlan. Integrating formal methods into a professional master of software engineering program. In Bowen and Hall [60], pages 71–85.

[163] D. Garlan. Making formal methods effective for professional software engineers. *Information and Software Technology*, 37(5-6):261–268, May–June 1995.

Revised version of [162].

[164] D. Garlan and N. Delisle. Formal specifications as reusable frameworks. In Bjørner et al. [32], pages 150–163.

[165] D. Garlan and N. Delisle. Formal specification of an architecture for a family of instrumentation systems. In Hinchey and Bowen [213], pages 55–72.

[166] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In Prehn and Toetenel [338], pages 31–45.

[167] S. L. Gerhart. Applications of formal methods: Developing virtuoso software. *IEEE Software*, 7(5):6–10, September 1990.

This is an introduction to a special issue on Formal Methods with an emphasis on Z in particular. It was published in conjunction with special Formal Methods issues of *IEEE Transactions on Software Engineering* and *IEEE Computer*. See also [120, 184, 303, 379, 421].

[168] S. L. Gerhart, D. Craigen, and T. J. Ralston. Observations on industrial practice using formal methods. In *Proc. 15th International Conference on Software Engineering (ICSE), Baltimore, Maryland, USA*, May 1993.

[169] S. L. Gerhart, D. Craigen, and T. J. Ralston. Experience with formal methods in critical systems. *IEEE Software*, 11(1):21–28, January 1994.

Several commercial and exploratory cases in which Z features heavily are briefly presented on page 24. See also [250].

[170] W. W. Gibbs. Software's chronic crisis. *Scientific American*, 271(3):86–95, September 1994.

[171] S. Gilmore. Correctness-oriented approaches to software development. Technical Report ECS-LFCS-91-147 (also CST-76-91), Department of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ, UK, 1991.

This PhD thesis provides a critical evaluation of Z, VDM and algebraic specifications.

[172] R. B. Gimson. The formal documentation of a Block Storage Service. Technical Monograph PRG-62, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK, August 1987.

[173] R. B. Gimson and C. C. Morgan. Ease of use through proper specification. In D. A. Duce, editor, *Distributed Computing Systems Programme*. Peter Peregrinus, London, 1984.

[174] R. B. Gimson and C. C. Morgan. The Distributed Computing Software project. Technical Monograph PRG-50, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK, July 1985.

[175] R. Gnatz. An algebraic approach to the standardization and certification of graphics software. *Computer Graphics Forum*, 2(2/3):153–166, 1983.

[176] J. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, Menlo Park, California, USA, August 1988.

[177] H. S. Goodman. Animating Z specifications in Haskell using a monad. Technical Report CSR-93-10, School of Computer Science, University of Birmingham, Birmingham B15 2TT, UK, November 1993.

[178] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: A theorem proving environment for Higher Order Logic*. Cambridge University Press, 1993.

[179] R. Gotzhein. Specifying open distributed systems with Z. In Bjørner et al. [32], pages 319–339.

[180] A. M. Gravell. Minimisation in formal specification and design. In Nicholls [313], pages 32–45.

[181] A. M. Gravell. What is a good formal specification? In Nicholls [315], pages 137–150.

[182] G. Guiho and C. Hennebert. SACEM software validation. In *Proc. 12th International Conference on Software Engineering (ICSE)*, pages 186–191. IEEE Computer Society Press, March 1990.

[183] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in five easy pieces. SRC Report 5, DEC Systems Research Center, Palo Alto, California, USA, 1985.

[184] J. A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

Formal methods are difficult, expensive, and not widely useful, detractors say. Using a case study and other real-world examples, this article challenges such common myths. See also [67].

[185] J. A. Hall. Using Z as a specification calculus for object-oriented systems. In Bjørner et al. [32], pages 290–318.

[186] J. A. Hall. Specifying and interpreting class hierarchies in Z. In Bowen and Hall [60], pages 120–138.

[187] J. G. Hall and J. A. McDermid. Towards a Z method: Axiomatic specification in Z. In Bowen and Hall [60], pages 213–229.

[188] P. A. V. Hall. Towards testing with respect to formal specification. In *Proc. Second IEE/BCS Conference on Software Engineering*, number 290 in Conference Publication, pages 159–163. IEE/BCS, July 1988.

[189] U. Hamer and J. Peleska. Z applied to the A330/340 CICS cabin communication system. In Hinchey and Bowen [213], pages 253–284.

[190] V. Hamilton. The use of Z within a safety-critical software system. In Hinchey and Bowen [213], pages 357–374.

[191] J. A. R. Hammond. Producing Z specifications from object-oriented analysis. In Bowen and Hall [60], pages 316–336.

[192] M. D. Harrison. Engineering human-error tolerant software. In Nicholls [317], pages 191–204.

[193] M. D. Harrison and H. Thimbleby, editors. *Formal Methods in HCI*. Cambridge University Press, 1990.

[194] W. Hasselbring. Animation of Object-Z specifications with a set-oriented prototyping language. In Bowen and Hall [60], pages 337–356.

[195] W. Hasselbring. *Prototyping Parallel Algorithms in a Set-Oriented Language*. Dissertation (University of Dortmund, Dept. Computer Science). Verlag Dr. Kovac, Hamburg, Germany, 1994.

This book presents the design and implementation of an approach to prototyping parallel algorithms with ProSet-Linda. The presented approach to designing and implementing ProSet-Linda relies on the use of the formal specification language Object-Z and the prototyping language ProSet itself.

[196] H. P. Haughton. Using Z to model and analyse safety and liveness properties of communication protocols. *Information and Software Technology*, 33(8):575–580, October 1991.

[197] I. J. Hayes. Specification directed module testing. *IEEE Transactions on Software Engineering*, 12(1):124–133, January 1986.

[198] I. J. Hayes. A generalisation of bags in Z. In Nicholls [313], pages 113–127.

[199] I. J. Hayes. Specifying physical limitations: A case study of an oscilloscope. Technical Report 167, Department of Computer Science, University of Queensland, St. Lucia 4072, Australia, July 1990.

[200] I. J. Hayes. Interpretations of Z schema operators. In Nicholls [315], pages 12–26.

[201] I. J. Hayes. Multi-relations in Z: A cross between multi-sets and binary relations. *Acta Informatica*, 29(1):33–62, February 1992.

[202] I. J. Hayes. VDM and Z: A comparative case study. *Formal Aspects of Computing*, 4(1):76–99, 1992.

[203] I. J. Hayes, editor. *Specification Case Studies*. Prentice Hall International Series in Computer Science, 2nd edition, 1993.

This is a revised edition of the first ever book on Z, originally published in 1987; it contains substantial changes to every chapter. The notation has been revised to be consistent with *The Z Notation: A Reference Manual* by Mike Spivey [381]. The CAVIAR chapter has been extensively changed to make use of a form of modularization.

Divided into four sections, the first provides tutorial examples of specifications, the second is devoted to the area of software engineering, the third covers distributed computing, analyzing the role of mathematical specification, and the fourth part covers the IBM CICS transaction processing system. Appendices include comprehensive glossaries of the Z mathematical and schema notation. The book will be of interest to the professional software engineer involved in designing and specifying large software projects.

The other contributors are W. Flinn, R. B. Gimson, S. King, C. C. Morgan, I. H. Sørensen and B. A. Sufrin.

[204] I. J. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *IEE/BCS Software Engineering Journal*, 4(6):330–338, November 1989.

[205] I. J. Hayes, C. B. Jones, and J. E. Nicholls. Understanding the differences between VDM and Z. *FACS Europe*, Series I, 1(1):7–30, Autumn 1993.

Also available as Technical Report UMCS-93-8-1, Department of Computer Science, University of Manchester, UK, 1993.

[206] I. J. Hayes and L. Wildman. Towards libraries for Z. In Bowen and Nicholls [70], pages 9–36.

[207] He Jifeng, C. A. R. Hoare, M. Fränzle, M. Müller-Ulm, E.-R. Olderog, M. Schenke, A. P. Ravn, and H. Rischel. Provably correct systems. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real Time and Fault Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 288–335. Springer-Verlag, 1994.

[208] He Jifeng, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *Proc. ESOP 86*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer-Verlag, 1986.

[209] D. Heath, D. Allum, and L. Dunckley. *Introductory Logic and Formal Methods*. A. Waller, Henley-on-Thames, UK, 1994.

[210] B. Hepworth. ZIP: A unification initiative for Z standards, methods and tools. In Nicholls [313], pages 253–259.

[211] B. Hepworth and D. Simpson. The ZIP project. In Nicholls [315], pages 129–133.

[212] J. V. Hill. Software development methods in practice. In *Microprocessor Based Protection Systems*. Elsevier, 1991.

[213] M. G. Hinchey and J. P. Bowen, editors. *Applications of Formal Methods*. Prentice Hall International Series in Computer Science, 1995.

A collection on industrial examples of the use of formal methods. Chapters relevant to Z include [97, 109, 165, 189, 190, 214, 277].

[214] M. G. Hinchey and J. P. Bowen. Applications of formal methods FAQ. In *Applications of Formal Methods* [213], pages 1–15.

[215] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.

[216] C. A. R. Hoare, He Jifeng, J. P. Bowen, and P.K. Pandya. An algebraic approach to verifiable compiling specification and prototyping of the ProCoS level 0 programming language. In *ESPRIT '90 Conference Proceedings*, pages 804–818. Kluwer Academic Publishers, 1990.

[217] C. A. R. Hoare and I. Page. Hardware and software: Closing the gap. *Transputer Communications*, 2(2):69–90, June 1994.

[218] I. S. C. Houston and S. King. CICS project report: Experiences and results from the use of Z in IBM. In Prehn and Toetenel [338], pages 588–596.

[219] P. L. Iachini. Operation schema iterations. In Nicholls [315], pages 50–57.

[220] IEEE. IEEE standard glossary of software engineering terminology. In *IEEE Software Engineering Standards Collection*. Elsevier Applied Science, 1991.

[221] V. Illingworth, editor. *Dictionary of Computing*. Oxford University Press, 3rd edition, 1990.

[222] M. Imperato. *An Introduction to Z*. Chartwell-Bratt, 1991.

Contents: Introduction; Set theory; Logic; Building Z specifications; Relations; Functions; Sequences; Bags; Advanced Z; Case study: a simple banking system.

[223] D. C. Ince. *An Introduction to Discrete Mathematics, Formal System Specification and Z*. Oxford Applied Mathematics and Computing Science Series. Oxford University Press, 2nd edition, 1993.

[224] INMOS Limited. *Transputer Instruction Set: A compiler writer's guide*. Prentice Hall, 1988.

[225] D. Jackson. Abstract model checking of infinite specifications. In Naftalin et al. [302], pages 519–531.

[226] J. Jacky. Formal specifications for a clinical cyclotron control system. *ACM SIGSOFT Software Engineering Notes*, 15(4):45–54, September 1990.

[227] J. Jacky. Formal specification and development of control system input/output. In Bowen and Nicholls [70], pages 95–108.

[228] J. Jacky. Specifying a safety-critical control system in Z. In Woodcock and Larsen [435], pages 388–402.

Revised version in [229].

[229] J. Jacky. Specifying a safety-critical control system in Z. *IEEE Transactions on Software Engineering*, 21(2):99–106, 1995.

Revised version of [228].

[230] J. Jacob. The varieties of refinements. In Morris and Shaw [301], pages 441–455.

[231] C. W. Johnson. Using Z to support the design of interactive safety-critical systems. *IEE/BCS Software Engineering Journal*, 10(2):49–60, March 1995.

[232] M. Johnson and P. Sanders. From Z specifications to functional implementations. In Nicholls [313], pages 86–112.

[233] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International Series in Computer Science, 2nd edition, 1990.

[234] C. B. Jones. Interference revisited. In Nicholls [315], pages 58–73.

[235] C. B. Jones, R. C. Shaw, and T. Denvir, editors. *5th Refinement Workshop*, Workshop in Computing. Springer-Verlag, 1992.

The workshop was held at Lloyd's Register, London, UK, 8–10 January 1992. See [366].

[236] R. B. Jones. ICL ProofPower. *BCS-FACS FACTS*, Series III, 1(1):10–13, Winter 1992.

[237] D. Jordan, J. A. McDermid, and I. Toyn. CADiZ – computer aided design in Z. In Nicholls [315], pages 93–104.

[238] M. B. Josephs. The data refinement calculator for Z specifications. *Information Processing Letters*, 27(1):29–33, 1988.

[239] M. B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3:9–18, 1988.

A theoretical paper on combining features of CSP and Z.

[240] M. B. Josephs. Specifying reactive systems in Z. Technical Report PRG-TR-19-91, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK, July 1991.

[241] M. B. Josephs and D. Redmond-Pyle. Entity-relationship models expressed in Z: A synthesis of structured and formal methods. Technical Report PRG-TR-20-91, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK, July 1991.

[242] D. H. Kemp. Specification of Viper1 in Z. Memorandum no. 4195, RSRE, Ministry of Defence, Malvern, Worcestershire, UK, October 1988.

[243] D. H. Kemp. Specification of Viper2 in Z. Memorandum no. 4217, RSRE, Ministry of Defence, Malvern, Worcestershire, UK, October 1988.

[244] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Software Series. Prentice Hall, 2nd edition, 1988.

[245] P. King. Printing Z and Object-Z LaTeX documents. Department of Computer Science, University of Queensland, May 1990.

A description of a Z style option '`oz.sty`', an extended version of Mike Spivey's '`zed.sty`' [378], for use with the LaTeX document preparation system [251]. It is particularly useful for printing Object-Z documents [87, 135].

[246] S. King. Z and the refinement calculus. In Bjørner et al. [32], pages 164–188.

Also published as Technical Monograph PRG-79, Oxford University Computing Laboratory, February 1990.

[247] S. King. The use of Z in the restructure of IBM CICS. In I. J. Hayes, editor, *Specification Case Studies*, International Series in Computer Science, chapter 14, pages 202–213. Prentice Hall, Hemel Hempstead, Hertfordshire, UK, 2nd edition, 1993.

[248] S. King and I. H. Sørensen. Specification and design of a library system. In McDermid [284].

[249] J. C. Knight and D. M. Kienzle. Preliminary experience using Z to specify a safety-critical system. In Bowen and Nicholls [70], pages 109–118.

[250] J. C. Knight and B. Littlewood. Critical task of writing dependable software. *IEEE Software*, 11(1):16–20, January 1994.

Guest editors' introduction to a special issue of *IEEE Software* on *Safety-Critical Systems*. A short section on formal methods mentions several Z books on page 18. See also [169].

[251] L. Lamport. *LaTeX User's Guide & Reference Manual: A document preparation system*. Addison-Wesley Publishers Ltd., 2nd edition, 1993.

Z specifications may be produced using the document preparation system LaTeX together with a special LaTeX style option. The most widely used style files are `fuzz.sty` [380], `zed.sty` [378] and `oz.sty` [245].

[252] L. Lamport. TLZ. In Bowen and Hall [60], pages 267–268. Abstract.

[253] K. C. Lano. Z^{++}, an object-orientated extension to Z. In Nicholls [315], pages 151–172.

[254] K. C. Lano. Refinement in object-oriented specification languages. In Till [405], pages 236–259.

[255] K. C. Lano and P. T. Breuer. From programs to Z specifications. In Nicholls [313], pages 46–70.

[256] K. C. Lano and H. P. Haughton. An algebraic semantics for the specification language Z^{++}. In *Proc. Algebraic Methodology and Software Technology Conference (AMAST '91)*. Springer-Verlag, 1992.

[257] K. C. Lano and H. P. Haughton. Reasoning and refinement in object-oriented specification languages. In O. L. Madsen, editor, *ECOOP '92: European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 78–97. Springer-Verlag, 1992.

[258] K. C. Lano and H. P. Haughton, editors. *Object Oriented Specification Case Studies*. Object Oriented Series. Prentice Hall International, 1993.

Contents: Chapters introducing object oriented methods, object oriented formal specification and the links between formal and structured object-oriented techniques; seven case studies in particular object oriented formal methods, including:

The Unix Filing System: A MooZ Specification; An Object-Z Specification of a Mobile Phone System; Object-oriented Specification in VDM^{++}; Specifying a Concept-recognition System in Z^{++}; Specification in OOZE; Refinement in Fresco; SmallVDM: An Environment for Formal Specification and Prototyping in Smalltalk.

A glossary, index and bibliography are also included. The contributors are some of the leading figures in the area, including the developers of the above methods and languages: Silvio Meira, Gordon Rose, Roger Duke, Antonio Alencar, Joseph Goguen, Alan Wills, Cassio Souza dos Santos, Ana Cavalcanti.

[259] K. C. Lano and H. P. Haughton. Reuse and adaptation of Z specifications. In Bowen and Nicholls [70], pages 62–90.

[260] K. C. Lano and H. P. Haughton. Formal development in B Abstract Machine Notation. *Information and Software Technology*, 37(5-6):303–316, May–June 1995.

[261] D. Learmount. Airline safety review: human factors. *Flight International*, 142(4238):30–33, 22–28 July 1992.

[262] D. Lightfoot. *Formal Specification using Z*. Macmillan, 1991.

Contents: Introduction; Sets in Z; Using sets to describe a system – a simple example; Logic: propositional calculus; Example of a Z specification document; Logic: predicate calculus; Relations; Functions; A seat allocation system; Sequences; An example of sequences – the aircraft example again; Extending a specification; Collected notation; Books on formal specification; Hints on creating specifications; Solutions to exercises. Also available in French.

[263] P. A. Lindsay. Reasoning about Z specifications: A VDM perspective. Technical Report 93-20, Department of Computer Science, University of Queensland, St. Lucia 4072, Australia, October 1993.

[264] P. A. Lindsay. On transferring VDM verification techniques to Z. In Naftalin et al. [302], pages 190–213.

Also available as Technical Report 94-10, Department of Computer Science, University of Queensland, 1994.

[265] P. A. Lindsay and E. van Keulen. Case studies in the verification of specifications in VDM and Z. Technical Report 94-3, Department of Computer Science, University of Queensland, St. Lucia 4072, Australia, March 1994.

[266] M. Love. Animating Z specifications in SQL*Forms3.0. In Bowen and Nicholls [70], pages 294–306.

[267] P. J. Lupton. Promoting forward simulation. In Nicholls [315], pages 27–49.

[268] K. Kumar M. D. Faser and V. K. Vaishnavi. Strategies for incorporating formal specifications in software development. *Communications of the ACM*, 37(10):74–86, October 1994.

[269] R. Macdonald. Z usage and abusage. Report no. 91003, RSRE, Ministry of Defence, Malvern, Worcestershire, UK, February 1991.

This paper presents a miscellany of observations drawn from experience of using Z, shows a variety of techniques for expressing certain class of idea concisely and clearly, and alerts the reader to certain pitfalls which may trap the unwary.

[270] D. MacKenzie. Computers, formal proof, and the law courts. *Notices of the American Mathematical Society*, 39(9):1066–1069, November 1992.

[271] B. P. Mahony and I. J. Hayes. A case-study in timed refinement: A central heater. In Morris and Shaw [301], pages 138–149.

[272] B. P. Mahony and I. J. Hayes. A case-study in timed refinement: A mine pump. *IEEE Transactions on Software Engineering*, 18(9):817–826, September 1992.

[273] B. P. Mahony, C. Millerchip, and I. J. Hayes. A boiler control system: A case-study in timed refinement. Technical report, Department of Computer Science, University of Queensland, St. Lucia 4072, Australia, 23 June 1993.

A specification and top-level design of a steam generating boiler system is presented as an example of the formal development of a real-time system.

[274] K. C. Mander and F. Polack. Rigorous specification using structured systems analysis and Z. *Information and Software Technology*, 37(5-6):285–291, May–June 1995.

Revised version of [332].

[275] A. Martin. Encoding W: A logic for Z in 2OBJ. In Woodcock and Larsen [435], pages 462–481.

[276] P. Mataga and P. Zave. Formal specification of telephone features. In Bowen and Hall [60], pages 29–50.

[277] P. Mataga and P. Zave. Multiparadigm specification of an AT&T switching system. In Hinchey and Bowen [213], pages 375–398.

[278] P. Mataga and P. Zave. Using Z to specify telephone features. *Information and Software Technology*, 37(5-6):277–283, May–June 1995.

Revised version of [276].

[279] I. Maung and J. R. Howse. Introducing Hyper-Z – a new approach to object orientation in Z. In Bowen and Nicholls [70], pages 149–165.

[280] M. D. May. Use of formal methods by a silicon manufacturer. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, University of Texas at Austin Year of Programming Series, chapter 4, pages 107–129. Addison-Wesley Publishing Company, 1990.

[281] M. D. May, G. Barrett, and D. E. Shepherd. Designing chips that work. In C. A. R. Hoare and M. J. C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, pages 3–19. Prentice Hall International Series in Computer Science, 1992.

[282] M. D. May and D. E. Shepherd. Verification of the IMS T800 microprocessor. In *Proc. Electronic Design Automation*, pages 605–615, London, UK, September 1987.

[283] J. A. McDermid. Special section on Z. *IEE/BCS Software Engineering Journal*, 4(1):25–72, January 1989.

A special issue on Z, introduced and edited by Prof. J. A. McDermid. See also [44, 92, 377, 428].

[284] J. A. McDermid, editor. *The Theory and Practice of Refinement: Approaches*

to the Formal Development of Large-Scale Software Systems. Butterworth Scientific, 1989.

This book contains papers from the 1st Refinement Workshop held at the University of York, UK, 7–8 January 1988. Z-related papers include [248, 308].

[285] J. A. McDermid. Formal methods: Use and relevance for the development of safety critical systems. In P. A. Bennett, editor, *Safety Aspects of Computer Control*. Butterworth-Heinemann, Oxford, UK, 1993.

This paper discusses a number of formal methods and summarizes strengths and weaknesses in safety critical applications; a major safety-related example is presented in Z.

[286] M. A. McMorran and S. Powell. *Z Guide for Beginners*. Blackwell Scientific, 1993.

[287] S. L. Meira and A. L. C. Cavalcanti. Modular object-oriented Z specifications. In Nicholls [315], pages 173–192.

[288] B. Meyer. On formalism in specifications. *IEEE Software*, 2(1):6–26, January 1985.

[289] C. Minkowitz, D. Rann, and J. H. Turner. A C++ library for implementing specifications. In France and Gerhart [159], pages 61–75.

[290] V. Mišić, D. Velašević, and B. Lazarević. Formal specification of a data dictionary for an extended ER data model. *The Computer Journal*, 35(6):611–622, December 1992.

[291] MoD. The procurement of safety critical software in defence equipment. Interim Defence Standard 00-55, Ministry of Defence, Directorate of Standardization, Kentigern House, 65 Brown Street, Glasgow G2 8EX, UK, 1991.

[292] J. D. Moffett and M. S. Sloman. A case study representing a model: To Z or not to Z? In Nicholls [315], pages 254–268.

[293] B. Q. Monahan and R. C. Shaw. Model-based specifications. In J. A. McDermid, editor, *Software Engineer's Reference Book*, chapter 21. Butterworth-Heinemann, Oxford, UK, 1991.

This chapter contains a case study in Z, followed by a discussion of the respective trade-offs in specification between Z and VDM.

[294] C. C. Morgan. Data refinement using miracles. *Information Processing Letters*, 26(5):243–246, January 1988.

[295] C. C. Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science, 2nd edition, 1994.

This book presents a rigorous treatment of most elementary program development techniques, including iteration, recursion, procedures, parameters, modules and data refinement.

[296] C. C. Morgan and J. W. Sanders. Refinement for Z. Programming Research Group, Oxford University Computing Laboratory, UK, 1988.

[297] C. C. Morgan and J. W. Sanders. Laws of the logical calculi. Technical Monograph PRG-78, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK, September 1989.

This document records some important laws of classical predicate logic. It is designed as a reservoir to be tapped by *users* of logic, in system development.

[298] C. C. Morgan and B. A. Sufrin. Specification of the Unix filing system. *IEEE Transactions on Software Engineering*, 10(2):128–142, March 1984.

[299] C. C. Morgan and T. Vickers, editors. *On the Refinement Calculus*. Formal Approaches to Computing and Information Technology series (FACIT). Springer-Verlag, 1994.

This book collects together the work accomplished at Oxford on the refinement calculus: the rigorous development, from state-based assertional specification, of executable imperative code.

[300] C. C. Morgan and J. C. P. Woodcock, editors. *3rd Refinement Workshop*, Workshops in Computing. Springer-Verlag, 1991.

The workshop was held at the IBM Laboratories, Hursley Park, UK, 9–11 January 1990. See [365].

[301] J. M. Morris and R. C. Shaw, editors. *4th Refinement Workshop*, Workshops in Computing. Springer-Verlag, 1991.

The workshop was held at Cambridge, UK, 9–11 January 1991. For Z related papers, see [19, 230, 271, 423, 430, 420].

[302] M. Naftalin, T. Denvir, and M. Bertran, editors. *FME'94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*. Formal Methods Europe, Springer-Verlag, 1994.

The 2nd FME Symposium was held at Barcelona, Spain, 24–28 October 1994. Z-related papers include [64, 103, 115, 147, 150, 155, 225, 264]. B-related papers include [118, 349, 395].

[303] K. T. Narayana and S. Dharap. Formal specification of a look manager. *IEEE Transactions on Software Engineering*, 16(9):1089–1103, September 1990.

A formal specification of the look manager of a dialog system is presented in Z. This deals with the presentation of visual aspects of objects and the editing of those visual aspects.

[304] K. T. Narayana and S. Dharap. Invariant properties in a dialog system. *ACM SIGSOFT Software Engineering Notes*, 15(4):67–79, September 1990.

[305] T. C. Nash. Using Z to describe large systems. In Nicholls [313], pages 150–178.

[306] C. Neesham. Safe conduct. *Computing*, pages 18–20, 12 November 1992.

[307] Ph. W. Nehlig and D. A. Duce. GKS-9x: The design output primitive, an approach to specification. *Computer Graphics Forum*, 13(3):C–381–C–392, 1994.

[308] D. S. Neilson. Hierarchical refinement of a Z specification. In McDermid [284].

[309] D. S. Neilson. From Z to C: Illustration of a rigorous development method. Technical Monograph PRG-101, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK, 1990.

[310] D. S. Neilson. Machine support for Z: The zedB tool. In Nicholls [315], pages 105–128.

[311] D. S. Neilson and D. Prasad. zedB: A proof tool for Z built on B. In Nicholls [317], pages 243–258.

[312] W. M. Newman and R. F. Sproull. *Principles of Interactive Computer Graphics*. Computer Science Series. McGraw-Hill, 2nd edition, 1981.

[313] J. E. Nicholls, editor. *Z User Workshop, Oxford 1989*, Workshops in Computing. Springer-Verlag, 1990.

Proceedings of the Fourth Annual Z User Meeting, Wolfson College & Rewley House, Oxford, UK, 14–15 December 1989. Published in collaboration with

the British Computer Society. For the opening address see [324]. For individual papers, see [28, 81, 82, 101, 124, 157, 180, 198, 210, 232, 255, 305, 328, 370, 382, 418].

[314] J. E. Nicholls. A survey of Z courses in the UK. In *Z User Workshop, Oxford 1990* [315], pages 343–350.

[315] J. E. Nicholls, editor. *Z User Workshop, Oxford 1990*, Workshops in Computing. Springer-Verlag, 1991.

Proceedings of the Fifth Annual Z User Meeting, Lady Margaret Hall, Oxford, UK, 17–18 December 1990. Published in collaboration with the British Computer Society. For individual papers, see [20, 76, 84, 98, 161, 181, 200, 211, 219, 234, 237, 253, 287, 292, 310, 314, 322, 344, 363, 419, 442]. The proceedings also includes an *Introduction and Opening Remarks*, a *Selected Z Bibliography*, a selection of posters and information on Z tools.

[316] J. E. Nicholls. Domains of application for formal methods. In *Z User Workshop, York 1991* [317], pages 145–156.

[317] J. E. Nicholls, editor. *Z User Workshop, York 1991*, Workshops in Computing. Springer-Verlag, 1992.

Proceedings of the Sixth Annual Z User Meeting, York, UK. Published in collaboration with the British Computer Society. For individual papers, see [15, 22, 117, 86, 125, 134, 192, 311, 316, 333, 352, 371, 402, 410, 433, 446].

[318] J. E. Nicholls. Plain guide to the Z base standard. In Bowen and Nicholls [70], pages 52–61.

[319] C. J. Nix and B. P. Collins. The use of software engineering, including the Z notation, in the development of CICS. *Quality Assurance*, 14(3):103–110, September 1988.

[320] A. Norcliffe and G. Slater. *Mathematics for Software Construction*. Series in Mathematics and its Applications. Ellis Horwood, 1991.

Contents: Why mathematics; Getting started: sets and logic; Developing ideas: schemas; Functions; Functions in action; A real problem from start to finish: a drinks machine; Sequences; Relations; Generating programs from specifications: refinement; The role of proof; More examples of specifications; Concluding remarks; Answers to exercises.

[321] A. Norcliffe and S. Valentine. Z readers video course. PAVIC Publications, 1992. Sheffield Hallam University, 33 Collegiate Crescent, Sheffield S10 2BP, UK.

Video-based Training Course on the Z Specification Language. The course consists of 5 videos, each of approximately one hour duration, together with supporting texts and case studies.

[322] A. Norcliffe and S. H. Valentine. A video-based training course in reading Z specifications. In Nicholls [315], pages 337–342.

[323] G. Normington. Cleanroom and Z. In Bowen and Nicholls [70], pages 281–293.

[324] B. Oakley. The state of use of formal methods. In Nicholls [313], pages 1–5.

A record of the opening address at ZUM'89.

[325] A. Palay et al. The Andrew toolkit: An overview. In *Proc. Winter USENIX Technical Conference, Dallas, USA*, pages 9–21, 1988.

[326] C. E. Parker. Z tools catalogue. ZIP project report ZIP/BAe/90/020, British

Aerospace, Software Technology Department, Warton PR4 1AX, UK, May 1991.

[327] M. Peltu. Safety in numbers. *Computing*, page 34, 12 January 1995.

[328] M. Phillips. CICS/ESA 3.1 experiences. In Nicholls [313], pages 179–185.

Z was used to specify 37,000 lines out of 268,000 lines of code in the IBM CICS/ESA 3.1 release. The initial development benefit from using Z was assessed as being a 9% improvement in the *total development cost* of the release, based on the reduction of programmer days fixing problems.

[329] R. Pike. The Blit: a multiplexed graphics terminal. *AT&T Bell Laboratories Technical Journal*, 63(8, part 2):1607–1631, October 1984.

[330] M. Pilling, A. Burns, and K. Raymond. Formal specifications and proofs of inheritance protocols for real-time scheduling. *IEE/BCS Software Engineering Journal*, 5(5):263–279, September 1990.

[331] P. R. H. Place and K. C. Kang. Safety-critical software: Status report and annotated bibliography. Technical Report CMU/SEI-92-TR-5 & ESC-TR-93-182, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213, USA, June 1993.

[332] F. Polack and K. C. Mander. Software quality assurance using the SAZ method. In Bowen and Hall [60], pages 230–249.

[333] F. Polack, M. Whiston, and P. Hitchcock. Structured analysis – a draft method for writing Z specifications. In Nicholls [317], pages 261–286.

[334] F. Polack, M. Whiston, and K. C. Mander. The SAZ project: Integrating SSADM and Z. In Woodcock and Larsen [435], pages 541–557.

[335] J. P. Potocki de Montalk. Computer software in civil aircraft. *Microprocessors and Microsystems*, 17(1):17–23, January/February 1993.

[336] B. F. Potter, J. E. Sinclair, and D. Till. *An Introduction to Formal Specification and Z*. Prentice Hall International Series in Computer Science, 1991.

Contents: Formal specification in the context of software engineering; An informal introduction to logic and set theory; A first specification; The Z notation: the mathematical language, relations and functions, schemas and specification structure; A first specification revisited; Formal reasoning; From specification to program: data and operation refinement, operation decomposition; From theory to practice.

[337] B. F. Potter and D. Till. The specification in Z of gateway functions within a communications network. In *Proc. IFIP WG10.3 Conference on Distributed Processing*. Elsevier Science Publishers (North-Holland), October 1987.

[338] S. Prehn and W. J. Toetenel, editors. *VDM'91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991. Volume 1: Conference Contributions.

The 4th VDM-Europe Symposium was held at Noordwijkerhout, The Netherlands, 21–25 October 1991. Papers with relevance to Z include [14, 29, 110, 130, 166, 218, 413, 422, 445]. See also [339].

[339] S. Prehn and W. J. Toetenel, editors. *VDM'91: Formal Software Development Methods*, volume 552 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991. Volume 2: Tutorials.

Papers with relevance to Z include [4, 431]. See also [338].

[340] I. Pyle. Software engineers and the IEE. *Software Engineering Journal*, 1(2):66–68, March 1986.

[341] I. C. Pyle. *Developing Safety Systems: A Guide Using Ada*. Prentice Hall, 1991.

[342] G-H. B. Rafsanjani and S. J. Colwill. From Object-Z to C^{++}: A structural mapping. In Bowen and Nicholls [70], pages 166–179.

[343] RAISE Language Group. *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall International, 1992.

[344] G. P. Randell. Data flow diagrams and Z. In Nicholls [315], pages 216–227.

[345] G. P. Randell. Improving the translation from data flow diagrams into Z by incorporating the data dictionary. Report no. 92004, RSRE, Ministry of Defence, Malvern, Worcestershire, UK, January 1992.

[346] D. Rann, J. Turner, and J. Whitworth. *Z: A Beginner's Guide*. Chapman & Hall, London, 1994.

[347] B. Ratcliff. *Introducing Specification Using Z: A Practical Case Study Approach*. International Series in Software Engineering. McGraw-Hill, 1994.

[348] N. R. Reizer, G. D. Abowd, B. C. Meyers, and P. R. H. Place. Using formal methods for requirements specification of a proposed POSIX standard. In *IEEE International Conference on Requirements Engineering (ICRE'94)*, April 1994.

[349] B. Ritchie, J. Bicarregui, and H. P. Haughton. Experiences in using the abstract machine notation in a GKS case study. In Naftalin et al. [302], pages 93–104.

[350] P. Rose. A partial specification of the M68000 microprocessor. Master's thesis, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK, September 1987.

A partial specification of the Motorola 68000 instruction set.

[351] A. R. Ruddle. Formal methods in the specification of real-time, safety-critical control systems. In Bowen and Nicholls [70], pages 131–146.

[352] M. Saaltink. Z and Eves. In Nicholls [317], pages 223–242.

[353] H. Saiedian. The mathematics of computing. *Journal of Computer Science Education*, 3(3):203–221, 1992.

[354] A. C. A. Sampaio and S. L. Meira. Modular extensions to Z. In Bjørner et al. [32], pages 211–232.

[355] J. W. Sanders. Two topics in interface refinement. Digest of Papers, Refinement Workshop, King's Manor, University of York, UK, 1988.

[356] M. Satyanarayanan. The ITC project: An experiment in large-scale distributed personal computing. CMU Report CMU-ITC-035, Information Technology Center (ITC), Carnegie-Mellon University, USA, October 1984.

[357] R. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[358] R. Scheifler and J. Gettys. *The X Window System*. Digital Press, Bedford, MA, USA, 1989.

[359] R. Scheifler and J. Gettys. The X window system. In A. R. Meyer, J. V. Guttag, R. L. Rivest, and P. Szolovits, editors, *Research Directions in Computer Science: An MIT Perspective*, chapter 5, pages 75–92. The MIT Press, 1991.

[360] R. Scheifler, J. Gettys, and R. Newman. *X Window System: C Library and Protocol Reference*. Digital Press, Bedford, MA, USA, 1988.

[361] S. A. Schuman and D. H. Pitt. Object-oriented subsystem specification. In L. G. L. T. Meertens, editor, *Program Specification and Transformation*, pages 313–341. Elsevier Science Publishers (North-Holland), 1987.

[362] S. A. Schuman, D. H. Pitt, and P. J. Byers. Object-oriented process specification. In C. Rattray, editor, *Specification and Verification of Concurrent Systems*, pages 21–70. Springer-Verlag, 1990.

[363] L. T. Semmens and P. M. Allen. Using Yourdon and Z: An approach to formal specification. In Nicholls [315], pages 228–253.

[364] L. T. Semmens, R. B. France, and T. W. G. Docker. Integrated structured analysis and formal specification techniques. *The Computer Journal*, 35(6):600–610, December 1992.

[365] C. T. Sennett. Using refinement to convince: Lessons learned from a case study. In Morgan and Woodcock [300], pages 172–197.

[366] C. T. Sennett. Demonstrating the compliance of Ada programs with Z specifications. In Jones et al. [235].

[367] D. E. Shepherd. Verified microcode design. *Microprocessors and Microsystems*, 14(10):623–630, December 1990.

This article is part of a special feature on *Formal aspects of microprocessor design*, edited by H. S. M. Zedan. See also [45].

[368] D. E. Shepherd and G. Wilson. Making chips that work. *New Scientist*, 1664:61–64, May 1989.

A general article containing information on the formal development of the T800 floating-point unit for the transputer including the use of Z.

[369] D. Sheppard. *An Introduction to Formal Specification with Z and VDM*. International Series in Software Engineering. McGraw Hill, 1995.

[370] A. Smith. The Knuth-Bendix completion algorithm and its specification in Z. In Nicholls [313], pages 195–220.

[371] A. Smith. On recursive free types in Z. In Nicholls [317], pages 3–39.

[372] G. Smith. *An Object-Oriented Approach to Formal Specification*. PhD thesis, Department of Computer Science, University of Queensland, St. Lucia 4072, Australia, October 1992.

A detailed description of a version of Object-Z similar to (but not identical to) that in [141]. The thesis also includes a formalization of temporal logic history invariants and a fully-abstract model of classes in Object-Z.

[373] G. Smith. A object-oriented development framework for Z. In Bowen and Hall [60], pages 89–107.

[374] G. Smith and R. Duke. Modelling a cache coherence protocol using Object-Z. In *Proc. 13th Australian Computer Science Conference (ACSC-13)*, pages 352–361, 1990.

[375] P. Smith and R. Keighley. The formal development of a secure transaction mechanism. In Prehn and Toetenel [338], pages 457–476.

[376] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, January 1988.

Published version of 1985 DPhil thesis.

[377] J. M. Spivey. An introduction to Z and formal specifications. *IEE/BCS Software Engineering Journal*, 4(1):40–50, January 1989.

[378] J. M. Spivey. A guide to the `zed` style option. Oxford University Computing Laboratory, December 1990.

A description of the Z style option '`zed.sty`' for use with the LaTeX document preparation system [251].

[379] J. M. Spivey. Specifying a real-time kernel. *IEEE Software*, 7(5):21–28, September 1990.

This case study of an embedded real-time kernel shows that mathematical techniques have an important role to play in documenting systems and avoiding design flaws.

[380] J. M. Spivey. *The fuzz Manual*. Computing Science Consultancy, 34 Westlands Grove, Stockton Lane, York YO3 0EF, UK, 2nd edition, July 1992.

The manual describes a Z type-checker and '`fuzz.sty`' style option for LaTeX documents [251]. The package is compatible with the book, *The Z Notation: A Reference Manual* by the same author [381].

[381] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

This is a revised edition of the first widely available reference manual on Z originally published in 1989. The book provides a complete and definitive guide to the use of Z in specifying information systems, writing specifications and designing implementations. See also the draft Z standard [79].

Contents: Tutorial introduction; Background; The Z language; The mathematical tool-kit; Sequential systems; Syntax summary; Changes from the first edition; Glossary.

[382] J. M. Spivey and B. A. Sufrin. Type inference in Z. In Nicholls [313], pages 6–31.

Also published as [383].

[383] J. M. Spivey and B. A. Sufrin. Type inference in Z. In Bjørner et al. [32], pages 426–438.

[384] R. M. Stein. Safety by formal design. *BYTE*, page 157, August 1992.

[385] S. Stepney. *High Integrity Compilation: A Case Study*. Prentice Hall, 1993.

[386] S. Stepney and R. Barden. Annotated Z bibliography. *Bulletin of the European Association of Theoretical Computer Science*, 50:280–313, June 1993.

[387] S. Stepney, R. Barden, and D. Cooper, editors. *Object Orientation in Z*. Workshops in Computing. Springer-Verlag, 1992.

This is a collection of papers describing various OOZ approaches – Hall, ZERO, MooZ, Object-Z, OOZE, Schuman & Pitt, Z^{++}, ZEST and Fresco (an object-oriented VDM method) – in the main written by the methods' inventors, and all specifying the same two examples. The collection is a revised and expanded version of a ZIP report distributed at the 1991 Z User Meeting at York.

[388] S. Stepney, R. Barden, and D. Cooper. A survey of object orientation in Z. *IEE/BCS Software Engineering Journal*, 7(2):150–160, March 1992.

[389] S. Stepney and S. P. Lord. Formal specification of an access control system. *Software – Practice and Experience*, 17(9):575–593, September 1987.

[390] P. Stocks. *Applying formal methods to software testing*. PhD thesis, Department of Computer Science, University of Queensland, St. Lucia 4072, Australia, 1993.

[391] P. Stocks and D. A. Carrington. Deriving software test cases from formal specifications. In *6th Australian Software Engineering Conference*, pages 327–340, July 1991.

[392] P. Stocks and D. A. Carrington. Test template framework: A specification-based testing case study. In *Proc. International Symposium on Software Testing and Analysis (ISSTA'93)*, pages 11–18, June 1993.

Also available in a longer form as Technical Report UQCS-255, Department of Computer Science, University of Queensland.

[393] P. Stocks and D. A. Carrington. Test templates: A specification-based testing framework. In *Proc. 15th International Conference on Software Engineering*, pages 405–414, May 1993.

Also available in a longer form as Technical Report UQCS-243, Department of Computer Science, University of Queensland.

[394] P. Stocks, K. Raymond, D. Carrington, and A. Lister. Modelling open distributed systems in Z. *Computer Communications*, 15(2):103–113, March 1992.

In a special issue on the practical use of FDTs (Formal Description Techniques) in communications and distributed systems, edited by Dr. Gordon S. Blair.

[395] A. C. Storey and H. P. Haughton. A strategy for the production of verifiable code using the B method. In Naftalin et al. [302], pages 346–365.

[396] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1991.

[397] B. A. Sufrin. Formal methods and the design of effective user interfaces. In M. D. Harrison and A. F. Monk, editors, *People and Computers: Designing for Usability*. Cambridge University Press, 1986.

[398] B. A. Sufrin. Formal specification of a display-oriented editor. In N. Gehani and A. D. McGettrick, editors, *Software Specification Techniques*, International Computer Science Series, pages 223–267. Addison-Wesley Publishing Company, 1986.

Originally published in Science of Computer Programming, 1:157–202, 1982.

[399] B. A. Sufrin. Effective industrial application of formal methods. In G. X. Ritter, editor, *Information Processing 89, Proc. 11th IFIP Computer Congress*, pages 61–69. Elsevier Science Publishers (North-Holland), 1989.

This paper presents a Z model of the Unix *make* utility.

[400] B. A. Sufrin and He Jifeng. Specification, analysis and refinement of interactive processes. In M. D. Harrison and H. Thimbleby, editors, *Formal Methods in Human-Computer Interaction*, volume 2 of *Cambridge Series on Human-Computer Interaction*, chapter 6, pages 153–200. Cambridge University Press, 1990.

A case study on using Z for process modelling.

[401] P. A. Swatman. Using formal specification in the acquisition of information systems: Educating information systems professionals. In Bowen and Nicholls [70], pages 205–239.

[402] P. A. Swatman, D. Fowler, and C. Y. M. Gan. Extending the useful application domain for formal methods. In Nicholls [317], pages 125–144.

[403] M. C. Thomas. The industrial use of formal methods. *Microprocessors and Microsystems*, 17(1):31–36, January/February 1993.

[404] M. Tierney. The evolution of Def Stan 00-55 and 00-56: an intensification of the "formal methods debate" in the UK. In *Proc. Workshop on Policy Issues in Systems and Software Development*, Brighton, UK, July 1991. Science Policy Research Unit.

[405] D. Till, editor. *6th Refinement Workshop*, Workshop in Computing. Springer-Verlag, 1994.

The workshop was held at City University, London, UK, 5–7 January 1994. See [156, 254].

[406] R. Took. The presenter – a formal design for an autonomous display manager. In I. Sommerville, editor, *Software Engineering Environments*, pages 151–169. Peter Peregrinus, London, 1986.

[407] S. Topp-Jørgensen. Reservation Service: Implementation correctness proofs. Programming Research Group, Oxford University Computing Laboratory, UK, 1987.

[408] I. Toyn and J. A. McDermid. CADiZ: An architecture for Z tools and its implementation. Technical document, Computer Science Department, University of York, York YO1 5DD, UK, November 1993.

[409] S. Valentine. The programming language Z^{--}. *Information and Software Technology*, 37(5-6):293–301, May–June 1995.

[410] S. H. Valentine. Z^{--}, an executable subset of Z. In Nicholls [317], pages 157–187.

[411] S. H. Valentine. Putting numbers into the mathematical toolkit. In Bowen and Nicholls [70], pages 9–36.

[412] M. J. van Diepen and K. M. van Hee. A formal semantics for Z and the link between Z and the relational algebra. In Bjørner et al. [32], pages 526–551.

[413] K. M. van Hee, L. J. Somers, and M. Voorhoeve. Z and high level Petri nets. In Prehn and Toetenel [338], pages 204–219.

[414] D. R. Wallace, D. R. Kuhn, and L. M. Ippolito. An analysis of selected software safety standards. *IEEE AES Magazine*, pages 3–14, August 1992.

[415] M. West, D. Nichols, J. Howard, M. Satyanarayanan, and R. Sidebotham. The ITC distributed file system: Principles and design. CMU Report CMU-ITC-039, Information Technology Center (ITC), Carnegie-Mellon University, USA, March 1985.

[416] M. M. West and B. M. Eaglestone. Software development: Two approaches to animation of Z specifications using Prolog. *IEE/BCS Software Engineering Journal*, 7(4):264–276, July 1992.

[417] C. Wezeman and A. Judge. Z for managed objects. In Bowen and Hall [60], pages 108–119.

[418] R. W. Whitty. Structural metrics for Z specifications. In Nicholls [313], pages 186–191.

[419] P. J. Whysall and J. A. McDermid. An approach to object-oriented specification using Z. In Nicholls [315], pages 193–215.

[420] P. J. Whysall and J. A. McDermid. Object-oriented specification and refinement. In Morris and Shaw [301], pages 151–184.

[421] J. M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, September 1990.

[422] J. M. Wing and A. M. Zaremski. Unintrusive ways to integrate formal specifications in practice. In Prehn and Toetenel [338], pages 545–570.

[423] K. R. Wood. The elusive software refinery: a case study in program development. In Morris and Shaw [301], pages 281–325.

[424] K. R. Wood. A practical approach to software engineering using Z and the refinement calculus. *ACM Software Engineering Notes*, 18(5):79–88, December 1993.

[425] J. C. P. Woodcock. Calculating properties of Z specifications. *ACM SIGSOFT Software Engineering Notes*, 14(4):43–54, 1989.

[426] J. C. P. Woodcock. Mathematics as a management tool: Proof rules for promotion. In *Proc. 6th Annual CSR Conference on Large Software Systems*, Bristol, UK, September 1989.

[427] J. C. P. Woodcock. Parallel refinement in Z. In *Proc. Workshop on Refinement*, January 1989.

[428] J. C. P. Woodcock. Structuring specifications in Z. *IEE/BCS Software Engineering Journal*, 4(1):51–66, January 1989.

[429] J. C. P. Woodcock. Transaction refinement in Z. In *Proc. Workshop on Refinement*, January 1989.

[430] J. C. P. Woodcock. Implementing promoted operations in Z. In Morris and Shaw [301], pages 366–378.

[431] J. C. P. Woodcock. A tutorial on the refinement calculus. In Prehn and Toetenel [339], pages 79–140.

[432] J. C. P. Woodcock. The rudiments of algorithm design. *The Computer Journal*, 35(5):441–450, October 1992.

[433] J. C. P. Woodcock and S. M. Brien. $\mathcal{W}$: A logic for Z. In Nicholls [317], pages 77–96.

[434] J. C. P. Woodcock, P. H. B. Gardiner, and J. R. Hulance. The formal specification in Z of Defence Standard 00-56. In Bowen and Hall [60], pages 9–28.

[435] J. C. P. Woodcock and P. G. Larsen, editors. *FME'93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*. Formal Methods Europe, Springer-Verlag, 1993.

The 1st FME Symposium was held at Odense, Denmark, 19–23 April 1993. Z-related papers include [72, 105, 154, 228, 275, 334].

[436] J. C. P. Woodcock and P. G. Larsen. Guest editorial. *IEEE Transactions on Software Engineering*, 21(2):61–62, 1995.

Best papers of FME'93 [435]. See [35, 31, 108, 229].

[437] J. C. P. Woodcock and M. Loomes. *Software Engineering Mathematics: Formal Methods Demystified*. Pitman, 1988.

Also published as: *Software Engineering Mathematics*, Addison-Wesley, 1989.

[438] J. C. P. Woodcock and C. C. Morgan. Refinement of state-based concurrent systems. In Bjørner et al. [32], pages 340–351.

Work on combining Z and CSP.

[439] R. Worden. Fermenting and distilling. In Bowen and Hall [60], pages 1–6.

[440] J. B. Wordsworth. Refinement tutorial: A storage manager. In *Proc. Workshop on Refinement*, January 1989.

[441] J. B. Wordsworth. A Z development method. In *Proc. Workshop on Refinement*, January 1989.

[442] J. B. Wordsworth. The CICS application programming interface definition. In Nicholls [315], pages 285–294.

[443] J. B. Wordsworth. *Software Development with Z: A A Practical Approach to Formal Methods in Software Engineering*. Addison-Wesley, 1993.

This book provides a guide to developing software from specification to code, and is based in part on work done at IBM's UK Laboratory that won the UK Queen's Award for Technological Achievement in 1992.

Contents: Introduction; A simple Z specification; Sets and predicates; Relations and functions; Schemas and specifications; Data design; Algorithm design; Specification of an oil terminal control system.

[444] Xiaoping Jia. *ZTC: A Type Checker for Z – User's Guide*. Institute for Software Engineering, Department of Computer Science and Information Systems, DePaul University, Chicago, IL 60604, USA, 1994.

ZTC is a type checker for the Z specification language. ZTC accepts two forms of input: LaTeX [251] with the `zed.sty` style option [378] and ZSL, an ASCII version of Z. ZTC can also perform translations between the two input forms. This document is intended to serve as both a user's guide and a reference manual for ZTC.

[445] P. Zave and M. Jackson. Techniques for partial specification and specification of switching systems. In Prehn and Toetenel [338], pages 511–525.

Also published as [446].

[446] P. Zave and M. Jackson. Techniques for partial specification and specification of switching systems. In Nicholls [317], pages 205–219.

[447] Y. Zhang and P. Hitchcock. EMS: Case study in methodology for designing knowledge-based systems and information systems. *Information and Software Technology*, 33(7):518–526, September 1991.

[448] Z archive. Oxford University Computing Laboratory, 1995.

A computer-based archive server at the Programming Research Group in Oxford is available for use by anyone with World-Wide Web (WWW) access, anonymous FTP access or an electronic mail address. This allows people interested in Z (and other things) to access various archived files. In particular, messages from the Z FORUM electronic mailing list [449] and a Z bibliography [46] are available.

The preferred method of access to the on-line Z archive is via the World-Wide Web (WWW) under the following 'URL' (Uniform Resource Locator):

`http://www.zuser.org/z/`

Simply follow the hyperlinks of interest.

Much of the Z archive is also available via anonymous FTP on the Internet. Type the command '`ftp ftp.comlab.ox.ac.uk`' (or alternatively if this does not work, '`ftp 163.1.27.2`') and use '`anonymous`' as the login id and your e-mail address as the password when prompted. The FTP command '`cd pub/Zforum`' will get you into the Z archive directory. The file `README` gives some general information and `00index` gives a list of the available files. The Z bibliography may be retrieved using the FTP command '`get z.bib`', for example. If you wish to access any of the compressed POSTSCRIPT files in the archive, please issue the '`binary`' command first.

For users without on-line access on the Internet, it is possible to access

parts of the Z archive using electronic mail, send a message to `archive-server@comlab.ox.ac.uk` with the '`Subject:`' line and/or the body of the message containing commands such as the following:

`help`	help on using the PRG archive server
`index`	general index of categories (e.g., '`z`')
`index z`	index of Z-related files
`send z z.bib`	send Z bibliography in BIBTEX format
`send z` *file1 file2* ...	send multiple files
`path` *name@site*	optionally specify return e-mail address

If you have serious problems accessing the Z archive using WWW, anonymous FTP access or the electronic mail server and thus need human help, or if you wish to submit an item for the archive, please send electronic mail to `archive-management@comlab.ox.ac.uk`.

[449] Z FORUM. Oxford University Computing Laboratory, 1986 onwards. Electronic mailing list: vol. 1.1–9 (1986), vol. 2.1–4 (1987), vol. 3.1–7 (1988), vol. 4.1–4 (1989), vol. 5.1–3 (1990).

Z FORUM is an electronic mailing list. It was initiated as an edited newsletter by Ruaridh Macdonald of DRA (formerly RSRE), Malvern, Worcestershire, UK, and is now maintained by Jonathan Bowen at the Oxford University Computing Laboratory. Contributions should be sent to `zforum@comlab.ox.ac.uk`. Requests to join or leave the list should be sent to `zforum-request@comlab.ox.ac.uk`. Messages are now forwarded to the list directly to ensure timeliness. The list is also gatewayed to the USENET newsgroup `comp.specification.z` at Oxford and messages posted on either will automatically appear on both. A message answering some frequently asked questions is maintained and sent to the list once a month. A list of back issues of newsletters and other Z-related material is available electronically via anonymous FTP from `ftp.comlab.ox.ac.uk:/pub/Zforum` in the file `00index` or via e-mail from the OUCL archive server [448]. For messages from a particular month (e.g., May 1995), access a file such as `zforum95-05`; for the most recent messages, see the file `zforum`.

Index