

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 likelihood 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 relatively 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 known 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 than 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 use 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 L^AT_EX 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 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 likelihood 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 by 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 mathematics 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 effectively 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 use 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 known 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 years’ 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 hopefully 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 difficulty. 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.