Jonathan P. Bowen, Michael G. Hinchey

High-Integrity System Specification and Design

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.

— C.A.R. Hoare

Preface

Errata, detected in Taylor's Logarithms. *London: 4to, 1792.* [sic]
...
6 Kk Co-sine of 14.18. 3 — 3398 — 3298
... — *Nautical Almanac* (1832)

In the list of ERRATA detected in Taylor's *Logarithms*, for cos. $4^\circ\,18'\,3''$, *read* cos. $14^\circ\,18'\,2''$. — *Nautical Almanac* (1833)

ERRATUM of the ERRATUM of the ERRATA of TAYLOR'S *Logarithms*. For cos. $4^\circ\,18'\,3''$, *read* cos. $14^\circ\,18'\,3''$.
— *Nautical Almanac* (1836)

In the 1820s, an Englishman named Charles Babbage designed and partly built a calculating machine originally intended for use in deriving and printing logarithmic and other tables used in the shipping industry. At that time, such tables were often inaccurate, copied carelessly, and had been instrumental in causing a number of maritime disasters.

Babbage's machine, called a 'Difference Engine' because it performed its calculations using the principle of partial differences, was intended to substantially reduce the number of errors made by humans calculating the tables. Babbage had also designed (but never built) a forerunner of the modern printer, which would also reduce the number of errors admitted during the transcription of the results.

Nowadays, a system implemented to perform the function of Babbage's engine would be classed as *safety-critical*. That is, the failure of the system to produce correct results could result in the loss of human life, mass destruction of property (in the form of ships and cargo) as well as financial losses and loss of competitive advantage for the shipping firm.

Computer systems now influence almost every facet of our lives. They wake us up in the morning, control the cooking of our food, entertain us, help in avoiding traffic congestion and control the vehicles in which we travel, they wash our clothes, and even give us cash from our bank accounts (sometimes!). Increasingly, they are being used in systems where they can have a great influence over our very existence. They control the flow of trains in the subway, signaling on railway lines, even traffic lights on the street. The failure of any of these systems would cause us

great inconvenience, and could conceivably result in accidents in which lives may be lost. As they control the actual flight of an airplane, cooling systems in chemical plants, feedback loops in nuclear power stations, etc., we can see that they allow the possibility of great disasters if they fail to operate as expected.

In recent years, the media, in the form of both television news and the popular science journals, have become very preoccupied with the failures of a number of safety-critical computer systems. A number of systems, or classes of system, seem to have particularly caught their attention; chief amongst these are various nuclear power plants where the cooling systems or shutdown loops have been demonstrated to be inconsistent, and recent air crashes which cannot be convincingly blamed on pilot error.

The introduction of computer systems to replace more traditional mechanical systems (consider, for example, Boeing's fly-by-wire system in the 777 jet, and the disastrous baggage-handling system at Denver airport) has made both the system development community and the general public more aware of the unprecedented opportunities for the introduction of errors that computers admit.

Many journalists have become self-styled authorities on techniques that will give greater confidence in the correctness of complex systems, and reduce the number and frequency of computer errors. *High-Integrity Systems*, or systems whose code is relied upon to be of the highest quality and error-free, are often both security- and safety-critical in that their failure could result in great financial losses for a company, mass destruction of property and the environment, and loss of human life. *Formal Methods* have been widely advocated as one of those techniques which can result in high-integrity systems, and their usage is being suggested in an increasing number of standards in the safety-critical domain. Notwithstanding that, formal methods remain one of the most controversial areas of modern software engineering practice. They are the subject of extreme hyperbole by self-styled 'experts' who fail to understand what formal methods actually are, and of deep criticism by proponents of other techniques who see formal methods as merely an opportunity for academics to exercise their intellects over whichever notation is the current 'flavor-of-the-month'.

This book serves as an introduction to the task of specification and design of high-integrity systems, with particular attention paid to formal methods throughout, as these (in our opinion) represent the most promising development in this direction. We do not claim that this book will tell the reader *everything* he/she needs to know, but we do hope that it will help to clarify the issues, and give a good grounding for further investigation.

Each of the major Parts of the book consists of expository material, couched at levels suitable for use by computer science and software engineering students (both undergraduate and graduate), giving an overview of the area, pointers to additional material, and introductions to the excellent papers which are reprinted in this volume.

For practicing software engineers too, both industrialists and academics, this book should prove to be of interest. It brings together some of the 'classic' works in

the field, making it an interesting book of readings for self-study, and a convenient, comprehensive reference.

Part 1 introduces the many problems associated with the development of large-scale systems, describes the system life-cycle, and suggests potential solutions which have been demonstrated to be particularly promising.

Traditionally, computer systems have been analyzed, specified and designed using a number of diagrammatic techniques. Over the years, different techniques and notations have been combined into various structured methods of development. These are introduced in Part 2, and a number of the more popular methods are described.

Part 3 goes on to describe formal methods, the major focus of this collection. The components of a formal method are identified; overviews of several representative formal methods are given, and major misconceptions regarding formal methods are identified and dispelled.

Object-Orientation has often been cited as a means of aiding in reducing complexity in system development. Part 4 introduces the subject, and discusses issues related to object-oriented development.

With increased performance requirements, and greater dispersal of processing power, concurrent and distributed systems have become very prevalent. Their development, however, can be exponentially more difficult than the development of traditional sequential systems. Part 5 discusses such issues, and describes two diverse approaches to the development of such systems.

Increasingly, complex concurrent and distributed systems are employed in areas where their use can be deemed to be *safety-critical*, and where they are relied upon to perform within strict timing constraints. Part 6 identifies the relationship of formal methods to safety-critical standards and the development of safety-critical systems. The appropriateness of a number of formal methods is discussed, and some interesting case studies are presented.

While formal methods have much to offer, many fail to address the more methodological aspects of system development. In addition, there has been considerable effort invested in the development of appropriate structured methods which it would be foolhardy to ignore. Part 7 presents the motivation for integrating structured and formal methods, as a means of exploiting the advantages of each.

Clearly the aim of system development is to derive a sensible implementation of the system that was specified at the outset. Part 8 introduces refinement of formal specifications, rapid prototyping and simulation, and the relative merits of executable specifications.

Part 9 addresses the mechanization of system development, and tool support via Computer-Aided Software Engineering (CASE). The future of CASE, and the advent of ‘visual formalisms’, exploiting graphical representation with formal underpinnings, is postulated.

Finally, a bibliography with recent references is included for those wishing to follow up on any of the issues raised in more depth. We hope this collection of papers and articles, together with the associated commentary, provide food for thought to

all those actively involved in or contemplating the production of computer-based high integrity systems.

Information associated with this book will be maintained on-line under the following URL (Uniform Resource Locator):

`http://www.fmse.cs.reading.ac.uk/hissd/`

Relevant developments subsequent to publication of this collection will be added to this resource.

J.P.B.
Reading

M.G.H.
Omaha

Table of Contents

Acknowledgements

We are grateful for the permissions to reprint the papers and articles included in this volume provided by authors and publishers. Full details of the original published sources is provided overleaf on page xvii. We are also grateful to all the authors whose work is reproduced in this collection. Robert France, Ian Hayes, Leslie Lamport, Nancy Leveson, Jonathan Ostroff and Jeannette Wing provided original LaTeX format sources for their contributions which helped tremendously in the preparation of the book. Norbert Fuchs, David Harel and Ian Hayes helped with proof reading of their contributions.

Angela Burgess and Michelle Saewert were very helpful in providing sources and scanning some of the IEEE articles in this collection which aided their preparation. The IEEE, through William Hagen, generously allowed all the contributions originally published in IEEE journals to be included in this collection for no payment. Deborah Cotton of the ACM allowed the contributions originally published by the ACM to be included at a very reasonable cost. Jonathan Ostroff contributed to the payment of the reproduction fee for his paper, which otherwise could not have been included.

Some of the material in Part 5 is based on parts of [34] and Part 6 is adapted from parts of [45].

The book has been formatted using the excellent (and free) LaTeX2$_\varepsilon$ Document Preparation System [154]. Many of the diagrams have been prepared using the xfig package. Thank you in particular to Mark Green for preparing many of the Cameron paper diagrams, John Hawkins for transcribing the Orr *et al.* diagrams and some of the Cameron diagrams, and Ken Williams for expertly reconstructing some of the diagrams for the Booch, Chikofsky & Rubenstein and Harel 1992 articles (all at The University of Reading).

Finally thank you to Rosie Kemp together with her assistants, Vicki Swallow and Karen Barker, at Springer-Verlag, London, for bringing this project to fruition in print. Rosie generously organized initial typing of many of the articles and was very helpful in organizing copyright permissions. Without her friendly reminders this book would probably never have reached completion.

List of Reprints

Norbert E. Fuchs. 583
Specifications are (Preferably) Executable.
Software Engineering Journal, 7(5):323–334, September 1992.

Susan Gerhart, Dan Craigen and Ted Ralston. 413
Experience with Formal Methods in Critical Systems.
IEEE Software, 11(1):21–28, January 1994.

Susan Gerhart, Dan Craigen and Ted Ralston. 429
Regulatory Case Studies.
IEEE Software, 11(1):30–39, January 1994.

J. Anthony Hall. 135
Seven Myths of Formal Methods.
IEEE Software, 7(5):11–19, September 1990.

David Harel. 623
On Visual Formalisms.
Communications of the ACM, 31(5):514–530, May 1988.

David Harel. 29
Biting the Silver Bullet: Toward a Brighter Future for System Development.
IEEE Computer, 25(1):8–20, January 1992.

Ian J. Hayes and Cliff B. Jones. 563
Specifications are not (Necessarily) Executable.
Software Engineering Journal, 4(6):330–338, November 1989.

C.A.R. Hoare. 303
Communicating Sequential Processes.
Communications of the ACM, 21(8):666–677, August 1978.

C.A.R. Hoare. 201
An Overview of Some Formal Methods for Program Design.
IEEE Computer, 20(9):85–91, September 1987.

Leslie Lamport. 331
A Simple Approach to Specifying Concurrent Systems.
Communications of the ACM, 32(1):32–45, January 1989.

Nancy G. Leveson. 447
Medical Devices: The Therac-25 Story.
Safeware: System Safety and Computers, Addison-Wesley Publishing Company, Inc., 1995, Appendix A, pages 515–553.

1. Specification and Design

Computers do not make mistakes, or so we are told; but computer software is written by human beings, who certainly *do* make mistakes. Errors may occur as a result of misunderstood or contradictory requirements [77, 234, 265], unfamiliarity with the problem, or simply due to human error during coding. Whatever the cause of the error, the costs of software maintenance (rectifying errors and adapting the system to meet changing requirements or changes in the environment) have risen dramatically over recent years. Alarmingly, these costs now greatly exceed the original programming costs.

The media have recently shown a great interest in computer error, in particular where *safety-critical systems* are involved. These are systems where a failure could result in the loss of human life, or the catastrophic destruction of property (e.g., flight-controllers, protection systems of nuclear reactors). Lately, however, many financial systems are being classed as 'safety-critical' since a failure, or poor security arrangements, could result in great financial losses or a breach of privacy, possibly resulting in the financial ruin of the organization.

Most major newspapers have at some time or other carried articles on Airbus disasters, or discussing fears regarding the correctness of the software running nuclear reactor control systems. The problem with the latter is that since the software is so complex, consisting of hundreds of thousands, or even millions, of lines of code, it can never be fully tested. Reading these articles, it appears that a number of journalists have set themselves up as self-appointed experts on the subject. The most common claim that they make is that had particular techniques been used at the outset, these problems could have been avoided completely. These claims are often completely without justification.

But how then are we to develop computer systems that will operate as expected, i.e., predictably or 'correctly'?

1.1 An Analogy

As complex computer systems influence every facet of our lives, controlling the cars we drive, the airplanes and trains we rely on others to drive for us, and even in

everyday machinery such as domestic washing machines, the need for *reliable* and *dependable* systems has become apparent.

With systems increasing rapidly both in size and complexity, it is both naïve and ludicrous to expect a programmer, or a development team, to write a program or system without stating clearly and unambiguously what is required of the program or suite of programs.

Surely nobody would hire a builder and just ask him to build a house. On the contrary, they would first hire an architect, state the features of the house that are *required* and those that are *desired* but not essential. They are likely to have many conflicting goals in terms of the features that are required and what is actually possible. There may be environmental constraints (e.g., the ground is too water-logged for a basement), financial constraints, governmental regulations, etc., all of which will influence what can actually be built.

Different members of the family are likely to have different requirements, some of which will be compatible, others which will not be. Before the house can be built, the family must come to agreement on which features the finished house will have. The architect will formulate these in the form of a set of blueprints, and construction can begin.

Often the construction team will discover errors, omissions and anomalies in the architect's plans, which have to be resolved in consultation with the architect and the family. These may be as a result of carelessness, or sometimes due to unexpected conditions in the environment (e.g., finding solid rock where the foundations should be laid). The problems will be resolved by changing the plans, which sometimes will require modifying other requirements and other aspects of the building.

Finally, when the house has been built, the architect will inspect the work, ensuring that all the relevant building quality standards have been adhered to and that the finished building corresponds to the plans that were drawn up at the outset. Assuming that everything is satisfactory, final payments will be made, and the family can move into their new home.

However, they will often uncover deficiencies in the work of the construction team: faulty wiring, piping, etc. that needs to be replaced. They may also find that the house does not actually meet their requirements; for example, it may be too small for their family meaning that eventually they need to add an extra room.

Even if they find that the house is ideal for their requirements at the outset, over time they will decide that they want changes made. This may mean new fixtures and fittings, new furniture, re-decorating, etc., all of which is a natural part of the existence of a house.

Developing a complex computer system follows a similar development process, or *life-cycle*, except that the development is likely to be less well understood, far more complex, and considerably more costly.

1.2 The Development Life-Cycle

Just as there is a set ordering of events in the construction of a house, similarly there is a set ordering of stages in the 'ideal' system development. We use the word 'ideal' advisedly here, as we will see shortly.

The software development life-cycle is usually structured as a sequence of phases or stages, each producing more and more detailed tangible descriptions of the system under consideration. These are often ordered in a 'waterfall' style as identified by Royce [223], and as illustrated in Figure 1.1, with each phase commencing on the completion of the previous phase.

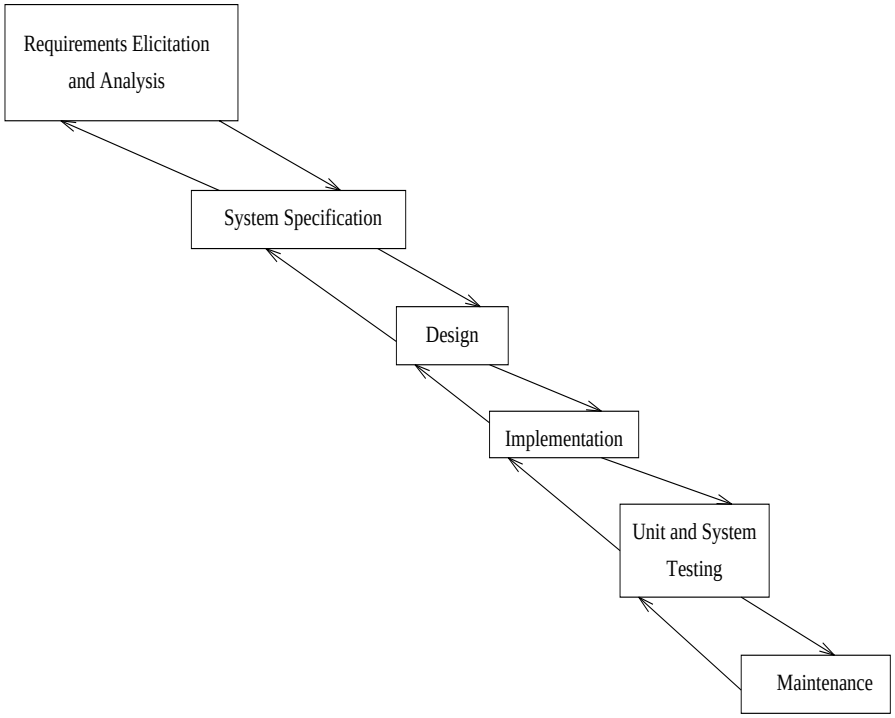

Figure 1.1. Waterfall model of system development (modified)

The first phase, *Requirements Elicitation and Analysis* involves the determination of the exact requirements of the system. It involves talking to end-users and system procurers (those actually contracting the development, and usually incurring the cost), both informally and in arranged interviews. It is likely than many inconsistencies and contradictions will be identified at this point, and these must be resolved. Some of these problems will be very obvious, others not so.

The deliverable at the end of this phase is the requirements specification, a document detailing the system requirements, be they functional (i.e., services which the system is expected to perform) or non-functional (i.e., restrictions or constraints placed on the system's provision of services). The specification is usually written in natural language, augmented as necessary with tables, diagrams, etc., and sometimes with the more precise parts of the requirements expressed in the notation of a structured development method (see Part 2) or a more rigorous formal specification language (see Part 3) [210].

The deliverable is the input to the next stage of the development, *System Specification*. At this stage, it is used in deriving an unambiguous specification of *what* the system should do, without saying *how* this is to be achieved [232]. This will almost certainly be written in the notation of a structured method, such as SA/SD [67, 261], Yourdon [262], SSADM [56], Mascot [222], or JSD [143], etc., or more commonly using the specification language of one of the more popular formal methods, such as VDM [141] or Z [268].

The System Specification, or *functional specification* is generally written in a highly abstract manner, constraining the implementation as little as possible. That is to say, any implementation that satisfies the specification should be acceptable, with no obligation on the way any particular constructs should be implemented. The object at this point is rather to develop an explicit model of the system that is clear, precise and as unambiguous as possible; the final implementation may bear no obvious resemblance to the model, with the *proviso* that it satisfies all of the constraints and all of the functionality of the model.

The intention is that specification and implementation are separated, and implementation issues are only considered at the appropriate juncture. Unfortunately, such a separation, although logical, is unrealistic. Phases of the life-cycle inevitably overlap; specification and implementation are the *already-fixed* and *yet-to-be-done* portions of the life-cycle [241], in that every specification, no matter how abstract, is essentially the implementation of some higher level specification. As such, the functional specification is an implementation of the requirements specification, which is itself implemented by the design specification.

The reader should understand, as a consequence, the difficulty of producing a good functional specification – that is, one that is both high-level enough to be readable and to avoid excluding reasonable implementations, and yet low-level enough to *completely* and *precisely* define the behavior of the implementation.

At the System Specification phase, the intention is to state *what* is required. This generally makes extensive use of implicit (functional) definition, where precise orderings are not specified. At the *Design* phase, however, the aim is to reduce the level of abstraction and to begin addressing *how* the system is to be implemented. This involves considering how various data are to be represented (e.g., considering the efficiency of various data structures), more explicit definition, and how various constructs may be decomposed and structured.

Again, the notation of a structured method or a formal specification language may be used at this point, but using a more operational style, using more realistic data representations, such as files, arrays and linked-lists, and lower level constructs.

The Design Specification (or simply 'Design') is used in deriving the actual implemented program. A program is itself just a specification, albeit an executable one. It is also the most accurate description of the system, as the execution of the system is based on the microcode that corresponds *directly* to the program. The program must, however, address many more issues than those dealt with in the more abstract specifications. It must consider interaction with the underlying hardware and operating system, as well as making efficient use of resources. In fact, the major distinction between an executable specification and an actual program is resource management [266].

At the design phase, the level of abstraction is reduced gradually in a process known as *stepwise refinement*, with more and more detail introduced at each step, the description at each step becoming a more low-level specification of the system. This process continues until the design is in a format where it can be almost transliterated into a programming language by a competent programmer in the *Implementation* phase.

The Implementation Phase, or what is traditionally described as 'programming', is no longer the major contributor to development costs. While programmers still make errors, and the program 'bug' is something that will always be familiar to us, the major cost of software development comes *after* the system has been implemented. It is then that the system is subjected to *Unit* and *System Testing* which aims to trap 'bugs'. However this increasingly tends to highlight inconsistencies and errors in the requirements specification, or in mapping these to the functional specification.

As much as 50% of the costs of system development may be due to the costs of system maintenance. Of this, only 17% is likely to be *corrective* (i.e., removing 'bugs'), just 18% is *adaptive* (i.e., modifying the software to add extra functionality, or to deal with changes in the environment), with a phenomenal 65% being due to *perfective* maintenance [171], much of which is due to errors at the earlier stages of development, such as incomplete and contradictory requirements, imprecise functional specification and errors in the design.

As anyone who has had experience of software development will quickly realize, such a view of the development cycle is very simplistic. As one can even see from the house-building analogy given in the previous section, the distinction between the various phases of development is not clear. System development is not a straightforward process, progressing from one stage to another in a linear fashion. Rather, it is an iterative process, whereby various stages may be repeated a number of times as problems and inconsistencies are uncovered, and as requirements are necessarily modified.

Royce's model [223] holds that system requirements and the system specification are frozen before implementation. However, at implementation, or during post-implementation testing, or in an extreme case, at some point during post-implement-

ation execution, errors in the system specification are often uncovered. Such errors require corrections to be made to the specification, or sometimes a reappraisal of the system requirements. One would hope that using (relatively) more recent developments such as formal specification techniques, such errors would be detected during system specification; unfortunately, although such techniques are often augmented with theorem provers and proof-checkers, errors may still be made in proofs, and system development remains a human activity which is prone to error.

It is not surprising then that Royce's 'waterfall' model has been criticized [93, 175] as being unrepresentative of actual system development. Nowadays, the more accepted model of the system life-cycle is one akin to Boehm's 'spiral' model [19] (as illustrated in Figure 1.2), which takes more account of iteration and the non-linear nature of software development, and allows for the re-evaluation of requirements and for alterations to the system specification even after the implementation phase.

1.3 The Transformational Approach

It should be pointed out that there is no definitive model of the system life-cycle, and the development process employed is likely to vary for different organizations and even for different projects within a given organization. An alternative approach to the life-cycle model is what has come to be known as the *transformational* or *evolutionary* approach to system development. This begins with a very simple, and inefficient, implementation of the system requirements, which is then transformed into an efficient program by the application of a sequence of simple transformations. The intention is that libraries of such transformations, which are known to preserve semantics and correctness, should be built up.

There are however, a number of flaws in the approach [95]:

- there are tasks for which even the simplest and most inefficient program is complex, whereas a specification of the task could easily be realized;
- to demonstrate the correctness of the transformation, it is necessary to verify that certain properties hold before the transformation may be applied, and after its completion; a specification of how the program is to function is required in order to verify these conditions;
- transformations are applied to parts of programs and not to entire systems; to ensure that transformations do not have side-effects, specifications of program parts are required;
- the idea of building up libraries of transformations is attractive, but has not worked in practice; transformations tend to be too specialized to be applicable to other systems, the resulting libraries becoming too large to be manageable.

The approach is not incongruous with the more traditional development life-cycle, but rather specifications are required if transformations are to be used to derive correct programs. In more recent years, the proponents of the transformational

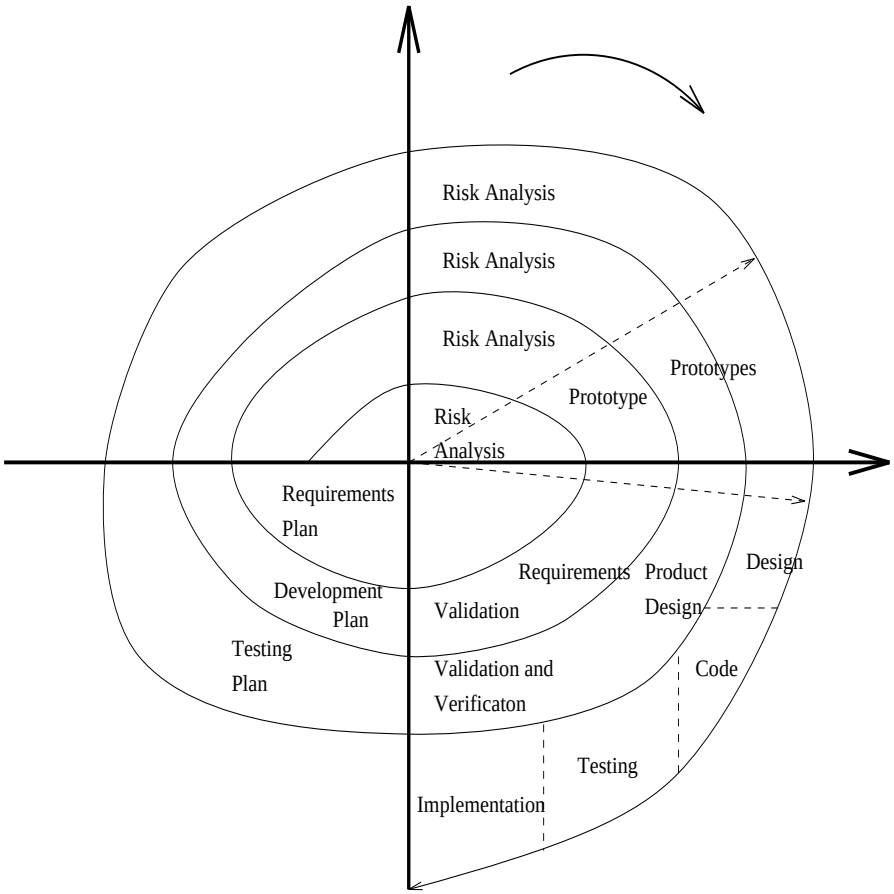

Figure 1.2. Spiral model of system development (simplified)

approach have come to realize the dependency, and *transformational programming* has come to refer to the application of correctness-preserving transformations to formal specifications, in a constructive approach to system development.

1.4 Silver Bullets

With computer systems being applied in more and more 'safety-critical' domains, the need to be assured of the 'correctness' of the system has become increasingly vital. When system failures can result in large-scale destruction of property, great financial loss, or the loss of human life, nothing can be left to chance. In the development of such complex systems, informal inferences are not satisfactory. Firstly, we require a definite means of *proving* that the system we are building adequately reflects all of the requirements specified at the outset. We must validate these requirements and determine that they do not conflict, and ensure that realizing those requirements would result in a satisfactory system. Secondly we must determine that these requirements are complete, and be able to demonstrate that all potential eventualities are covered. Finally, we must be able to *prove* that a particular implementation satisfies each of the requirements that we specified.

We have seen that system development is not a one-pass process, but rather involves multiple iterations, subject as it is to the imprecision of natural language and the indecision and whim of procurers and end-users. Even with increased levels of automation, Computer-Aided Software Engineering (CASE) workbenches (see Part 9), more precise specifications (see Part 2) and more appropriate design methods (see Parts 2, 4, 5 and 6), system development will remain an imprecise process, subject to human input, and human error. As such, the system development process will always be the subject of further research, and a source of possible improvements.

In his widely-quoted and much-referenced article, *No Silver Bullet* (reprinted in this volume), Fred Brooks, also of *Mythical Man-Month* fame [51], warns of the dangers of complacency in system development [50]. He stresses that unlike hardware development, we cannot expect to achieve great advances in productivity in software development unless we concentrate on more appropriate development methods. He highlights how software systems can suddenly turn from being well-behaved to behaving erratically and uncontrollably, with unanticipated delays and increased costs (e.g., for a spectacular and expensive example, see the ARIANE 5 failure in 1996 [174]). Brooks sees software systems as 'werewolves', and rightly points out that there is no single technique, no 'Silver Bullet', capable of slaying such monsters.

On the contrary, more and more complex systems are run on distributed, heterogeneous networks, subject to strict performance, fault tolerance and security constraints, all of which may conflict. Many engineering disciplines must contribute to the development of complex systems in an attempt to satisfy all of these requirements. No single technique is adequate to address all issues of complex system development; rather, different techniques must be applied at different stages of development to ensure unambiguous requirements statements, precise specifications that

are amenable to analysis and evaluation, implementations that satisfy the requirements and various goals such as re-use, re-engineering and reverse engineering of legacy code, appropriate integration with existing systems, ease of use, predictability, dependability, maintainability, fault-tolerance, etc.

Brooks differentiates between the *essence* (i.e., problems that are necessarily inherent in the nature of software) and *accidents* (i.e., problems that are secondary and caused by current development environments and techniques). He points out the great need for appropriate means of coming to grips with the conceptual difficulties of software development – that is, for appropriate emphasis on specification and design; he writes:

> *I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation.*

In his article he highlights some successes that have been achieved in gaining improvements in productivity, but points out that these address problems in the current development process, rather than those problems inherent in software itself. In this category, he includes: the advent of high-level languages (such as Ada [8]), time-sharing, unified programming environments, object-oriented programming, techniques from artificial intelligence, expert systems, automatic programming, , program verification, and the advent of workstations. These he sees as 'non-bullets' as they will not help in slaying the werewolf.

He sees software reuse [88], rapid prototyping (discussed in Part 8), incremental development (akin to the transformational approach described earlier) and the employment of top-class designers as potential starting points for the 'Silver Bullet', but warns that none in itself is sufficient.

Brooks' article has been very influential, and remains one of the 'classics' of software engineering. His viewpoint has been criticized, however, as being overly pessimistic and for failing to acknowledge some promising developments.

Harel, in his article *Biting the Silver Bullet* (also reprinted in this Part), points to developments in CASE and Visual Formalisms (see Part 9) as potential 'bullets' [109]. Harel's view is far more optimistic. He writes five years after Brooks, and has seen the developments in that period. The last forty years of system development have been equally difficult, according to Harel, and using a conceptual 'vanilla' framework, we devised means of overcoming many difficulties. Now, as we address more complex systems, we must devise similar frameworks that are applicable to the classes of systems we are developing.

Concentrating on reactive systems (see Part 6), he describes one such 'vanilla' framework, with appropriate techniques for modeling system behavior and analyzing that model. Harel, as many others, believes that appropriate techniques for modeling must have a rigorous mathematical semantics, and appropriate means for representing constructs (disagreeing with Brooks, who sees representational issues as primarily *accidental*), using visual representations that can be meaningful to engineers and programmers; he says:

> *It is our duty to forge ahead to turn system modeling into a predominantly visual and graphical process.*

He goes on to describe his concepts in more detail in his paper *On Visual Formalisms* [108], reprinted in Part 9.

Software engineering is a wide-ranging discipline in general requiring expertise in a number of related areas to ensure success. Software quality is of increasing importance as the use of software becomes more pervasive. Formal example, the Software Engineering Institute (SEI, based at Carnegie-Mellon University, Pittsburg) and Mitre Corporation have proposed a Capability Maturity Model (CMM) for assessing an organization's software process capability [82].

Those interested in exploring the topic of software engineering further are recommended to read one of the comprehensive reference sources on this subject (e.g., see [176, 181, 217]). For the future, software architecture [231] is emerging as an approach in which typically software components are designed to interface with each other in a similar way that hardware components are designed to fit together in other engineering disciplines. This has proved to be a difficult problem, but may improve the hope for more software reuse in future products [88]. However software reuse should be undertaken with caution, since when misapplied, disastrous consequences can result (e.g., see [174]). In the industrial application of any technique to aid software development, including for high-integrity systems, adequate and dependable tool support is vital for success [245].

No Silver Bullet: Essence and Accidents of Software Engineering (Brooks)

TO BE ADDED: Brooks: No Silver Bullet

TO BE ADDED: 18 pages

Biting the Silver Bullet: Toward a Brighter Future for System Development (Harel)

TO BE ADDED: Harel: Biting the Silver Bullet

TO BE ADDED: 24 pages

2. Structured Methods

In the early 1960s, as it became obvious that computer systems would be developed by teams of professionals rather than by individual programmers, and that as the functionality required would push available resources to the limit, more appropriate specification and design methods (as discussed in the previous Part) were needed.

Various notations and techniques that had evolved for use by programmers, such as structured text, pseudo-code and flow-charts were useful for individual programs, but insufficient for use in large scale *system* development. For this reason, a number of new notations and techniques evolved during the 1960s and '70s to aid in the description, specification and design of computer systems.

These techniques were generally referred to as *structured techniques* or (when applied at the analysis phase of system development) *structured analysis techniques* because they based the design of the system on structural decomposition and emphasized structured programming [142]. As more and more notations evolved, proposals were made to combine them in various ways to provide complete descriptions of a system, and a *structured method* of development.

2.1 Structured Notations

Development methods such as Yourdon [262], Structured Design [261] and the Structured Systems Analysis and Design Methodology (SSADM) [56] are in fact unifying approaches, drawing together various structured notations, together with Hierarchy Charts, Decision Tables, Data Dictionaries, Structured English, etc., in an attempt to devise a coherent methodology.

Such methodologies vary in the notations that they see as essential, and in the emphasis they place on different techniques; there are also minor syntactic differences in their representation of various constructs.

Methodology: the Experts Speak (reprinted here) is a unique paper in that it provides an overview of various methodologies and notations, with the descriptions being written by the actual originators of the methodologies [199].

Ken Orr differentiates between a *method* and a *methodology*, which is strictly speaking the study of methods. He gives a detailed account of the history of the

development of what has become known as the Warner-Orr method (more correctly named *Data-Structured Systems Development*, or DSSD), and how it relates to various other techniques and approaches. The name itself emphasizes the application of the approach to information systems and the reliance on the description of a system according to the structure of its data. As such, *Entity diagrams* describing relationships between entities (objects of concern in the system) play a major rôle, with *Assembly-line diagrams* describing logical flows through the system.

Chris Gane describes the approach co-developed with Trish Sarson, which emphasizes the use of Entity-Relationship Diagrams (ERDs) and Data-Flow Diagrams (DFDs). The ERD (described in more detail by Peter Chen) describes the logical relationships between entities and their attributes (data) in a representation-independent manner, but focusing on the structure of the data to be stored. The DFD, as the name suggests, focuses on the flow of data through a system, delimiting the system by considering only the relevant data flows. DFDs are written at various levels of abstraction, with the top level, or context diagram, defining the scope of the system. More and more detailed levels are derived, with a 'bubble' representing a process, and arrows representing the flow of data. Each process can be broken down into a more detailed DFD showing its structure and the flow of data within it. This process continues, in a *top-down* fashion, until the process is at a level at which it can be described in pseudo-code or structured text.

Edward Yourdon describes partitioning a system in terms of DFDs, and the rôle of DFDs in the Yourdon method, while Larry Constantine describes the Structured-Design Approach, which emphasizes the use of models (design methods), measures of system cohesion and complexity, and methods of development. He describes the need for tool support and automation in the application of structured methods, and overviews the rôle of CASE (Computer-Aided Software Engineering) technology, which is described in more detail in Part 9.

Most of the structured methods in popular usage have evolved to meet the needs of information systems (one of the primary applications of computers in the 1960s), although most have been extended to meet the needs of concurrent and real-time systems, with the addition of State-Transition Diagrams (STDs) and other notations to address changes in state, causality, etc. One method however, Jackson System Development (JSD), was developed with concurrency and the requirements of control systems specifically in mind.

2.2 The Jackson Approach

With the introduction of JSP (Jackson Structured Programming) [142], Michael Jackson greatly simplified the task of program design. The success of his approach lies in its intuitiveness and the simple graphical notation that facilitates the design of programs which obey the laws of structured programming – that use only the simple constructs of sequence, selection and iteration, and avoid the 'harmful' goto statement [69] or its equivalent.

Eight years later, with the evident need to address *system design* rather than merely program design, Jackson introduced JSD, or Jackson System Development (often erroneously, but still appropriately, referred to as Jackson Structured Design), which has had a major influence on many subsequent development methods, in particular on both structured and formal methods intended for use in the development of concurrent systems.

In his paper *An Overview of JSD* (reprinted here), John Cameron describes the method, with many examples illustrating its application from analysis through to implementation [55].

The method incorporates JSP itself, in that JSP descriptions of each program or *process* in the system are combined by placing such processes running in parallel and in communication over various types of buffered communication channels. As such, the description of a system is at two levels – each process is described in terms of JSP, while the entire system is described as a network of such processes, executing independently. A process becomes suspended when it requires a particular input, and resumes again when that input is available to it.

As the method does not address process scheduling, clearly a certain degree of non-determinism is inherent [137]. Although it is considered inappropriate in conventional programming languages, non-determinism can be advantageous in system design. JSD also supports high levels of abstraction in that there are generally more processes in the network than available processors in the implementation environment. In the implementation phase of JSD, the number of processes in the network is reduced, often just to a single process (for execution in a single processor environment).

Methodology: The Experts Speak (Orr, Gane, Yourdon, Chen & Constantine)

TO BE ADDED: Orr et al: Methodology: The Experts Speak

TO BE ADDED: 20 pages

An Overview of JSD (Cameron)

TO BE ADDED: Cameron: An overview of JSD

TO BE ADDED: 50 pages

3. Formal Methods

Although they are widely cited as one of those techniques that can result in high-integrity systems [31], and are being mandated more and more in certain applications (see Part 6), formal methods remain one of the most controversial areas of current software engineering practice [119].

They are unfortunately the subject of extreme hyperbole by self-styled 'experts' who fail to understand exactly what formal methods are; and, of deep criticism and subjective evaluation by proponents of other techniques who see them as merely an opportunity for academics to exercise their intellects using mathematical hieroglyphics. Notwithstanding, the level of application of formal techniques in the specification and design of complex systems has grown phenomenally, and there have been a significant number of industrial success stories [125, 106, 160].

Whether one accepts the need for formal methods or not, one must acknowledge that a certain degree of formality is required as a basis for all system development. Conventional programming languages are themselves, after all, formal languages. They have a well-defined formal semantics, but unfortunately as we have already seen, deal with particular implementations rather than a range of possible implementations.

The formal nature of programming languages enables analysis of programs, and offers a means of determining definitively the expected behavior of a program (in the case of closed systems; with open systems the situation is complicated by environmental factors). In a similar fashion, formality at earlier stages enables us to rigorously examine and manipulate requirements specifications and system designs, to check for errors and miscomprehensions. A formal notation makes omissions obvious, removes ambiguities, facilitates tool support and automation, and makes reasoning about specifications and designs an exact procedure (e.g., using a formal verification environment [13]), rather than 'hand-waving'. Systems can be formally verified at various levels of abstraction, and these can be formally linked [36, 155], but normally the higher levels (e.g., requirements [64] and specification) are the most cost effective.

3.1 What are Formal Methods?

The term ‘formal methods’ is rather a misleading one; it originates in formal logic, but nowadays is used in computing to cover a much broader spectrum of activities based upon mathematical ideas [133].

So-called formal methods are not so much ‘methods’ as formal systems. While they also support design principles such as decomposition and stepwise refinement [187], which are found in the more traditional structured design methods, the two primary components of formal methods are a notation and some form of deductive apparatus (or proof system).

3.2 Formal Specification Languages

The notation used in a formal method is called a formal specification language or ‘notation’ to emphasize its potential non-executability. The language is ‘formal’ in that it has a formal semantics and consequently can be used to express specifications in a clear and unambiguous manner.

Programming languages are formal languages, but are not considered appropriate for use in formal specifications for a number of reasons:

– Firstly, very few programming languages have been given a complete formal semantics (Ada and Modula-2 are exceptions), which makes it difficult to prove programs correct and to reason about them.
– Secondly, when programming languages (particularly imperative languages) are used for specifications, there is a tendency to over-specify the ordering of operations. Too much detail at an early stage in the development can lead to a *bias* towards a particular implementation, and can result in a system that does not meet the original requirements.
– Thirdly, programming languages are inherently executable, even if they are declarative in nature. This forces executability issues to the fore, which may be inappropriate at the early stages of development, where the ‘what’ rather than the ‘how’ should be considered.

The key to the success of formal specification is that we *abstract* away from details and consider only the essential relationships of the data. We need to move away from the concrete, which has an indeterminate semantics, and use a formal language so that we can specify the task at hand in a manner that is clear and concise. In this way, abstraction both shortens and clarifies the specification.

Mathematics are used as the basis for specification languages, and formal methods in general. This is because mathematics offer an unchanging notation with which computer professionals should be familiar; in addition the use of mathematics allows us to be very precise and to provide convincing arguments to justify our solutions. This allows us to *prove* that an implementation satisfies the mathematical specification. More importantly, however, mathematics allows us to generalize a

problem so that it can apply to an unlimited number of different cases (in this way, there is no *bias* towards a particular implementation), and it is possible to model even the most complex systems using relatively simple mathematical objects, such as sets, relations and functions [41, 206].

3.3 Deductive Apparatus

The deductive apparatus is an equally important component of a formal method. This enables us to propose and verify properties of the specified system, sometimes leading people to believe erroneously that formal methods will eliminate the need for testing.

Using the deductive apparatus (proof system) of a formal method, it is possible to prove the correctness of an implemented system *with respect to its specification*. Unfortunately, the media, and many computer science authors too, tend to forget to mention the specification when writing about *proof of correctness*; this is a serious oversight, and is what has led some people to believe that formal methods are something almost 'magical'. A proof of correctness demonstrates that a mathematical model of an implementation 'refines', with respect to some improvement ordering, a mathematical specification, not that the actual real-world implementation meets the specification.

We cannot speak of absolute correctness when verifying a system, and to suggest that the proof system enables us to definitively prove the correctness of an implementation is absurd, but the production of 'correct' programs is still a subject of debate [253]. Mathematical proof has essentially been a social process historically and accelerating this process using tool support in a software engineering context is difficult to achieve [178]. What a proof system does do, however, is to let us prove rigorously that the system we have implemented satisfies the requirements determined at the outset. If these requirements were not what we really intended, then the implementation will not function as we intended, but may still be correct with respect to those particular requirements.

The deductive apparatus does however let us *validate* these original requirements; we may propose properties and using a rigorous mathematical argument demonstrate that they hold. While natural language is notorious for introducing contradictory requirements which are often only discovered during implementation, using formal methods we may demonstrate that requirements are contradictory (or otherwise) *before* implementation.

Similarly, natural language requirements tend to result in ambiguity and incomplete specifications. Often when reading systems requirements we have to ask ourselves questions such as 'But what happens if . . . ?', and have no way of determining the answer. With formal methods, however, we can *infer* the consequences based on the requirements that have been specified.

Validation of requirements and *verification* of the matching implementation against those requirements are both useful complementary techniques in aiding the reduction of errors and formal methods can help in both this areas [74].

3.4 Myths of Formal Methods

The computer industry is slow to adopt formal methods in system development. There is a belief that formal methods are difficult to use and require a great deal of mathematical ability. Certainly some of the symbols look daunting to the uninitiated, but it's really just a matter of learning the notation [47]. For complete formal development including proofs and refinement (a process whereby a formal specification is translated systematically into a lower-level implementation, often in a conventional programming language), a strong mathematical background is required, but to write and understand specifications requires only a relatively basic knowledge of mathematics.

There is also a misconception that formal methods are expensive. Experience has now shown that they do not necessarily increase development costs; while costs are increased in the initial stages of development, coding and maintenance costs are reduced significantly, and overall development costs can be lower [123].

It has been suggested that formal methods could result in error-free software and the end of testing. But will they? In a word, no; but they are certainly a step in the right direction to reduce the amount of testing necessary. Indeed, the test phase may become a certification phase, as in the Cleanroom approach [73, 183, 184, 215], if the number of errors are reduced sufficiently. Formal methods and testing are both complementary and worthwhile techniques that are useful in attempting to construct 'correct' software [62]. Formal methods aim to avoid the inclusion of errors in the first place whereas testing aims to detect and remove errors if they have been introduced.

Formal methods enable us to rigorously check for contradictory requirements and to reason about the effects of those requirements. That unfortunately does not mean that we will eliminate requirements errors completely. Proofs are still performed by humans, and are thus still prone to error. Many automated proof assistants are now available which check proof justifications and some can even generate proof obligations as a guide to the construction of a proof. But as we all well know, computer-based tools may themselves contain errors. As research in this area progresses, we can anticipate simplified proofs in the future.

Unfortunately refinement and proof techniques are not exploited as much as they might be [132]. Most developers using formal methods tend to use them at the requirements specification and system specification stages of development [81]. This is still worthwhile, and is indeed where the greatest pay-offs from the use of formal methods have been highlighted, since most errors are actually introduced at the requirements stage rather than during coding. But, as specifications are not refined to executable code, coding is still open to human error. Refinement *is* difficult, and certainly does not guarantee error-free code, as mistakes can still be made during the refinement process. Forthcoming refinement tools should simplify the refinement process and help to eliminate errors. There is disagreement as to how much refinement can be automated, but in any case these tools should help us to eliminate the scourge of computer programming – the ubiquitous 'bug'.

After over a quarter of a century of use, one would have hoped that misconceptions and 'myths' regarding the nature of formal methods, and the benefits that they can bring, would have been eliminated, or at least diminished. Unfortunately, this is not the case; many developers still believe that formal methods are very much a 'toy' for those with strong mathematical backgrounds, and that formal methods are expensive and just deal with proving programs correct. We must attempt to learn lessions from the experience of applying formal methods in practice [68].

In a seminal article *Seven Myths of Formal Methods* (reprinted in this Part), Anthony Hall attempts to dispel many 'myths' held by developers and the public at large [105]. By reference to a single case study, he cites seven major 'myths' and provides an argument against these viewpoints.

While Hall deals with myths held by non-specialists, the authors, writing five years later, identify yet more myths of formal methods. What is disconcerting is that these myths, described in *Seven More Myths of Formal Methods* (reprinted here) are myths accepted to be valid by specialist developers [39]. By reference to several highly successful applications of formal methods, all of which are described in greater detail in [125], we aim to dispel many of these misconceptions and to highlight the fact that formal methods projects can indeed come in on-time, within budget, produce correct software (and hardware), that is well-structured, maintainable, and which has involved system procurers and satisfied their requirements. There is still debate on how formal specification can actually be in practice [163].

3.5 Which Formal Method?

Formal methods are not as new as the media would have us believe. One of the more commonly-used formal methods, the Vienna Development Method (VDM) [17, 146, 141], first appeared in 1971, Hoare first proposed his language of Communicating Sequential Processes (CSP) [128, 129, 127] in 1978, and the Z notation [28, 145, 236, 268] has been under development since the late 1970s. In fact the first specification language, Backus-Naur Form (BNF), appeared as long ago as 1959. It has been widely accepted that *syntax* can be formally specified for quite some time, but there has been more resistance to the formal specification of *semantics*.

Over the last twenty years, these formal methods and formal languages have all changed quite considerably, and various extensions have been developed to deal with, for example, object-orientation and temporal (timing) aspects in real-time systems. But which is the best one to use?

This is very subjective; in fact, it is not really a question of 'which is best' but 'which is most appropriate'. Each of the commonly used formal methods have their own advantages, and for particular classes of system some are more appropriate than others. Another consideration is the audience for which the specification is intended. Some people argue that VDM is easier to understand than Z because it is more like a programming language; others argue that this causes novices to introduce too much detail. Really it is a matter of taste, and often it depends on the 'variety' of Z or

VDM you are using. Both of these methods have been undergoing standardization by the International Standards Organization (ISO) [141, 268], and VDM is now an accepted ISO standard. Newer formal methods such as the B-Method (pehaps one of the more successful tool-supported formal developments methods, with a good range of supporting books [2, 15, 115, 156, 230, 260]) and RAISE [218] tend to have increasing tool support, but depend on a critical mass of users to ensure successful transfer into genuine industrial application [100].

Another consideration is the extent to which the formal specification language is executable (see Part 8 for a discussion of the relative merits and demerits of executable specification languages). OBJ [94], for example, has an executable functional programming subset, and CSP is almost executable in the parallel programming language Occam [139].

We can divide specification languages into essentially three classes:

- Model-oriented
- Property-oriented
- Process algebras

Model-oriented approaches, as exemplified by Z and VDM, involve the explicit specification of a state model of the system's desired behavior in terms of abstract mathematical objects such as sets, relations, functions, and sequences (lists).

Property-oriented approaches can be further subdivided into *axiomatic methods* and *algebraic methods*. Axiomatic methods (e.g., Larch [103]) use first-order predicate logic to express preconditions and postconditions of operations over abstract data types, while algebraic methods (e.g., Act One, Clear and varieties of OBJ) are based on multi- and order-sorted algebras and relate properties of the system to equations over the entities of the algebra.

While both model-oriented and property-oriented approaches have been developed to deal with the specification of sequential systems, process algebras have been developed to meet the needs of concurrent systems. The best known theories in this class are Hoare's Communicating Sequential Processes (CSP) [128, 129] and Milner's Calculus of Communicating Systems (CCS) [185], both of which describe the behavior of concurrent systems by describing their algebras of communicating processes.

It is often difficult to classify specification language, and these categories merely act as guidelines; they are certainly not definitive – some languages are based on a combination of different classes of specification language in an attempt to exploit the advantages of each. The protocol specification language LOTOS [140, 247], for example, is based on a combination of Act One and CCS; while it can be classed as an algebraic method, it certainly exhibits many properties of a process algebra also, and has been successfully used in the specification of concurrent and distributed systems. Similarly, in some ways CSP may be considered to be a process *model* since the algebraic laws of CSP can (and indeed should, for peace of mind) be proven correct with respect to an explicit model [131].

An advantage of reasoning using formal methods is that unlike software testing in general, they can be used effectively on systems with large or infinite states.

Sometimes however, a system may have a sufficiently small state space for an alternative approach to be used. Model checking [179] (with tool support such as SPIN [136] allows exhaustive analysis of a finite system, normally with tool support, for certain types of problem, including requirements [16]. Where this approach can be applied (e.g., for hardware and protocols), it may be deemed the preferred option because of the increased confidence provided by the use of a mechanical tool without the necessity of a large amount of intervention and guidance by the engineer that most more general purpose proof tools require.

In general, model-based formal methods [41] are considered easier to use and to understand; proofs of correctness and refinement, it has been said, are equally difficult for each class. But in some cases, algebraic methods are more appropriate and elegant. In fact, using a library of algebraic laws may help to split the proof task and make some of the proofs reusable. For the future, it is hoped that greater unification of the various approaches will be achieved [134].

In her article *A Specifier's Introduction to Formal Methods* (reprinted here), Jeannette Wing gives an excellent overview of formal methods, and more in-depth detail on the various issues raised above [255]. She describes the differences between the different classes of formal methods and provides some very simple examples to illustrate the differences between different formal methods; the article also includes an excellent classified bibliography.

We saw in Part 1, however, how each specification is in fact a lower level implementation of some other specification. The program itself is also a specification, again written in a formal language. While formal languages of the sort that have been described thus far in this Part are concerned with clarity and simplicity, programs must consider efficiency and the correct implementation of requirements.

As such, while 'broad-spectrum' notations such as Z and VDM can be used at all stages in the development, at the final stages of development, there is a considerable 'jump' from neat mathematical notations to the complex details of optimizing programs. To overcome this, we often require to use different specification languages at different levels of abstraction and at different stages of development. C.A.R. Hoare in *An Overview of Some Formal Methods for Program Design* (reprinted here) describes how a variety of formal methods and notations may be used in the design of a single system, and highlights how functional programming can aid in reducing the size of this 'jump' [130]. A goal is to unify the various programming paradigms [134].

In the article *Ten Commandments of Formal Methods* (reprinted here), the authors discuss a number of maxims which, if followed, may help to avoid some of the common pitfalls that could be encountered in the practical application of formal methods [38]. The future of formal methods is still unsure, but the potential benefits are large if the techniques are incorporated into the design process in a sensible manner [58, 173]. We will discuss how method integration and executable specifications can also help in Parts 7 and 8, respectively.

Seven Myths of Formal Methods (Hall)

TO BE ADDED: Hall: 7 Myths of Formal Methods

TO BE ADDED: 18 pages

Seven More Myths of Formal Methods (Bowen & Hinchey)

TO BE ADDED: Bowen and Hinchey: 7 More Myths of Formal Methods

TO BE ADDED: 14 pages

A Specifier's Introduction to Formal Methods (Wing)

TO BE ADDED: Wing: A Specifier's Introduction to Formal Methods

TO BE ADDED: 34 pages

An Overview of Some Formal Methods for Program Design (Hoare)

TO BE ADDED: Hoare: An Overview of Some Formal Methods

TO BE ADDED: 16 pages

Ten Commandments of Formal Methods (Bowen & Hinchey)

TO BE ADDED: Bowen and Hinchey: Ten Commandments of Formal Methods

TO BE ADDED: 14 pages

4. Object-Orientation

Although object-oriented programming languages appeared in the late 1960s with the advent of Simula-67, it was not until the late 1980s that the object paradigm became popular. Over the last decade we have seen the emergence of many object-oriented programming languages, and extensions to existing languages to support object-oriented programming (e.g., C++, Object Pascal).

The object paradigm offers many distinct advantages both at implementation, and also at earlier phases in the life-cycle. Before we consider these, first let us consider what we mean by Object-Orientation.

4.1 The Object Paradigm

Object-orientation, or the object paradigm, considers the Universe of Discourse to be composed of a number of independent entities (or *objects*). An object provides a *behavior*, which is a set of operations that the object can be requested to carry out on its data. An object's actions are carried out by internal computation, and by requesting other objects to carry out operations. Each object is responsible for its own data (or state), and only the object that owns particular data can modify it. Objects can only effect changes to data owned by another object by sending a message requesting the change.

Thus, an object can be defined as the encapsulation of a set of operations or *methods* which can be invoked externally, and of a state which remembers the effect of the methods.

Object-oriented design [60] bases the modular decomposition [203] of a software system on the classes of objects that the system manipulates, not on the function the system performs. Abstract Data Types (ADTs) are essential to the object-oriented approach, as object-oriented design is essentially the construction of software systems as structured collections of ADT implementations. We call the implementation of an Abstract Data Type a *module*.

The object-oriented approach supports the principle of data abstraction, which encompasses two concepts:

1. Modularization, and
2. Information Hiding.

4.2 Modularization

Modularization is the principle whereby a complex system is sub-divided into a number of self-contained entities or modules, as described above. All information relating to a particular entity within the system is then contained within the module. This means that a module contains all the data structures and algorithms required to implement that part of the system that it represents.

Clearly then, if errors are detected or changes are required to a system, modularization makes it easy to identify where the changes are required, simply by identifying the entity that will be affected. Similarly for reuse, if we can identify entities in existing systems that are to be replicated (perhaps with minor alterations) in a new system, then we can identify potentially reusable code, routines, etc., by finding the module that implements the particular entity. Since the module is self-contained, it can be transported in its entirity to the new system, save for checking that any requests it makes of other modules can be satisfied.

4.3 Information Hiding

The term *Information Hiding* refers to keeping implementation details 'hidden' from the user. The user only accesses an object through a protected interface. This interface consists of a number of operations which collectively define the behavior of the entity. By strictly controlling the entry points to a program, it is not possible for other modules or other programs to perform unexpected operations on data. In addition, other modules are not dependent on any particular implementation of the data structure.

This makes it possible to replace one implementation with another equivalent implementation, and (more importantly) to reuse a module in another system without needing to know how the data structures are actually implemented. We also know that the state of the data is not dependent on any spurious updates from other modules.

The generic term for those techniques that realize data abstraction is *encapsulation*. By encapsulating both the data and the operations on that data in a single, manageable Module (as in Modula-2 and Modula-3), Package (as in Ada) or Unit (as in Object Pascal), we facilitate their transportation and reuse in other systems.

4.4 Classes

Object-oriented systems generally support one or more techniques for the classification of objects with similar behaviors. The concept of a *class* originated in Simula-67, the purpose being to allow groups of objects which share identical behavior to be created.

This is done by providing a number of templates (i.e., classes) for the creation of objects within a system. The class template provides a complete description of a class in terms of its external interface and internal algorithms and data structures. The primary advantage of this approach is that the implementation of a class need only be carried out once. Descriptions of classes may then be reused in other systems, and with specialization may even be reused in the implementation of new classes.

Specialization is the process whereby one creates a new class from an existing class, and is thereby effectively moving a step nearer the requirements of the application domain. A new system can take advantage of classes as they have been defined in other systems, and also specialize these further to suit the requirements.

Specialization can take one of four forms:

1. adding new behavior;
2. changing behavior;
3. deleting behavior;
4. a combination of 1 to 3 above.

Object-oriented languages and systems vary in their support for the different forms of specialization. Most support the addition and changing of behavior, while only a few support the deletion of behavior. Those languages and systems that only allow for the addition of behavior are said to exhibit *strict inheritance*, while those that also support the deletion and changing of behavior are said to exhibit *non-strict inheritance*. From the point of view of software reuse, while support for any form of class inheritance is of great use, languages exhibiting non-strict inheritance offer the greatest possibilities.

4.5 Genericity and Polymorphism

Both Ada and CLU support the use of generic program units (and). This means that instead of writing separate routines to perform the same operation on different types (e.g., sorting an array of integers, sorting an array of reals), only one ‘template’ routine needs to be written, which can then be *instantiated* with the required type. That is, an instance of the generic subprogram or package is created with the actual type replacing the generic type. The instantiation is achieved by passing the required type as a parameter to the generic program unit.

While genericity enables the reuse of a generic ‘template’, and avoids having to write separate routines for each type to be operated on, it does not reduce the number of routines in the system. That is, a separate instance of the instantiated

template exists for each type that is to be handled by the system; one routine does not accommodate more than one type.

One routine accommodating more than one type is termed *polymorphism*. In terms of object-oriented computing it may be defined as the ability of behavior to have an interpretation over more than one class. For example, it is common for the method print to be defined over most of the classes in an object-oriented system. Polymorphism can be achieved in two ways – through sub-classing, and through overloading. As such, it is supported by most object-oriented languages and systems.

Overloading, with which we can achieve polymorphism, is evident in most programming languages. Overloading the meaning of a term occurs when it is possible to use the same name to mean different things. For example, in almost all programming languages, we would expect to be able to use the plus-sign (+) to mean the addition of both fixed-point and floating-point numbers. Many programming languages also interpret the plus-sign as disjunction (logical OR). In this way, we can reuse the code written to perform the addition of two integers A and B, to mean the addition of two real numbers A and B, or the disjunction of A and B (i.e., $A \vee B$), etc.

4.6 Object-Oriented Design

Object-oriented programming languages clearly offer benefits in that they permit a more intuitive implementation of a system based on the real-life interaction of entities, and offer much support for software reuse, avoiding 're-inventing the wheel'.

The object paradigm also offers benefits at earlier stages in the life-cycle, however, facilitating greater savings through the reuse of specification and design components as well as code, and a more intuitive approach to modeling the real-world application.

In his paper *Object-Oriented Development* (reprinted in this Part), which has become perhaps the most widely-referenced paper in the field, Grady Booch describes the motivation for an object-oriented approach to system design, and outlines his own approach to object-oriented development [22]. This approach has become known as the 'Booch approach' or simply *Object-Oriented Design* [23]. The development process is clearly simplified with the use of an object-oriented programming language (such as Ada) for the final implementation, although implementation in languages not supporting the object paradigm is also possible.

Booch is careful to point out that object-oriented design is a partial life-cycle method, that must be extended with appropriate requirements and specification methods. To this end, a large number of object-oriented analysis and design methodologies (e.g., see [263]) have evolved, many based firmly on the various structured methods discussed in Part 2.

These approaches differ greatly, some being minor extensions to existing methods, others being entirely new approaches. In *Object-Oriented and Conventional Analysis and Design Methodologies: Comparison and Critique* (reprinted here), Fichman and Kemerer review approaches to object-oriented analysis and design,

comparing various approaches and the notations and techniques that they use, and concluding that the object-oriented approach represents a radical change from more conventional methods [78].

The formal methods community too have been quick to realize the benefits of object-orientation, and how it can smooth the transition from requirements, through specification, design and finally into implementation in an object-oriented programming language [157]. The object paradigm also greatly facilitates the reuse of formal specifications and of system maintenance at the specification level rather than merely at the level of executable code [30]. A large number of differing approaches to object-orientation in Z and VDM have emerged, with none as yet being taken as definitive. Readers are directed to [159] for overviews of the various approaches.

A number of other object-oriented development approaches exist, such as the Fusion method used at Hewlett-Packard [61]. More recently, Booch *et al.*'s Unified Modeling Language (UML) has been very successful [25, 226]. Although it has been developed by Rational, Inc., it is non-proprietory and is being adopted widely [76]. This language is formalizable [158] and thus may be used as a formal modelling notation [80] although further work in this area would be worthwhile. There are possibilities of combining UML with other formal notations [83]. Project management issues are also very important [24, 224]. Further developments and use of object-oriented approaches look likely for the future.

Object-Oriented Development (Booch)

TO BE ADDED: Booch: Object-Oriented Development

TO BE ADDED: 24 pages

Object-Oriented and Conventional Analysis and Design Methodologies: Comparison and Critique (Fichman & Kemerer)

TO BE ADDED: Fichman and Kemerer: Object-Oriented & Conventional Anal.

TO BE ADDED: 34 pages

5. Concurrent and Distributed Systems

In a concurrent system, two or more activities (e.g., processes or programs) progress in some manner in parallel with each other. A distributed system consists of a number of independent computer systems connected together so that they can cooperate with each other in some manner. Inevitably these two concepts are intertwined.

The last decade and more has seen a rapid expansion of the field of distributed computing systems. The two major forces behind the rapid adoption of distributed systems are *technical* and *social* ones, and it seems likely that the pressure from both will continue for some time yet.

Technical reasons: The two major technical forces are *communication* and *computation*. Long haul, relatively slow communication paths between computers have existed for a long time, but more recently the technology for fast, cheap and reliable *local area networks* (LANs) has emerged and dominated the field of cooperative computing. These LANs allow the connection of large numbers of computing elements with a high degree of information sharing. They typically run at 10–100 Mbits per second (for example, the ubiquitous Ethernet) and have become relatively cheap and plentiful with the advances of microelectronics and microprocessor technology. In response, the *wide area networks* (WANs) are becoming faster and more reliable. Speeds are set to increase dramatically with the use of fiber optics and the introduction of ATM (Asynchronous Transfer Mode) networks.

Social reasons: Many enterprises are cooperative in nature – e.g., offices, multinational companies, university campuses etc. – requiring sharing of resources and information. Distributed systems can provide this either by integrating preexisting systems, or building new systems which inherently reflect sharing patterns in their structure. A further, and somewhat contrary, motivation is the desire for autonomy of resources seen by individuals in an enterprise. The great popularity and ease of use of distributed information systems such as the World Wide Web (WWW) distributed hypermedia system on the Internet [14] are transforming the way in which information is made available for the future.

Distributed systems can offer greater *adaptability* than localized ones. Their nature forces them to be designed in a modular way and this can be used to advantage to allow incremental (and possibly dynamic) changes in performance and functionality by the addition or removal of elements.

The *performance* of distributed systems can be made much better than that of centralized systems and with the fall in the price of microprocessors can also be very much cheaper. However, this performance gain is usually manifested in the form of greater *capacity* rather than *response*. Increasing the latter is limited by the ability to make use of parallelism, which is mainly a software problem, and will probably see most progress in the area of *tightly coupled systems*. These are multiprocessor systems with very fast communications, such as is provided by shared memory systems and Transputer systems. This is in contrast to the *loosely coupled systems*, typified by asynchronous, autonomous computers on a LAN.

Availability can be increased because the adaptability of distributed systems allows for the easy addition of redundant elements. This is a potential benefit which has only been realized in a few systems. In many, the system as a whole becomes less available because it is made dependent on the availability of all of (the large number of) its components. This again is largely a software problem.

5.1 Concurrent Systems

The advent of concurrent systems has complicated the task of system specification and design somewhat. Concurrent systems are inherently more complex, offering the possibility of parallel and distributed computation, and the increased processing power that this obviously facilitates.

Standard methods of specifying and reasoning about computer systems are not sufficient for use with concurrent systems. They do not allow for side-effects, the occurrence of multiple events simultaneously, nor for the synchronization required between processes to ensure data integrity, etc. A specialized environment will normally be required for effective design [188].

A major approach to the handling of concurrency in a formal manner has been the use of process algebras and models. Hoare's CSP (Communicating Sequential Processes) [129, 220] and Milner's CCS (Calculus of Communicating Systems) [185] are perhaps the two foremost and widely accepted examples of such an approach. C.A.R. Hoare's original paper on CSP [128], *Communicating Sequential Processes* is reprinted in this Part since it was seminal to this field. Subsequently the approach has been given a more formal footing [52], and also expanded in several directions to handle real-time, probability, etc. [127, 220].

A more recent paper by Leslie Lamport, *A Simple Approach to Specifying Concurrent Systems*, is also reprinted here, and provides a more current view of the field [152]. Lamport provides an approach to the specification of concurrent systems with a formal underpinning, using a number of examples. The *transition axiom* method described provides a logical and conceptual foundation for the description of con-

currency. Safety and liveness properties are separated for methodological rather than formal reasons.

The method described in this paper has been refined using the formal logic TLA (the Temporal Logic of Actions) [153]. The major advance has been to write this style of specification as a TLA formula, providing a more elegant framework that permits a simple formalization of all the reasoning.

Since the two included papers with this Part are mainly on *concurrent* issues, the rest of this Part redresses the balance by discussing *distributed* systems in more depth. [4] is a major survey which, while quite old, gives a good and authoritative grounding in the concepts behind the programming of concurrent systems for those who wish to follow this area up further.

The future for concurrent systems looks very active [59]. In particular the boundaries of hardware and software are becoming blurred as programmable hardware such as Field Programmable Gate Arrays (FPGA) becomes more prevalent. Hardware is parallel by its very nature, and dramatic speed-ups in naturally parallel algorithms can be achieved by using a hardware/software co-design approach [196].

5.2 Distributed Systems

To consider the issues and problems in designing distributed systems we need to define their fundamental properties. For this we can use the analysis of LeLann [164] who lists the following characteristics:

1. They contain an arbitrary number of processes.
2. They have a modular architecture and possibly dynamic composition.
3. The basic method of communication is message passing between processes.
4. There is some system-wide control.
5. There are variable (non-zero) message delays.

The existence of the last item means that centralized control techniques cannot be used, because there may never be a globally consistent state to observe and from which decisions can be made. Distributed systems carry over most of the problems of centralized systems, but it is this last problem which gives them their unique characteristics.

5.3 Models of Computation

Two popular models for distributed systems are the ‘object–action’ model and the ‘process–message’ model. In the former, a system consists of a set of objects (e.g. files) on which a number of operations (e.g. read and write) are defined; these operations can be invoked by users to change the state of the objects to get work done. In the latter model, a system is a set of processes (e.g. clients and file severs) prepared

to exchange messages (e.g. read file request, write file request); receipt of messages causes processes to change state and thus get work done.

The terminology in this area can be confusing, but 'object–action' systems roughly correspond to 'monitor-based' and 'abstract data type' systems while 'process–message' systems correspond to 'client–server' systems. Lauer and Needham [161] have shown these models to be duals of each other. Most systems can be placed in one of these categories.

This characterization should not be applied too zealously. Both models essentially provide a way of performing computations and it is relatively straightforward to transform one type of system to the other. As already mentioned, because of the nature of the hardware, the basic method of communication in distributed systems is message passing between processes but in many cases the two models can be found at different layers within a system.

5.4 Naming Considerations

Objects and processes have to be named so that they can be accessed and manipulated. Since there are many sorts of objects in a system it is tempting to adopt many sorts of names. This must be tempered by a desire for conceptual simplicity. An important property of names is their scope of applicability; e.g., some may only be unique within a particular machine. Examples of such contexts are names of different sorts of objects which reside in a WAN, on a single LAN, in a service, on a server or somewhere in the memory of a particular machine.

Transforming names from one form to another is a very important function, even in centralized systems. One example is transforming a string name for a UNIX file to an *inode* number. Because distributed systems are so much more dynamic it is even more important to delay the binding of 'logical' and 'physical' names. Some sort of *mapping* or *name service* is required. Since this is such an important function, availability and reliability are vital. The domains and ranges of the mapping are often wide but sparse so in many systems names are structured to allow more efficient searches for entries.

5.5 Inter-Process Communication

The exchange of messages between processes – Inter-Process Communication (IPC) – is the basic form of communication in distributed systems, but note that it is easy to build the actions of object–action systems on top of this mechanism. For these, the idea of *remote procedure call* (RPC) forms a useful extension from the world of single machine local procedure calls. How transparent these should be from issues of location and errors is a matter of some debate. Whether RPCs or messages are used, the destination has to be located, using the mapping facilities already sketched. The use of RPCs has become popular because they represent an easy to use, familiar

communication concept. They do not, however, solve all problems, as discussed in [242].

The structure of units of activity is very important. Work in recent years (in centralized as well as distributed systems) has led to the realization that a single kind of activity is inadequate in system design. Many have adopted a two level structuring. The outer level, called *heavy-weight processes* (HWPs), representing complete address spaces, are relatively well protected and accounted for, but switching between them is slow because of the large amount of state involved. Distribution is performed at the level of HWPs. They are typified by UNIX processes.

Within an HWP, a number of *light-weight processes* (LWPs) operate; these have much less state associated with them, are unprotected, share the address space of the containing HWP but it is possible to switch between these relatively rapidly. For example, there may be a number of HWPs on each node. One of these may represent a server. The server may be structured as a number of LWPs, one dedicated to the service of each client request.

5.6 Consistency Issues

The abnormal (e.g., crashes) and normal (e.g., concurrent sharing) activity of a system may threaten its consistency. The problem of concurrency control is well-known in multiprocessing systems. This problem and its solutions becomes harder when the degree of sharing and the amount of concurrency increase in distributed systems. In particular, the lack of global state first of all makes the solutions more difficult and also introduces the need for replication which causes more consistency problems.

Maintaining consistency requires the imposition of some ordering of the events within a system. The substantial insight of Lamport [151] is that the events in a distributed system only define a *partial order* rather than a *total order*. Required orderings can be achieved by extending existing centralized mechanisms, such as *locking*, or using *time-stamp* based algorithms.

Making systems resilient to faults in order to increase their reliability is an often quoted, rarely achieved feature of distributed systems. Both of these issues have been tackled with the notion of *atomic actions*. Their semantics are a conceptually simple ‘all or nothing’, but their implementation is rather harder, requiring essentially images of ‘before’ states should things prove sufficiently difficult that the only option is to roll back. This gets much harder when many objects are involved. Much work from the database world, particularly the notions of *transactions* and *two-phase commit* protocols have been adapted and extended to a distributed world. Atomic actions match quite well to RPC systems that provide ‘at-most-once’ semantics.

Another source of problems for consistency is dynamic reconfiguration of the system itself, adding or perhaps removing elements for maintenance while the system is still running.

5.7 Heterogeneity and Transparency

Heterogeneity must often be handled by operating systems (OSs) and particularly by distributed operating systems. Examples of possible heterogeneity occur in network hardware, communication mechanisms, processor architectures (including multiprocessors), data representations, location etc. In order that such disparate resources should not increase system complexity too much, they need to be accommodated in a coherent way. The key issue of such systems is *transparency*. How much of the heterogeneity is made visible to the user and how much is hidden depends very much on the nature of the heterogeneity. Other factors apart from heterogeneity can be made transparent to users, such as plurality of homogeneous processors in, for example, a processor pool.

The choice to be made is generally one of providing users with conceptually simpler interfaces, or more complicated ones which allow for the possibility of higher performance. For example, if location of processes is not transparent and can be manipulated, users can co-locate frequently communicating elements of their applications. If location is transparent the system will have to infer patterns of communication before it can consider migrating processes.

Issues of transparency cut across many other issues (e.g. naming, IPC and security) and need to be considered along with them.

5.8 Security and Protection

A *security policy* governs who may obtain, and how they may modify, information. A *protection mechanism* is used to reliably enforce a chosen security policy. The need to identify users and resources thus arises.

The distribution of a system into disjoint components increases the independence of those components and eases some security problems. On the other hand, a network connecting these components is open to attack, allowing information to be tapped or altered (whether maliciously or accidentally), thus subverting the *privacy* and *integrity* of their communication. In such a situation each component must assume much more responsibility for its own security.

Claims about identity must be *authenticated*, either using a local mechanism or with some external agency, to prevent impersonation. The total security of a system cannot rely on all the kernels being secure since there may be many types of these (for all the different processors) and a variable number of instances of them. It would be easy to subvert an existing instance or to insert a new one. Thus authentication functions should be moved out of kernels.

Access control is a particular security model. There exists a conceptual matrix which identifies the *rights* that *subjects* (e.g. users, processes, process groups) have over all *objects* (e.g. data, peripherals, processes). Two popular realizations of this are *access control lists* (ACLs) and *capabilities*. In the former, an ACL is associated with each object, listing the subjects and their rights over it. In the latter, each subject possesses a set of capabilities each of which identifies an object and the rights

that the subject has over it. Each realization has its own merits and drawbacks, but capability based distributed systems seem to be more numerous.

Capabilities must identify objects and rights, so it is natural to extend a naming scheme to incorporate this mechanism. They must also be difficult to forge, which is harder to achieve without a trusted kernel. Drawing them from a sparse space and validating them before use allows them to be employed in distributed systems.

Cryptographic techniques can be used to tackle a number of these problems including ensuring privacy and integrity of communications, authentication of parties and digital signatures as proof of origins [190]. VLSI technology is likely to make their use, at least at network interfaces, standard, though they can be used at different levels in the communications hierarchy for different purposes.

Many of these techniques depend on the knowledge that parties have about encryption *keys* and thus *key distribution* becomes a new problem which needs to be solved. Use of *public key* systems can ease this, but many systems rely on some (more or less) trusted registry of keys, or an authentication server. The parallels with name servers again show the interdependence of naming and security.

5.9 Language Support

The trend has been for generally useful concepts to find themselves expressed in programming languages, and the features of distributed systems are no exception. There are many languages that incorporate the notions of processes and messages, or light-weight processes and RPCs.

Other languages are object-oriented and can be used for dealing with objects in a distributed environment and in others still the notion of atomic actions is supported. A choice must be made between the shared variable paradigm and the use of communication channels, as adopted by Occam [139] for example. The latter seems to be more tractable for formal reasoning and thus may be the safest and most tractable approach in the long run.

The problem of separately compiled, linked and executed cooperating programs are to some extent a superset of those already found in programs with separately compiled modules in centralized systems. The notion of a server managing a set of objects of a particular abstract data type and its implementation as a module allows module interfaces to be used as the unit of binding before services are used.

5.10 Distributed Operating Systems

Distributed Operating Systems (DOSs) have been the subject of much research activity in the field of distributed systems. Operating systems (OSs) control the bare resources of a computing system to provide users with a more convenient abstraction for computation. Thus the user's view of a system is mostly determined by the OS. A DOS is an OS built on distributed resources. There is much argument over what abstraction for computation a DOS should provide for its users.

The term *network operating system* (NOS) has faded in its usage, but requires some mention in the history of the development of distributed systems. Like many terms it means different things to different people. For many, a NOS is a 'guest level extension' applied to a number of existing centralized operating systems which are then interconnected via a network. These systems are characterized by the high degree of autonomy of the nodes, the lack of system-wide control and the non-transparency of the network.

On the other hand, DOSs are normally systems designed from scratch to be integrated and exercise much system-wide control. Others distinguish the two not on the lines of implementation but on the lines of the view of computation provided to users: in a DOS it is generally transparent while in a NOS it is not; special utilities are needed to use network facilities. See [34] for some examples of distributed operating systems. Further information on distributed systems in general may be found in [189], and [267] contains large bibliographies on the subject.

Communicating Sequential Processes (Hoare)

TO BE ADDED: Hoare: Communicating Sequential Processes

TO BE ADDED: 28 pages

A Simple Approach to Specifying Concurrent Systems (Lamport)

TO BE ADDED: Lamport: A Simple Approach to Specifying Concurrent Systems

TO BE ADDED: 28 pages

6. Real-Time and Safety-Critical Systems

A system is one in which the timing of the output is significant [195]. Such a system accepts inputs from the 'real world' and must respond with outputs in a timely manner (typically within milliseconds – a response time of the same order of magnitude as the time of computation – otherwise, for example, a payroll system could be considered 'real-time' since employees expect to be paid at the end of each month). Many real-time systems are *embedded* systems, where the fact that a computer is involved may not be immediately obvious (e.g., a washing machine). Real-time software often needs to be of high integrity [10].

The term *safety-critical system* has been coined more recently as a result of the increase in concern and awareness about the use of computers in situations where human lives could be at risk if an error occurs. Such systems are normally real-time embedded systems. The use of software in safety-critical systems has increased by around an order of magnitude in the last decade and the trend sees no sign of abating, despite continuing worries about the reliability of software [172, 92].

The software used in computers has become progressively more complex as the size and performance of computers has increased and their price has decreased [216]. 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. Some have attempted to use a 'safe' subset of languages known to have problematic features [8, 113]. In any case, when developing safety-critical systems, a *safety case* should be made to help ensure the avoidance of dangerous failures [254].

6.1 Real-Time Systems

Real-time systems may be classified into two broad types. *Hard real-time* systems are required to meet explicit timing constraints, such as responding to an input within a certain number of milliseconds. The temporal requirements are an essential part of the required behavior, not just a desirable property. *Soft real-time* systems relax this requirement somewhat in that, while they have to run and respond in

real-time, missing a real-time deadline occasionally only causes degradation of the system, not catastrophic failure.

In the first paper in this Part [200], Jonathan Ostroff considers the specification, design and verification of real-time systems, particularly with regard to *formal methods*, with mathematical foundation. He addresses the application of various formal methods that are suitable for use in the development of such systems, including the difficulties encountered in adopting a formal approach.

As well as the formal correctness of a real-time system [120, 147], there are other issues to consider. Predicting the behavior of a real-time system can be problematic, especially if interrupts are involved. The best approach is to ensure predictability by constructing such systems in a disciplined manner [104]. Often real-time systems incorporate more than one computer and thus all the difficulties of a parallel system must also be considered as well, while still trying to ensure system safety, etc. [238].

Hybrid systems [102] generalize on the concept of real-time systems. In the latter, real-time is a special continuous variable that must be considered by the controlling computer. In a hybrid system, other continuous variables are modeled in a similar manner, introducing the possibility of differential equations, etc. Control engineers may need to interact effectively with software engineers to produce a satisfactory system design. The software engineer involved with such systems will need a much broader education than that of many others in computing.

6.2 Safety-Critical Systems

The distinguishing feature of safety-critical software is its ability to put human lives at risk. Neumann has cataloged a large number of accidents caused by systems controlled by computers, many as a result of software problems [192] and software failures are an issue of continuing debate [114]. One of the most infamous accidents where software was involved is the Therac-25 radiotherapy machine which killed several people [169]. There was no hardware interlock to prevent overdosing of patients and in certain rare circumstances, the software allowed such a situation to occur.

The approaches used in safety-critical system development depends on the level of risk involved, which may be categorized depending on what is acceptable (both politically and financially) [12]. The analysis, perception and management of risk is an important topic in its own right, as one of the issues to be addressed when considering safety-critical systems [221].

The techniques that are suitable for application in the development of safety-critical systems are a subject of much debate. The following extract from the BBC television program *Arena* broadcast in the UK during October 1990 (quoted in [45]) illustrates the publicly demonstrated gap between various parts of the computing industry, in the context of the application of formal methods to safety-critical systems:

> Narrator: '*. . . 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: '*If you can't write down a mathematical description of the behavior 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 behavior. 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 sort of exchange is typical of the debate between the various software engineering factions involved with safety-critical systems [209, 211, 212]. As indicated above, some suggest that formal methods, based on mathematical techniques, are a possible solution to help reduce errors in software, especially in safety-critical systems ehere correctness is of prime importance [9]. 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. In any case, assessment of approaches to software development, as well as safety-critical software itself [213], is required to determine the most appropriate techniques for use in the production of future software for high-integrity systems [166, 174].

6.3 Formal Methods for Safety-Critical Systems

As has previously been mentioned, the take up of formal methods is not yet great in industry, but their use has often been successful when they have been applied appropriately [243]. 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 [63] provides a snapshot view of the situation by comparing some significant projects which have made serious use of such techniques. Some sections of industry are applying formal methods effectively and satisfactorily to safety-critical systems (e.g., see [66, 123, 225]). Medical applications are an example sector where formal methods are being considered and used (see for example, [144, 149]).

[46], included in this Part, provides a survey of selected projects and companies that have used formal methods in the design of safety-critical systems, as well as

a number of safety-related standards. In critical systems, reliability and safety are paramount, although software reliability is difficult to assess [54]. Extra cost involved in the use of formal methods is acceptable, and the use of mechanization for formal proofs may be worthwhile for critical sections of the software, and at various levels of abstraction [27, 118]. 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 rigorous proofs or just specification alone. Formal documentation of key components may provide significant benefits to the development of many industrial software-based systems.

The *human-computer interface* (HCI) [71] is an increasingly important component of most software-based systems, including safety-critical systems. Errors often occur due to misunderstandings caused by poorly constructed interfaces [162]. It is suspected that the 'glass cockpit' interface of fly-by-wire aircraft may be a contributing factor in some crashes, although absolute proof of this is difficult to obtain.

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 [202]. 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 [112].

6.4 Standards

Until the last decade, there have been few standards concerned specifically with software for safety-critical systems. Now a plethora are in the process of being or have recently been introduced or revised [250], as well as even more that are applicable to the field of software engineering in general [180].

An important trigger for the exploitation of research into new techniques such as formal methods could be the interest of regulatory bodies or standardization committees (e.g., the *International Electrotechnical Commission*). A significant number of emerging safety-related standards are now explicitly mentioning formal methods as a possible approach for use on systems of the highest integrity levels [26, 37]. Many are strongly recommending the use of formal methods, requiring specific explanation if they are not used.

A major impetus has already been provided in the UK by promulgation of the proposed MoD Interim Defence Standard 00-55 [186], the draft of which mandates the use of formal methods, languages with sound formal semantics and at least a *rigorous argument* to justify software designs. The related Interim Defence Standard 00-56 has itself been subjected to formal analysis [258].

It is important that standards should not be prescriptive, or that parts that are prescriptive should be clearly separated and marked as such. Goals should be set and the onus placed on the software supplier to demonstrate 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.

6.5 Legislation

Governmental legislation is likely to provide increasing motivation to apply appropriate techniques in the development of safety-critical systems. For example, the Machine Safety Directive, legislation issued by the European Commission, has been effective from January 1993. 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. A maximum penalty of three months imprisonment or a large fine are possible. Suppliers will have to demonstrate that they are using best working practice to avoid conviction.

However, care should be taken in not overstating the effectiveness of a particular technique. For example, 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 [177]. If extravagant claims are made, it is quite possible that a similar case could occur again. The UK MoD 00-55 Interim Defence Standard [186] differentiates between *formal proof* and *rigorous argument*, 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.

6.6 Education and Professional Issues

Most modern comprehensive standard textbooks on software engineering aimed at computer science undergraduate courses now include information on real-time and safety-critical systems (e.g., see [233]). However there are not many textbooks devoted solely to these topics, although they are becoming available (e.g., see [167, 240]). Real-time and safety-critical issues are seen as specialist areas, although this is becoming less so as the importance of safety-critical systems increases in the field of software engineering.

Undergraduate computer science courses normally includes a basic introduction to the relevant mathematics (e.g., discrete mathematics such as set theory and predicate logic [204, 99]), which is needed for accurate reasoning about computer-based systems (see, for example, [65, 86]). This is especially important for safety-critical systems where all techniques for reducing faults should be considered [135]. This

will help improve matters in the future, although there is a long lag time between education and practical application. It is important to combine the engineering aspects with the mathematical underpinning in courses [205, 207, 208, 87]. Unfortunately, this is often not the case.

In the past, it has been necessary for companies to provide their own training or seek specialist help for safety-critical system development, although relevant courses are quite widely available from both industry and academia in some countries. For example, see a list of formal methods courses using the Z notation in the UK as early as 1991 [193]. Particular techniques such as formal methods may be relevant for the highest level of assurance [227, 244, 256].

There are now more specialized advanced courses (e.g., at Masters level) specifically aimed at engineers in industry who may require the addition of new skills for application in the development of safety-critical systems. For example, see the modula MSc in safety-critical systems engineering at the University of York in the UK [248]. Engineers can take a number of intensive modules, each typically of a week in length. When enough have been undertaken, an MSc degree can be awarded. It is normally easier for working engineers to take a full week off work every so often rather than individual days for an extended period.

Accreditation of courses by professional societies is increasingly important. For example, in the UK, the British Computer Society (BCS) insists on certain subject areas being covered in an undergraduate computer science course to gain accredation (and hence eased membership of the society by those who have undertaken the course through exemption from professional examinations). Most universities are keen for their courses to comply if possible, and will normally adapt their course if necessary to gain accreditation. This gives professional bodies such as the BCS considerable leverage in what topics are included in the majority of undergraduate computer science courses in a particular country.

As well as technical aspects of computing, computer ethics [29] and related professional issues [89] must also be covered (e.g., see [6]). Liability, safety and reliability are all increasingly important areas to be considered as the use of computers in safety-critical systems becomes more pervasive [235]. Software engineers should be responsible for their mistakes if they occur through negligence rather than genuine error [116] and should normally follow a code of ethics [97] or code of conduct. These codes are often formulated by professional societies and members are obliged to follow them.

There are suggestions that some sort of certification of safety-critical system developers should be introduced. The *licensing* of software engineers by professional bodies has been a subject for discussion in the profession [11]. This is still an active topic for debate. However there are possible drawbacks as well as benefits in introducing tight regulations since suitably able and qualified engineers may be inappropriately excluded and under-qualified engineers may be certified without adequate checks on their continued performance and training.

6.7 Technology Transfer

Technology transfer is often fraught with difficulties and is inevitably (and rightly) a lengthy process. Problems and misunderstandings at any stage can lead to overall failure [211, 228]. A technology should be well established before it is applied, especially in critical applications where safety is of great importance. Awareness of the benefits of relevant techniques [3] must be publicized to a wide selection of both technical and non-technical people, especially outside the research community (e.g., as in [237]). The possibilities and limitations of the techniques available must be well understood by the relevant personnel to avoid costly mistakes. Management awareness and understanding of the effects of a particular technique on the overall development process is extremely important.

Journals and magazines concerned with software engineering in general have produced special issues on safety-critical systems which will help to raise awareness among both researchers and practitioners (for example, see [150, 170]). Awareness of safety issues by the general public is increasing as well [75, 198]. This Part includes two related articles from the former, reporting on a major survey of projects using formal methods [63], many concerned with safety-critical systems. The first article [90] reports on the general experience industry has had in applying formal methods to critical systems. The second article [91] covers four specific instances in more detail. Each has been successful to varying degrees and for various aspects of application. The article provides a useful comparison of some of the possible approaches adopted in each of the presented studies. The third article, by Leveson and Turner, gives a complete account of the investigation into the accidents, and subsequent deaths, caused by the Therac-25 radiation therapy machine. The investigation indicates how testing can never be complete, due to the impact that the correction of one 'bug' can have on other parts of the system.

Organizations such as the *Safety-Critical Systems Club* in the UK [219] have organized regular meetings which are well attended by industry and academia, together with a regular club newsletter for members. Subsequently the *European Network of Clubs for REliability and Safety of Software* (ENCRESS) has been formed to coordinate similar activities through the European Union (e.g., see [101]). The *European Workshop on Industrial Computer Systems, Technical Committee 7* (EWICS TC7) on Reliability, Safety and Security, and the *European Safety and Reliability Association* also have an interest in safety-critical systems. There seems to be less coordinated activity in the US of this nature, although the IEEE provides some support in the area of standards.

Unfortunately, the rapid advances and reduction in cost of computers in recent years has meant that time is not on our side. More formal techniques are now sufficiently advanced that they should be considered for selective use in software development for real-time and safety-critical systems, 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.

Software standards, especially those concerning safety, are likely to provide a motivating force for the use of emerging techniques, and it is vital that sensible and

realistic approaches are suggested in emerging and future standards. 00-55 [186] provides one example of a forward-looking standard that may indicate the direction of things to come in the development of real-time safety-critical systems.

Formal Methods for the Specification and Design of Real-Time Safety-Critical Systems (Ostroff)

TO BE ADDED: Ostroff: Formal Methods for the Specification and Design ...

TO BE ADDED: 46 pages

Experience with Formal Methods in Critical Systems (Gerhart, Craigen & Ralston)

TO BE ADDED: Gerhart et al: Experience with FMs in Critical Systems

TO BE ADDED: 16 pages

Regulatory Case Studies (Gerhart, Craigen & Ralston)

TO BE ADDED: Gerhart et al: Regulatory Case Studies

TO BE ADDED: 18 pages

Medical Devices: The Therac-25 Story (Leveson)

TO BE ADDED: Leveson: Therac-25 accidents

TO BE ADDED: 38 pages

Safety-Critical Systems, Formal Methods and Standards (Bowen & Stavridou)

TO BE ADDED: Bowen and Stavridou: Safety-Critical Systems, Formal Methods and Standards

TO BE ADDED: 44 pages

7. Integrating Methods

7.1 Motivation

In general, no one method is the best in any given situation. Often it is advantageous to use a combination of methods for a particular design, applying each technique in a manner to maximize its benefits [53]. It is possible to integrate the use of formal methods with other less formal techniques. Indeed, often, formal methods provide little more than a mathematical notation, perhaps with some tool support. Combining such a notation with a methodological approach can be helpful in providing a rigorous underpinning to system design.

A number of successful applications of to real-life problems (see, for example, papers in [125]), have helped to dispel the myth that formal methods are merely an academic exercise with little relevance to practical system development. Increasingly, the computer industry is accepting the fact that the application of formal methods in the development process (particularly when used with safety-critical systems) can ensure an increase in levels of confidence regarding the 'correctness' of the resulting system, while improving complexity control, and in many cases reducing development costs (again, see [125]).

Formal methods have now been used, to some extent at least, in many major projects. This trend seems set to continue, as a result of decisions by the UK Ministry of Defence, and other government agencies, to mandate the use of formal methods in certain classes of applications [37].

That is not to say that formal methods have been universally accepted. Many agree that formal methods are still not employed as much in practice as they might be, or as they *should* be [182, 191]. A lot of software development is still conducted on a completely *ad hoc* basis. At best it is supported by various structured methods; at worst, it is developed using a very naïve approach – i.e., the approach taken by many undergraduates when they first learn to program: 'write the program and base the design on this afterwards'.

Structured methods are excellent for use in requirements elicitation, and interaction with system procurers. They offer notations that can be understood by non-specialists, and which can be offered as a basis for a contract. In general, they support all phases of the development life-cycle, from requirements analysis, through specification, design, implementation, and maintenance. However, they offer no

means of reasoning about the validity of a specification – i.e., whether all requirements are satisfied by the specification, or whether certain requirements are mutually exclusive. Unsatisfied requirements are often only discovered post-implementation; conflicting requirements may be detected during implementation.

Formal methods, on the other hand, allow us to reason about requirements, their completeness, and their interactions. They also enable *proof of correctness* – that is, that an implementation satisfies its specification. The problem is that formal methods are perceived to be difficult to use due to their high dependency on mathematics. Certainly personnel untrained in mathematics and the techniques of formal methods will be loath to accept a formal specification presented to them without a considerable amount of additional explanation and discussion. But, as Hall [105] points out, one does not need to be a mathematician to be able to use and understand formal methods. Most development staff should have a sufficient grounding in mathematics to enable them to understand formal specifications and indeed to write such specifications. Formal proofs and refinement (the translation of a specification into a lower level implementation) do require a considerable degree of mathematical ability, however, as well as a great deal of time and effort.

7.2 Integrating Structured and Formal Methods

In the traditional (structured) approach to software development, problems are analyzed using a collection of diagrammatic notations, such as *Data-Flow Diagrams* (DFDs), *Entity-Relationship-Attribute Diagrams* (ERADs) and *State-Transition Diagrams* (STDs). In general, these notations are informal, or, at best, semi-formal, although work on making them more formal is in progress [7]. Only after the problem has been analyzed sufficiently are the possible sequences of operations considered, from which the most appropriate are chosen.

When using formal specification techniques, however, personnel must begin to think in terms of the derivation of a model of reality (either explicit or implicit – depending on the formal specification language being used, and the level of abstraction of the specification). In the specification of a functional system (one in which output is a relation over the current state and the input) this involves relating inputs to outputs by means of predicates over the state of the system. In reactive systems, of which concurrent, real-time and distributed systems are representative, the specification is complicated by the need to consider side-effects, timing constraints, and fairness. In either case, the mismatch, or 'gap' between the thought processes that are required at the analysis stage and those needed to formally specify a system is significant, and has been termed the *Analysis–Specification Semantic Gap* [214].

In traditional software development, a high-level specification is translated to a design incrementally, in a process known as *stepwise refinement*. This process continues until the design is couched in such terms that it can be easily implemented in a programming language. Effectively what the system designer is doing is deriving an implicit tree of possible implementations, and performing a search of the possibilities, eliminating those which are infeasible in the current development environment,

and selecting the most appropriate of the rest. The tree is potentially (and normally) infinite (up to renaming) and so an explicit tree is never derived.

In implementing a formal specification, however, the developer must change from the highly abstract world of sets, sequences and formal logic, to considering their possible implementations in terms of a programming language. Very few programming languages support such constructs, and certainly not efficiently. As a result, this requires determining the most appropriate data structures to implement the higher level entities (*data refinement*), and translating the operations already defined to operate on arrays, pointers, records, etc., (*operation refinement*).

The disparity here has been termed the *Specification–Implementation Semantic Gap*, and is clearly exacerbated by the lack of an intermediate format. Such a 'gap' represents a major difficulty for the software engineering community. Suggestions for its elimination vary greatly . . . from the introduction of programming languages supporting higher-level primitives [138], to the use of specification languages which have executable subsets [84], e.g., CSP [129, 127] with Occam [139], OBJ [94], or with inherent tool support, e.g., Larch [103], Nqthm [48], B [2], PVS [201]. The ProCoS project [43] on 'Provably Correct Systems' has been exploring the formal foundations of the techniques to fill in these gaps from requirements through specification, design, compilation [118, 239], and ultimately down to hardware [33, 155], but many problems remain to be solved, especially those concerning scaling.

A means by which structured and formal methods are integrated to some extent could help in systems of an industrial scale [100]. A method based on such an integration would offer the best of both worlds:

- it offers the structured method's support for the software life cycle, while admitting the use of more formal techniques at the specification and design phases, supporting refinement to executable code, and proof of properties;
- it presents two different views of the system, allowing different developers to address the aspects that are relevant to them, or of interest to them;
- it provides a means of formally proving the correctness of an implementation *with respect to its specification*, while retaining a structured design that will be more acceptable to non-specialists;
- the formal method may be regarded as assigning a formal semantics to the structured method, enabling investigations of its appropriateness, and the study of possible enhancements;
- the structured design may be used as the basis for insights into the construction of a formal specification.

The final point above is quite contentious. A number of people have cited this as a disadvantage of the approach and something that should not be encouraged. The view is that such an approach severely restricts levels of abstraction and goes against many of the principles of formal specification techniques. On the other hand, there is a valid argument that such an approach is often easier for those unskilled in the techniques of formal specification to follow, and can aid in the management of size and complexity, and provide a means of structuring specifications.

7.3 An Appraisal of Approaches

A number of experiments have been conducted in using formal methods and structured methods in parallel. Leveson [165], Draper [72] and Bryant [5] all report successes in developing safe and reliable software systems using such techniques. In such cases, non-specialists were able to deal with more familiar notations, such as DFDs (Data-Flow Diagrams), while specialists concentrated on more formal investigations, highlighting ambiguities and errors admitted by more conventional methods.

There is a severe restriction, however, in that the conventional specification and design, and the formal approach will be engaged upon by different personnel. With differing levels of knowledge of the system and implementation environment, and lack of sufficient feedback between the two groups, it is unlikely that the benefits of such an approach will be adequately highlighted. In fact, Kemmerer [148] warns of potential negative effects.

A more integrated approach, whereby formal methods and more traditional techniques are applied in a unified development method, offers greater prospects for success. A number of groups have been working on such integrated methods, their approaches varying greatly – from transliterations of graphical notations into mathematical equivalents, to formalizing the transformations between both representations.

Semmens, France and Docker, in their paper *Integrated Structured Analysis and Formal Specification Techniques* (reprinted here) give an excellent and very complete overview of the various approaches to method integration, both in academia and in industry [229].

Integrated Structured Analysis and Formal Specification Techniques
(Semmens, France & Docker)

TO BE ADDED: Semmens, France and Docker: Integrated SA and FST

TO BE ADDED: 24 pages

8. Implementation

8.1 Refinement

As we discussed in earlier Parts, formal specifications are expressed at high levels of abstraction and in terms of abstract mathematical objects, such as sets, sequences and mappings, and with an emphasis on clarity rather than efficiency. But we want our programs to be efficient and since most programming languages do not support these abstract data types, how do we use the formal specification in system development?

Just as in traditional design methods, such as SSADM and Yourdon, we gradually translate our formal specification into its equivalent in a programming language. This process is called *refinement* (or *reification* in VDM). *Data refinement* involves the transition from abstract data types to more concrete data types such as record-structures, pointers and arrays, and the verification of this transition.

All operations must then be translated so that they now operate on the more concrete data types. This translation is known as *operation refinment* or *operation modeling*, and gives rise to a number of *proof obligations* that each more concrete operation is a *refinement* of some abstract equivalent. By 'refinement', we mean that it performs *at least* the same functions as its more abstract equivalent, but is in some sense 'better' – i.e., more concrete, more efficient, less non-deterministic, terminating more often, etc.

These proof obligations may be discharged by constructing a *retrieve function* for each operation which enables us to return from an operation to its more abstract equivalent. In VDM, we must also construct an *abstraction function* which does the opposite, and brings us from the abstract operation to the more concrete one. More generally there may be a retrieve relation [257].

The refinement process is an iterative one, as shown in Figure 8.1. Except for simple problems, we would never go straight from an abstract specification directly to code. Instead, we translate our data types and operations into slightly more concrete equivalents at each step, with the final step being the translation into executable code.

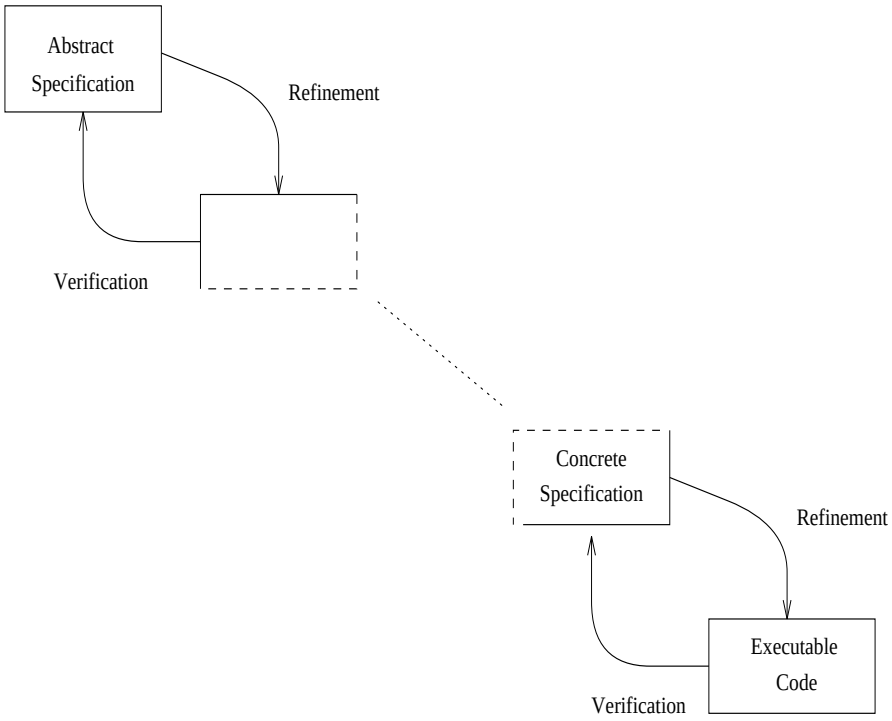

Figure 8.1. The refinement process

8.2 Rapid Prototyping and Simulation

Rapid System Prototyping (RSP) and simulation have much in common in the sense that both involve the derivation and execution of an incomplete and inefficient version of the system under consideration. They do, however, have different aims (although these are certainly not incompatible), and are applied at different stages in the system life-cycle.

Prototyping is applied at the earlier stages of system development as a means of *validating* system requirements. It gives the user an opportunity to become *au fait* with the 'look-and-feel' of the final system, although much of the logic will still not have been implemented. The aim is to help in determining that the developer's view of the proposed system is coincident with that of the users. It can also help to identify *some* inconsistencies and incompatibilities in the stated requirements. It cannot, for example, be used to determine whether the requirements of efficiency of operation and requirements of ease of maintenance are mutually satisfiable. The prototype will in general be very inefficient, and will not necessarily conform to the stated design objectives.

Best practice holds that the code for a prototype should be discarded before implementation of the system. The prototype was merely to aid in eliciting and determining requirements and for *validation* of those requirements; that is, determining that we are building the 'right' system [19]. It may have a strong *bias* towards particular implementations, and using it in future development is likely to breech design goals, resulting in an inefficient implementation that is difficult to maintain. Retaining a prototype in future development is effectively equivalent to the transformational or evolutionary approach described above, with a certain degree of circumvention of the specification and design phases.

Simulation fits in at a different stage of the life-cycle. It is employed after the system has been specified, to *verify* that an implementation may be derived that is consistent both with the explicitly stated requirements, and with the system specification; in other words, that we are building the system 'right' [19]. While prototyping had the aim of highlighting inconsistencies in the requirements, simulation has the aim of highlighting requirements that are left unsatisfied, or only partly satisfied.

Both rapid prototyping and simulation suffer from one major drawback. Like testing, which can only highlight the presence of software bugs, but not their absence [70], prototyping and simulation can only demonstrate the existence of contradictory requirements or the failure to fully satisfy particular requirements. They cannot demonstrate that no contradictory requirements exist, nor that all specified requirements are satisfied, respectively [126]. That is why attention has begun to be focused on the use of formal methods in both Rapid System Prototyping and simulation, as formal methods can actually augment both of these areas with *proof* [121].

The use of executable specification languages and the animation of formal specifications are clearly two means of facilitating prototyping and simulation, while retaining the ability to prove properties.

8.3 Executable Specifications

We make a distinction between the concept of executable specifications and that of specification , although many authors consider them to be identical.

In our view, specifications are 'executable' when the specification language inherently supports explicit execution of specifications. While the means by which executions of such specifications are performed are varied and interesting in themselves, they are not of concern to us here.

An executable specification language offers one distinct advantage – it augments the conceptual model of the proposed system, derived as part of the system specification phase, with a behavioral model of that same system. As Norbert Fuchs points out in his paper *Specifications are (Preferably) Executable* (reprinted in this Part), this permits validation and verification (as appropriate) at earlier stages in the system development than when using traditional development methods [84].

There is a fine line between executable specifications and actual implementations – that of resource management [266]. While a good specification only deals with the functionality and performance properties of the system under consideration, implementations must meet performance goals in the execution environment through the optimal use of resources.

In their paper *Specifications are not (Necessarily) Executable* (also reprinted in this Part), Ian Hayes and Cliff Jones criticize the use of executable specifications on the grounds that they unnecessarily constrain the range of possible implementations [117]. While specifications are expressed in terms of the problem domain in a highly abstract manner, the associated implementation is usually much less 'elegant', having, as it does, to deal with issues of interaction with resources, optimization, meeting timing constraints, etc. Hayes and Jones claim that implementors may be tempted to follow the algorithmic structure of the executable specification, although this may still be far from the ideal, producing particular results in cases where a more implicit specification would have allowed a greater range of results.

They also claim that while executable specifications can indeed help in early validation and verification, it is easier to prove the correctness of an implementation with respect to a highly abstract equivalent specification rather than against an implementation with different data and program structures. This is crucial; it indicates that executable specifications, while permitting prototyping and simulation, in the long run may hinder *proof of correctness*.

8.4 Animating Formal Specifications

While executable specifications incorporate inherent support in the specification language, animation applies to specification languages which are not normally executable. In this category we include the animation of Z in Prolog [252] (logic programming [85]) and Miranda [49] (functional programming), and the direct translation of VDM to SML (Standard ML) [197], as well as the interpretation and compilation of Z as a set-oriented programming language [249], etc.

Specification languages such as VDM and Z are not *intended* to be directly executable, but by appropriately restating them directly in the notation of a declarative programming language, become so. And, as Fuchs illustrates in his paper *Specifications are (Preferably) Executable*, with appropriate manipulations such animations can be made reasonably efficient.

This approach seems preferable to executable specification languages. It too provides a behavioral model of the system, but without sacrificing abstraction levels. It supports rapid prototyping and even more powerful simulation, but prototypes and simulations are not used in future development. The refinement of the specification to a lower-level implementation, augmented with the discharge of various *proof obligations* ensures that the eventual implementation in a conventional (procedural) programming language satisfies the original specification.

Specifications are not (Necessarily) Executable (Hayes & Jones)

TO BE ADDED: Hayes and Jones: Specifications are not executable

TO BE ADDED: 20 pages

Specifications are (Preferably) Executable (Fuchs)

TO BE ADDED: Fuchs: Specifications are executable

TO BE ADDED: 26 pages

9. CASE

Just as mechanization made the Industrial Revolution in Britain possible at the beginning of the 19th century, so too is mechanization in system development seen as a means to increased productivity and a 'Systems Revolution'.

Indeed, in Part 1 we saw Harel's criticism of Brooks' view [50] of the state of the software development industry due to his failure to adequately acknowledge developments in CASE (Computer-Aided Software Engineering) technology and visual formalisms [109]. In addition, in Part 2 we saw the originators of various structured methods recognize the fact that certain levels of automated support are vital to the successful industrialization of their respective methods [100].

9.1 What is CASE?

CASE is a generic term used to mean the application of computer-based tools (programs or suites of programs) to the software engineering process.

This loose definition serves to classify compilers, linkers, text editors, etc., as 'CASE tools', and indeed these should be classified in these terms. More usually, however, the term 'CASE' is intended to refer to a CASE workbench or CASE environment – an integrated suite of programs intended to support a large portion (ideally all) of the system development life-cycle.

To date, most CASE workbenches have focused on information systems and on supporting diagrammatic notations from various structured methods (DFDs, ERDs, etc.) rather than any specific methodology. In their article *CASE: Reliability Engineering for Information Systems* (reprinted in this Part), Chikofsky and Rubenstein provide an excellent overview of the motivation for the use of CASE workbenches in the development of reliable information systems, and of the advantages of such an approach [57].

That is not to say that the application of CASE is limited to the domain of information systems. CASE workbenches have been applied successfully to components of real-time and safety-critical systems, and some provide support for State-Transition Diagrams (STDs), Decision Tables, Event-Life History Diagrams, and other notations employed by various 'real-time' structured methods such as SART [251], DARTS [96] and Mascot [222].

9.2 CASE Workbenches

As was previously pointed out, commercially available CASE workbenches generally do not support any particular methodology (although there are exceptions) but rather a range of differing notations that may be tailored to a particular (standard or 'home-grown') methodology. They differ greatly in the degree to which they support various methodologies and, as one might expect, in the levels of functionality that they provide. We can, however, determine a minimal set of features that we would expect all realistic CASE workbenches to support:

- a consistent interface to various individual tools comprising the workbench, with the ability to exchange data easily between them;
- support for entering various diagrammatic notations with a form of syntax-directed editor (that is, that the editor can aid the user by prohibiting the derivation of syntactically incorrect or meaningless diagrams);
- an integrity checker to check for the consistency and completeness of designs;
- the ability to exchange data with other tools (perhaps using CDID, the CASE Data Interchange Format);
- report generation facilities;
- text editing facilities.

9.3 Beyond CASE

We can see from the above that CASE workbenches concentrate primarily on the early stages of system development. A number of workbenches, however, incorporate tools to generate skeleton code from designs as an aid to prototyping. Indeed, some vendors claim code that is of a sufficiently high quality that it may be used in the final implementation. Generally such workbenches are referred to as *application generators*.

From our discussion of the system life-cycle in this and previous Parts, it should be clear that we ideally require a CASE workbench that will support all aspects of system development from requirements analysis through to post-implementation maintenance. A workbench supporting all aspects of the software engineering process is generally termed a *Software Engineering Environment*, or SEE. A useful SEE can reasonably be expected to support an implementation of some form of software metric, project costing, planning, scheduling and evaluation of progress.

Realistically, large-scale system development involves a large number of personnel, working on various aspects of the development. Different personnel may modify the same software components [264], or may be modifying components required by others, perhaps due to changing requirements, or as a result of errors highlighted during unit testing. Therefore, co-ordination between developers and control over modification to (possibly distributed) software components is required, and software configuration management is desirable in a system development support environment. Such support is provided by an *Integrated Project Support Environment* or IPSE.

9.4 The Future of CASE

We see tool support becoming as important in formal development as it has been in the successful application of more traditional structured methods. In *Seven More Myths of Formal Methods* (reprinted in Part 3), we described some widely available tools to support formal development, and emphasized our belief that such tools will become more integrated, providing greater support for project management and configuration management [39]. In essence, we foresee the advent of Integrated Formal Development Support Environments (IFDSEs), the formal development equivalent of IPSEs.

As we also saw in Part 8, method integration is also an area auguring great potential. As a first step towards IFDSEs, visual formalisms offer great promise; in his paper *On Visual Formalisms* (reprinted in this Part) David Harel realizes the great importance of visual representations and their relationship to underlying formalisms [108].

His paper describes his own works on Statecharts, supported by the STATEMATE tool [110, 111], whereby an intuitive graphical notation that may be used in the description of reactive systems is given a formal interpretation in terms of 'higraphs', and for which support environments covering most phases of the life-cycle already exist. The approach has proven to be very popular and very successful in practice, and indeed many see visual formalisms as providing the basis for the next generation of CASE tools.

CASE: Reliability Engineering for Information Systems (Chikofsky & Rubenstein)

TO BE ADDED: Chikofsky and Rubenstein: CASE: Reliability Engineering for Information Systems

TO BE ADDED: 10 pages

On Visual Formalisms (Harel)

TO BE ADDED: Harel: On Visual Formalisms

TO BE ADDED: 36 pages

Glossary

Accreditation: The formal approval of an individual organization's services (e.g., a degree course at a university) by a *professional body* provided that certain specific criteria are met. Cf. *certification*.

Animation: The direct execution of a *specification* for *validation* purposes. This may not be possible in all cases, or may be possible for only part of the specification. Cf. *rapid prototype*.

Availability: A measure of the delivery of proper *service* with respect to the alternation of proper (desirable) and improper (undesirable) service.

Assertion: A *predicate* that should be true for some part of a program *state*.

CASE: Computer-Aided Software Engineering. A programming support environment for a particular method of software production supported by an integrated set of tools.

Class: A form of abstract data type in *object-oriented* programming.

Certification: The formal Indorsement of an individual by a *professional body* provided that certain specific criteria concerning education, training, experience, etc. are met. This is likely to be of increasing importance for personnel working on *high-integrity systems*, especially when *software* is involved. Cf. *accreditation*. The term is also applied to the rigorous demonstration or official written guarantee of a system meeting its *requirements*.

Code: *Executable program software* (normally as opposed to data on which the program operates).

Concurrent system: A *system* in which several processes, normally with communication between the processes, are active simultaneously. Cf. *distributed system*.

Deduction: A system of reasoning in a *logic*, following inference steps to arrive at a desired logical conclusion.

Dependability: A property of a computing system which allows reliance to be justifiably placed on the *service* it delivers. The combined aspects of *safety*, *reliability* and *availability* must normally be considered.

Design: The process of moving from a specification to an executable program, either *formally* or *informally*.

Development: The part of the *life-cycle* where the system is actually produced, before it is delivered to the customer. After this time, maintenance of the system is normally undertaken, and this is often far more costly than the originally development.

Distributed system: A system that is implemented on a number of physically separate computer systems, with communication, normally via a network. Cf. *concurrent system*.

Embedded system: A system in which the controlling computer forms an integral part of the system as a whole.

Emulation: A completely realistic imitation of a system by another different system that is indistinguishable from the original via some interface for all practical purposes. Cf. *simulation*.

Error: A mistake in the specification, design or operation of a system which may cause a *fault* to occur.

Executable specification: Thought by some to be an oxymoron, this is a high-level *formal specification* that can be *animated*.

Execution: The running of a *program* to perform some *operation*.

Failure: A condition or event in which a system is unable to perform one or more of its required functions due to a *fault* or *error*.

Fault: An undesirable system state than may result in a *failure*.

Fault avoidance: Prevention by construction of fault occurrence or introduction. For example, *formal methods* help in fault avoidance.

Fault tolerance: The provision by redundancy of a *service* complying with the specification in spite of faults. *N-version programming* is an example of fault tolerance.

Fault removal: The reduction by verification or testing of the presence of pre-existing faults.

Fault forecasting: The estimation by evaluation of the presence, creation and consequences of faults.

Formal methods: Techniques, notations and tools with a mathematical basis, used for *specification* and reasoning in *software* or *hardware* system *development*. The Z notation for *formal specification*) and the B-Method (for formal development) are leading examples of formal methods.

Formal notation: A language with a mathematical semantics, used for *formal specification*, reasoning and *proof*.

Formal specification: A *specification* written in a *formal notation*, potentially for use in *proof of correctness*.

Genericity: A program unit that can accept parameters that are types and subprograms as well as variables and values. Cf. *polymorphism*.

Hardware: The physical part of a computer system. Cf. *software*.

HCI: The means of communication between a human user or operator and a computer-based system.

High-integrity system: A system that must be trusted to work dependably and may result in unacceptable loss or harm otherwise. This includes *safety-critical systems* and other critical systems where, for example, *security* or financial considerations may be paramount.

Hybrid system: A system which includes a combination of both analogue and digital (e.g., a controlling computer) aspects.

Implementation: An efficiently executable version of a *specification* produced through a *design* process.

Informal: An indication of an absence of mathematical underpinning. E.g., cf. an informal notation like English or diagrams with a *formal notation* like Z.

Information hiding: The encapsulation of data within program components or *modules*, originally proposed by David Parnas, and now widely accepted as a good *object-oriented* programming development principle.

Integrity: A system's ability to avoid undesirable alteration due to the presence of errors.

Life-cycle: The complete lifetime of a system from its original conception to its eventual obsolescence. Typically phases of the life-cycle include *requirements*, *specification*, *design*, coding, testing, integration, commissioning, operation, maintenance, decommissioning, etc.

Logic: A scheme for reasoning, *proof*, inference, etc. Two common schemes are propositional logic and *predicate logic* which is propositional logic generalized with quantifiers. Other logics, such a modal logics, including *temporal logics* which handle time – e.g., Temporal Logic of Actions (TLA), Interval Temporal Logic (ITL) and more recently Duration Calculus (DC) – are also available. Schemes may use first-order logic or higher-order logic. In the former, functions are not allowed on predicates, simplifying matters somewhat, but in the latter they are, providing greater power. Logics include a calculus which allows reasoning in the logic.

Method integration: The combination of two or more techniques or notations to improve the development process by benefits from the strengths of each. Typically the approaches may be a combination of *formal* and informal methods.

Methodology: The study of methods. Also sometimes used to mean a set of related methods.

Model: A representation of a system – for example, an abstract *state* of the system and a set of *operations* on that state. The model may not cover all the features of the actual system being modeled.

Module: A structuring technique for programs, etc., allowing the breakdown of a system into smaller parts with well-defined interfaces.

N-version programming: The *implementation* of several programs from a common *specification* with the aim of reducing *faults*. Any variation in the operation of programs points to *errors* that can than be corrected. Also known as *diverse programming*.

Object-oriented: An approach where all component parts (e.g., processes, files, operations, etc.) are considered as objects. Messages may be passed between objects. See also *classes* and *information hiding*.

Operation: The performance of some desired action. This may involve the change of state of a system, together with inputs to the operation and outputs resulting from the operation. To specify such an operation, the *before state* (and inputs) and the *after state* (and outputs) must be related with constraining predicates.

Polymorphism: A high-level programming language feature allowing arguments to procedures and functions to act on a whole *class* of data types rather that just a single type. Cf. *genericity*.

Postcondition: An *assertion* (e.g., a *predicate*) describing the *state* after an *operation*, normally in terms of the state before the operation. Cf. *precondition*.

Precondition: An *assertion* (e.g., a *predicate*) which must hold on the *state* before an *operation* for it to be successful. Cf. *postcondition*.

Predicate: A constraint between a number of variables which produces a truth value (e.g., *true* or *false*). Predicate *logic* extends the simpler propositional logic with quantifiers allowing statements over potentially infinite numbers of objects (e.g., all, some).

Professional body: A (normally national or international) organization that members of a particular profession (e.g., engineers) may join. *Accreditation*, *certification* and *standards* are activities in which such organizations are involved. Examples include the Association of Computing Machinery (ACM) and the Institute of Electrical and Electronics Engineers (IEEE), based in the USA but with international membership, and the British Computer Society (BCS) and the Institution of Electrical Engineers (IEE), based in the UK.

Program: A particular piece of *software* (either a programming language or its matching executable machine *code*) to perform one or more *operations*.

Proof: A series of mathematical steps forming an argument of the correctness of a mathematical statement or theorem using some *logic*. For example, the *validation* of a desirable property for a formal specification could undertaken by proving it correct. Proof may also be used to perform a formal *verification* or 'proof of correctness' that an implementation meets a specification. A less formal style of reasoning is *rigorous argument*, where a proof outline is sketched informally, which may be done if the effort of undertaking a fully formal proof is not considered cost-effective.

Provably correct systems: A system that has been formally specified and implemented may be proven correct by demonstrating a mathematical relationship between the specification and the implementation in which the implementation

is 'better' in some sense (e.g., more deterministic, terminates more often, etc.) with respect to some set of *refinement* laws.

Rapid prototype: A quickly produced and inefficient implementation of a *specification* that can be executed for *validation* purposes. Cf. *animation*.

Real-time: A system where the timing aspects are important (e.g., the timing of external events is of comparable time to that of the computation undertaken by the controlling computer) is known as a real-time system.

Refinement: The stepwise transformation of a specification towards an implementation (e.g., as a program). Cf. *abstraction*, where unnecessary implementation detail is ignored in a specification.

Reliability: A measure of the continuous delivery of proper *service* (where service is delivered according to specified conditions) or equivalently of the time to failure.

Requirements: A statement of the desired properties for an overall system. This may be formal or informal, but should normally be as concise and easily understandable as possible.

Rigorous argument: An informal line of reasoning that could be formalized as a *proof* (given time). This level of checking may be sufficient and much more cost-effective than a fully formal proof for many systems.

Risk: An event or action with an associated loss where uncertainty or chance together with some choice are involved.

Safety: A measure of the continuous delivery of a *service* free from occurrences of catastrophic failures.

Safety-critical system: A system where failure may result in injury or loss of life. Cf. *high-integrity system*.

Safety-related: A system that is not necessarily safety-critical, but nevertheless where safety is involved, is sometimes called a *safety-related system*.

Security: Control of access to or updating of data so that only those with the correct authority may perform permitted operations. Normally the data represents sensitive information.

Service: The provision of one or more *operations* for use by humans or other computer systems.

Simulation: The imitation of (part of) a system. A simulation may not be perfect; for example, it may not run in *real-time*. Cf. *emulation*.

Software: The programs executed by a computer system. Cf. *hardware*.

Specification: A description of *what* a system is intended to do, as opposed to *how* it does it. A specification may be *formal* (mathematical) or *informal* (natural language, diagrams, etc.). Cf. an *implementation* of a specification, such as a program, that actually performs and executes the actions required by a specification.

Standard: An agreed document or set of documents produced by an official body designed to be adhered to by a number of users or developers (or an associated product) with the aim of overall improvement and compatibility. Standards bodies include the International Organization for Standardization (ISO), based in Switzerland but with international authority, and the American National Standards Institute (ANSI), based in the USA.

State: A representation of the possible values that a system may have. In an abstract specification, this may be modeled as a number of sets. By contrast, in a concrete program implementation, the state typically consists of a number of data structures, such as arrays, files, etc. When modeling sequential systems, each operation may include a *before state* and an *after state* which are related by some constraining predicates. The system will also have an *initial state*, normally with some additional constraints, from which the system starts at initialization.

Structured notation: A notation to aid in system analysis. A structured approach aims to limit the number of constructs available to those that allow easy and hence convincing reasoning.

System: A particular entity or collection of related components under consideration.

Temporal logic: A form of modal *logic* that includes operators specifically to deal with timing aspects (e.g., always, sometimes, etc.).

Validation: The checking of a system (e.g., its specification) to ensure it meets its (normally informal) requirements. This helps to ensure that the system does what is expected by the customer, but may be rather subjective. *Animation* or *rapid prototyping* may help in this process (e.g., as a demonstration to the customer). *Proof* of expected properties of a *formal specification* is another worthwhile approach. Cf. *verification*.

Verification: The checking of an implementation to ensure it meets its specification. This may be done formally (e.g., by *proof*) or information (e.g., by testing). This helps to ensure that the system does what has been specified in an objective manner, but does not help ensure that the original specification is correct. Cf. *validation*.

Bibliography

1. M. Abadi & L. Lamport (1993) Composing Specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(1):73–132.
2. J-R. Abrial (1996) *The B Book: Assigning Programs to Meanings*. Cambridge University Press.
3. J-R. Abrial, E. Börger and H. Langmaack (1996) *Formal Methods for Industrial Applications*. Springer-Verlag, Lecture Notes in Computer Science, volume 1165.
4. G.R. Andrews & F.B. Scheider (1983) Concepts and Notations for Concurrent Programming. *ACM Computing Surveys*, 15(1), March 1983. Reprinted in [267], Chapter 2, pp 29–84.
5. S. Aujla, A. Bryant & L. Semmens (1993) A Rigorous Review Technique: Using Formal Notations within Conventional Development Methods. In *Proc. 1993 Software Engineering Standards Symposium (SESS'93)*, IEEE Computer Society Press, pp 247–255.
6. R. Ayres (1999) *The Essence of Professional Issues in Computing*. Prentice Hall, Essence of Computing series, Hemel Hempstead.
7. L. Baresi & M. Pezzè (1998) Towards Formalizing Structured Analysis. *ACM Transactions on Software Engineering and Methology (TOSEM)*, 7(1):80–107, January 1998.
8. J.G.P. Barnes (1997) High Integrity Ada – The SPARK approach. Addison-Wesley.
9. L. Barroca & J.A. McDermid (1992) Formal Methods: Use and Relevance for the Development of Safety Critical Systems. *The Computer Journal*, 35(6):579–599, December 1992.
10. BCS (1996) High Integrity Real-Time Software. Report of the Foresight Defence and Aerospace Panel, British Computer Society, UK.
11. BCS (1998) Licensed to Work. *The Computer Bulletin*, p 18, November 1998.
12. R. Bell & D. Reinert (1993) Risk and System Integrity Concepts for Safety-related Control Systems. *Microprocessors and Microsystems*, 17(1):3–15, January/February 1993.
13. C. Bernardeschi, A. Fantechi, S. Gnesi, S. Larosa, G. Mongardi & D. Romano (1998) A Formal Verification Environment for Railway Signaling System Design. *Formal Methods in System Design*, 12(2):139–161, March 1998.
14. T. Berners-Lee (1996) World Wide Web Past Present and Future. *IEEE Computer*, 29(10):69–77, October 1996.
15. D. Bert, editor (1998) *B'98: Recent Advances in the Development and Use of the B Method*. Springer-Verlag, Lecture Notes in Computer Science, volume 1393.
16. R. Bharadwaj & C. Heitmeyer (1997) *Model Checking Complete Requirements Specifications Using Abstraction*. Technical Report NRL/MR/5540--97-7999, Center for High Assurance Systems, Information Technology Division, Naval Research Laboratory, Washington, DC 20375-5320, USA, November 1997.
17. J.C. Bicarregui (1998) *Proof in VDM: Case Studies*. Springer-Verlag, Formal Approaches to Computing and Information Technology (FACIT) series, London.

18. B.W. Boehm (1975) The High Cost of Software. In E. Horowitz, editor, *Practical Strategies for Developing Large Software Systems*, Addison-Wesley, London.
19. B.W. Boehm (1981) *Software Engineering Economics*, Prentice Hall, Hemel Hempstead and Englewood Cliffs.
20. B.W. Boehm (1988) A Sprial Model of Software Development and Maintenance. *IEEE Computer*, 21(5):61–72, May 1988.
21. E.A. Boiten, H.A. Partsch, D. Tuijnman & N. Völker (1992) How to Produce Correct Software – An Introduction to Formal Specification and Program Development by Transformations. *The Computer Journal*, 35(6):547–554, December 1992.
22. * G. Booch (1986) Object-Oriented Development. *IEEE Transactions on Software Engineering*, 12(2):211–221, February 1986.
(See page 237 in this collection.)
23. G. Booch (1994) *Object-Oriented Analysis and Design with Applications*, 2nd edition. Addison Wesley Longman, Object Technology Series.
24. G. Booch (1996) *Object Solutions: Managing the Object-Oriented Project*. Addison Wesley Longman, Object Technology Series.
25. G. Booch, J. Rumbaugh & I. Jacobson (1999) *The Unified Modeling Language User Guide*. Addison Wesley Longman, Object Technology Series.
26. J.P. Bowen (1993) Formal Methods in Safety-Critical Standards. In *Proc. 1993 Software Engineering Standards Symposium (SESS'93)*, IEEE Computer Society, pp 168–177.
27. J.P. Bowen, editor (1994) *Towards Verified Systems*, Elsevier, Real-Time Safety Critical Systems, Volume 2, 1994.
28. J.P. Bowen (1996) *Formal Specification and Documentation using Z: A Case Study Approach*. International Thomson Computer Press, International Thomson Publishing, London.
29. J.P. Bowen (1997) The Ethics of Safety-Critical Systems. In D. Gritzalis & T. Anderson,editors, *Reliability, Quality and Safety of Software-Intensive Systems: Application experiences, trends and perspectives, and key theoretical issues*. European Commission ESPRIT/ESSI Programme ENCRESS (21542) Project, pp 253–267. Revised version to appear in *Communications of the ACM*.
30. J.P. Bowen, P.T. Breuer & K.C. Lano (1993) Formal Specifications in Software Maintenance: From code to Z^{++} and back again. *Information and Software Technology*, 35(11/12):679–690, November/December 1993.
31. J.P. Bowen, R.W. Butler, D.L. Dill, R.L. Glass, D. Gries, J.A. Hall, M.G. Hinchey, C.M. Holloway, D. Jackson, C.B. Jones, M.J. Lutz, D.L. Parnas, J. Rushby, H. Saiedian, J. Wing & P. Zave (1996) An Invitation to Formal Methods. *IEEE Computer*, 29(4):16–30, April 1996.
32. J.P. Bowen, A. Fett & M.G. Hinchey, editors (1998) *ZUM'98: The Z Formal Specification Notation*. Springer-Verlag, Lecture Notes in Computer Science, volume 1493.
33. J.P. Bowen, M. Fränzle, E-R. Olderog & A.P. Ravn (1993) Developing Correct Systems. In *Proc. Fifth Euromicro Workshop on Real-Time Systems*, Oulu, Finland, 22–24 June 1993. IEEE Computer Society Press, pp 176–187.
34. J.P. Bowen & T. Gleeson (1990) Distributed Operating Systems. In [267], Chapter 1, pp 3–28.
35. J.P. Bowen & J.A. Hall, editors (1994) *Z User Meeting, Cambridge 1994*, Springer-Verlag, Workshops in Computing.
36. J.P. Bowen, He Jifeng, R.W.S. Hale & J.M.J. Herbert (1995) Towards Verified Systems: The SAFEMOS Project. In C. Mitchell and V. Stavridou, editors, *Mathematics of Dependable Systems*, Clarendon Press, Oxford, pp 23–48.
37. J.P. Bowen & M.G. Hinchey (1994) Formal Methods and Safety-Critical Standards. *IEEE Computer*, 27(8):68–71, August 1994.

38. * J.P. Bowen & M.G. Hinchey (1995) Ten Commandments of Formal Methods. *IEEE Computer*, 28(4):56–63, April 1995.
(See page 217 in this collection.)
39. * J.P. Bowen & M.G. Hinchey (1995) Seven More Myths of Formal Methods. *IEEE Software*, 12(4):34–41, July 1995.
(See page 153 in this collection.)
40. J.P. Bowen & M.G. Hinchey, editors (1995) *ZUM'95: The Z Formal Specification Notation*. Springer-Verlag, Lecture Notes in Computer Science, volume 967.
41. J.P. Bowen & M.G. Hinchey (1997) Formal Models and the Specification Process. In [246], Chapter 107, pp 2302–2322.
42. J.P. Bowen, M.G. Hinchey & D. Till, editors (1997) *ZUM'97: The Z Formal Specification Notation*. Springer-Verlag, Lecture Notes in Computer Science, volume 1212.
43. J.P. Bowen, C.A.R. Hoare, H. Langmaack, E-R. Olderog and A.P. Ravn (1996) A ProCoS II Project Final Report: ESPRIT Basic Research project 7071. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 59:76–99, June 1996.
44. J.P. Bowen & J.E. Nicholls, editors (1993) *Z User Workshop, London 1992*, Springer-Verlag, Workshops in Computing.
45. J.P. Bowen & V. Stavridou (1993) The Industrial Take-up of Formal Methods in Safety-Critical and Other Areas: A Perspective. In [259], pp 183–195.
46. * J.P. Bowen & V. Stavridou (1993) Safety-Critical Systems, Formal Methods and Standards. *Software Engineering Journal*, 8(4):189–209, July 1993.
(See page 485 in this collection.)
47. J.P. Bowen & V. Stavridou (1994) Formal Methods: Epideictic or Apodeictic? *Software Engineering Journal*, 9(1):2, January 1994 (Personal View).
48. R.S. Boyer & J.S. Moore (1988) *A Computational Logic Handbook*. Academic Press, London.
49. P.T. Breuer & J.P. Bowen (1994) Towards Correct Executable Semantics for Z. In [35], pp 185–209.
50. * F.P. Brooks, Jr. (1987) No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, April 1987. Originally published in H.-J. Kugler, editor (1986) *Information Processing '86*, Proc. IFIP Congress, Dublin, Ireland, Elsevier Science Publishers B.V. (North-Holland).
(See page 11 in this collection.)
51. F.P. Brooks, Jr. (1995) *The Mythical Man-Month: Essays on Software Engineering*, 20th anniversary edition. Addison-Wesley, Reading, Massachusetts.
52. S.D. Brookes, C.A.R. Hoare & A.W. Roscoe (1984) A Theory of Communicating Sequential Processes. *Journal of the Association for Computing Machinery*, 31:560–599, 1984.
53. A. Bryant & L. Semmens, editors (1996) *Methods Integration*, Proc. Methods Integration Workshop, Leeds, 25–26 March 1996. Springer-Verlag, Electronic Workshops in Computing.
URL: *http://www.springer.co.uk/ewic/workshops/MI96/*
54. R.W. Butler & G.B. Finelli (1993) The Infeasibility of Experimental Quantification of Life Critical Software Reliability. *IEEE Transactions on Software Engineering*, 19(1):3–12, January 1993.
55. * J.R. Cameron (1986) An Overview of JSD. *IEEE Transactions on Software Engineering*, 12(2):222–240, February 1986.
(See page 77 in this collection.)
56. CCTA (1990) *SSADM Version 4 Reference Manual*, NCC Blackwell Ltd.
57. * E.J. Chikofsky & B.L. Rubenstein (1988) CASE: Reliability Engineering for Information Systems. *IEEE Software*, 5(2):11–16, March 1988.
(See page 613 in this collection.)

58. E. Clarke, J. Wing *et al.* (1996) Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.
59. R. Cleaveland, S.A. Smolka *et al.* (1996) Strategic Directions in Concurrency Research. *ACM Computing Surveys*, 28(4):607–625, December 1996.
60. P. Coad & E. Yourdon (1991) *Object Oriented Design*, Yourdon Press/Prentice Hall, New York.
61. D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes amd P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice Hall, Object-Oriented Series, Hemel Hempstead.
62. J. Cooke (1998) *Constructing Correct Software: The basics*. Springer-Verlag, Formal Approaches to Computing and Information Technology (FACIT) series, London.
63. D. Craigen, S. Gerhart & T. Ralston (1993) *An International Survey of Industrial Applications of Formal Methods*. NIST GCR 93/626, Atomic Energy Control Board of Canada, U.S. National Institute of Standards and Technology, and U.S. Naval Research Laboratories, 1993.
64. J. Crow & B. Di Vito (1998) Formalizing Space Shuttle Software Requirements: Four Case Studies. *ACM Transactions on Software Engineering and Methology (TOSEM)*, 7(3):296–332, July 1998.
65. C.N. Dean & M.G. Hinchey, editors (1996) *Teaching and Learning Formal Methods*. Academic Press, International Series in Formal Methods, London.
66. B. Dehbonei & F. Mejia (1994) Formal Methods in the Railway Signalling Industry. In M. Naftalin, T. Denvir & M. Bertran, editors, *FME'94: Industrial Benefit of Formal Methods*, Springer-Verlag, Lecture Notes in Computer Science, volume 873, pp 26–34.
67. T. DeMarco (1978) *Structured Analysis and System Specification*. Yourdon Press, New York.
68. J. Dick & E. Woods (1997) Lessons Learned from Rigorous System Software Development. *Information and Software Technology*, 39(8):551–560, August 1997.
69. E.W. Dijkstra (1968) Goto Statement Considered Harmful, *Communications of the ACM*, 11(3):147–148 (Letter to the Editor).
70. E.W. Dijkstra (1981) Why Correctness must be a Mathematical Concern. In R.S. Boyer & J.S. Moore, editors, *The Correctness Problem in Computer Science*, Academic Press, London, pp 1–6.
71. A. Dix, J. Finlay, G. Abowd & R. Beale (1998) *Human-Computer Interaction*, 2nd edition. Prentice Hall, Hemel Hempstead & Englewood Cliffs.
URL: *http://www.hiraeth.com/books/hci/*
72. C. Draper (1993) Practical Experiences of Z and SSADM. In [44], pp 240–254.
73. M. Dyer (1992) *The Cleanroom Approach to Quality Software Development*, John Wiley & Sons, Series in Software Engineering Practice.
74. S. Easterbrook & J. Callahan (1998) Formal Methods for Verification and Validation of Partial Specifications: A case study. *Journal of Systems and Software*, 40(3):199–210, March 1998.
75. The Economist (1997) Fasten your Safety Belts. *The Economist*, pp 69–71, 11–17 January 1997.
76. H.-E. Eriksson & M. Penker (1998) *UML Toolkit*. Wiley Computer Publishing, New York.
77. S.R. Faulk (1995) *Software Requirements: A Tutorial*. Technical Report NRL/MR/-5546--95-7775, Center for High Assurance Systems, Information Technology Division, Naval Research Laboratory, Washington, DC 20375-5320, USA, November 1995.
78. * R.G. Fichman & C.F. Kemerer (1992) Object-Oriented and Conventional Analysis and Design Methodologies: Comparison and Critique. *IEEE Computer*, 25(10):22–39, October 1992.
(See page 261 in this collection.)

79. J.S. Fitzgerald, P.G. Larsen, T. Brookes & M. Green (1995) Developing a Security-Critical System Using Formal and Conventional Methods. In [125], Chapter 14, pp 333–356.
80. R. France, A. Evans & K.C. Lano (1997) The UML as a Formal Modeling Notation. In H. Kilov, B. Rumpe & I. Simmonds, editors, *Proc. OOPSLA'97 Workshop on Object-Oriented Behavioral Semantics (with an Emphasis on Semantics of Large OO Business Specifications)*, TUM-I9737, Institut für Informatik, Technische Universität München, Munich, Germany, pp 75–81, September 1997.
81. M.D. Fraser, K. Kumar & V.K. Vaishnavi (1994) Strategies for Incorporating Formal Specifications in Software Development. *Communications of the ACM*, 37(10):74–86, October 1994.
82. M.D. Fraser & V.K. Vaishnavi (1997) A Formal Specifications Maturity Model. *Communications of the ACM*, 40(12):95–103, December 1997.
83. V. Friesen, Nordwig & M. Weber (1998) Object-Oriented Specification of Hybrid Systems Using UML^h and ZimOO. In [32], pp 328–346.
84. * N.E. Fuchs (1992) Specifications are (Preferably) Executable. *Software Engineering Journal*, 7(5):323–334, September 1992.
(See page 583 in this collection.)
85. N.E. Fuchs, editor (1997) Logic Program Synthesis and Transformation, Springer-Verlag, Lecture Notes in Computer Science, volume 1463.
86. D. Garlan (1995) Making Formal Methods Effective for Professional Software Engineers. *Information and Software Technology*, 37(5/6):261–268, May/June 1995.
87. D. Garlan (1996) Effective Formal Methods Education for Professional Software Engineers. In [65], Chapter 2, pp 11–29.
88. D. Garlan, R. Allen & J. Ockerbloom (1995) Architectural Mismatch: Why Reuse is So Hard. *IEEE Software*, 12(6):17–26, November 1995.
89. C. Gaskell & A.A. Takang (1996) Professional Issues in Software Engineering: The Perspective of UK Academics. *IEE Computing & Control Engineering Journal*, 7(6)287–293.
90. * S. Gerhart, D. Craigen & T. Ralston (1994) Experience with Formal Methods in Critical Systems, *IEEE Software*, 11(1):21–28, January 1994.
(See page 413 in this collection.)
91. * S. Gerhart, D. Craigen & T. Ralston (1994) Regulatory Case Studies, *IEEE Software*, 11(1):30–39, January 1994.
(See page 429 in this collection.)
92. W.W. Gibbs (1994) Software's Chronic Crisis. *Scientific American*, 271(3):86–95, September 1994.
93. G.R. Gladden (1982) Stop the Life Cycle – I Want to Get Off. *ACM Software Engineering Notes*, 7(2):35–39.
94. J.A. Goguen & T. Winkler (1988) *Introducing OBJ3*, Technical Report SRI-CSL-88-9, SRI International, Melo Park, CA, August 1988.
95. J.A. Goguen (1981) More Thoughts on Specification and Verification. *ACM SIGSOFT*, 6(3):38–41.
96. H. Gomaa (1993) *Software Design Methods for Concurrent and Real-Time Systems*, Addison-Wesley, Reading, Massachusetts.
97. D. Gotterbarn, K. Miller & S. Rogerson (1997) Software Engineering Code of Ethics. *Communications of the ACM*, 40(11):110–118, November 1997.
98. J.N. Gray (1986) An Approach to Decentralized Computer Systems. *IEEE Transactions on Software Engineering*, SE-12(6):684–692, June 1986.
99. D. Gries & F.B. Schneider (1993) *A Logical Approach to Discrete Math*. Springer-Verlag, New York.
100. K. Grimm (1998) Industrial Requirements for the Efficient Development of Reliable Embedded Systems. In [32], pp 1–4.

101. D. Gritzalis, editor (1997) *Reliability, Quality and Safety of Software-Intensive Systems*. Chapman & Hall, London, on behalf of the International Federation for Information Processing (IFIP).
102. R.L. Grossman, A. Nerode, A.P. Ravn & H. Rischel, editors (1993) *Hybrid Systems*. Springer-Verlag, Lecture Notes in Computer Science, volume 736.
103. J.V. Guttag & J.J. Horning (1993) *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, Texts and Monographs in Computer Science, New York.
104. W.A. Halang & A.D. Stoyenko (1991) *Constructing Predictable Real Time Systems*. Kluwer Academic Publishers, Boston.
105. * J.A. Hall (1990) Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, September 1990.
(See page 135 in this collection.)
106. J.A. Hall (1996) Using Formal Methods to Develop an ATC Information System. *IEEE Software*, 13(2):66–76, March 1996.
107. D. Harel (1987) Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
108. * D. Harel (1988) On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.
(See page 623 in this collection.)
109. * D. Harel (1992) Biting the Silver Bullet: Toward a Brighter Future for System Development. *IEEE Computer*, 25(1):8–20, January 1992.
(See page 29 in this collection.)
110. D. Harel & A. Haamad (1996) The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methology (TOSEM)*, 5(4):293–333, October 1996.
111. D. Harel & M. Politi (1998) *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw-Hill, London & New York.
112. M.D. Harrison (1992) Engineering Human Error Tolerant Software. In [194], pp 191–204.
113. L. Hatton (1995) *Safer C: Developing for High-Integrity and Safety-Critical Systems*. McGraw-Hill International Series in Software Engineering, London & New York.
114. L. Hatton (1997) Software Failures – Follies and Fallacies. *IEE Review*, 43(2):49–52.
115. H. Haughton & K.C. Lano (1996) *Specification in B: An Introduction Using the B Toolkit*. World Scientific Publishing Company, Imperial College Press, London.
116. The Hazards Forum (1995) *Safety-Related Systems: Guidance for Engineers*. The Hazards Forum, 1 Great George Street, London SW1P 4AA, UK.
URL: *http://www.iee.org.uk/PAB/safe_rel.htm*
117. * I.J. Hayes & C.B. Jones (1989) Specifications are not (Necessarily) Executable. *Software Engineering Journal*, 4(6):330–338, November 1989.
(See page 563 in this collection.)
118. He Jifeng (1995) *Provably Correct Systems: Modelling of Communication Languages and Design of Optimized Compilers*. McGraw-Hill, International Series on Software Engineering, London & New York.
119. C. Heitmeyer (1997) Formal Methods: A Panacea or Academic Poppycock? In [42], pp 3–9.
120. C. Heitmeyer & D. Mandrioli, editors (1996) *Formal Methods for Real-Time Computing*. John Wiley & Sons, Trends in Software.
121. S. Hekmatpour & D.C. Ince (1989) *System Prototyping, Formal Methods and VDM*, Addison-Wesley, Reading, Massachusetts.
122. A. Heydon, M. Maimone, J.D. Tygar, J. Wing & A.M. Zaremski (1990) Miró: Visual Specification of Security. *IEEE Transactions on Software Engineering*, 16(10):1185–1197, October 1990.

123. J.V. Hill (1991) Software Development Methods in Practice, In A. Church, editor, *Microprocessor Based Protection Systems*, Routledge.
124. M.G. Hinchey (1993) The Design of Real-Time Applications. In *Proc. RTAW'93, 1st IEEE Workshop on Real-Time Applications*, New York City, 13–14 May 1993, IEEE Computer Society Press, pp 178–182.
125. M.G. Hinchey & J.P. Bowen, editors (1995) *Applications of Formal Methods*. Prentice Hall International Series in Computer Science, Hemel Hempstead & Englewood Cliffs.
126. M.G. Hinchey & S.A. Jarvis (1994) Simulating Concurrent Systems with Formal Methods. In *Proc. WMC'94, Winter Multiconference of the Society for Computer Simulation*, Tempe, Arizona, January 1994.
127. M.G. Hinchey & S.A. Jarvis (1995) *Concurrent Systems: Formal Development in CSP*. McGraw-Hill International Series in Software Engineering, London & New York.
128. * C.A.R. Hoare (1978) Communicating Sequential Processes, *Communications of the ACM*, 21(8):666–677, August 1978.
(See page 303 in this collection.)
129. C.A.R. Hoare (1985) *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, Hemel Hempstead & Englewood Cliffs.
130. * C.A.R. Hoare (1987) An Overview of Some Formal Methods for Program Design. *IEEE Computer*, 20(9):85–91, September 1987.
(See page 201 in this collection.)
131. C.A.R. Hoare (1995) Algebra and Models. In [118], Chapter 1, pp 1–14.
132. C.A.R. Hoare (1996) How did Software get so Reliable Without Proof? In M-C. Gaudel & J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, Springer-Verlag, Lecture Notes in Computer Science, volume 1051, pp 1–17.
133. C.A.R. Hoare (1996) The Logic of Engineering Design. *Microprocessing and Microprogramming*, 41(8–9):525–539.
134. C.A.R. Hoare & He Jifeng (1998) *Unifying Theories of Programming*. Prentice Hall International Series in Computer Science, Hemel Hempstead and Englewood Cliffs.
135. G.J. Holzmann (1996) Formal Methods for Early Fault Detection. In B. Jonsson & J. Parrow, editors, *Formal Techniques in Real-Time Fault-Tolerant Systems*, Springer-Verlag, Lecture Notes in Computer Science, volume 1135, pp 40–54.
136. G.J. Holzmann (1997) The Model Checker Spin. *IEEE Transactions on Software Engineering*. 23(5):279–295, May 1997.
137. M.E.C. Hull, A. Zarea-Aliabadi & D.A. Guthrie (1989) Object-Oriented Design, Jackson System Development (JSD) Specifications and Concurrency, *Software Engineering Journal*, 4(3), March 1989.
138. D.C. Ince (1992) Arrays and Pointers Considered Harmful. *ACM SIGPLAN Notices*, 27(1):99–104, January 1992.
139. Inmos Ltd. (1988) *Occam2 Reference Manual*, Prentice Hall International Series in Computer Science, Hemel Hempstead and Englewood Cliffs.
140. International Standards Organization (1989) *Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour*, Document Number ISO 8807:1989. URL: *http://www.iso.ch/cate/d16258.html*
141. International Standards Organization (1996) *Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language*, Document Number ISO/IEC 13817-1:1996. URL: *http://www.iso.ch/cate/d22988.html*
142. M.A. Jackson (1975) *Principles of Program Design*, Academic Press, London.
143. M.A. Jackson (1983) *System Development*. Prentice Hall International Series in Computer Science, Hemel Hempstead and Englewood Cliffs.
144. J. Jacky (1995) Specifying a Safety-Critical Control System in Z. *IEEE Transactions on Software Engineering*, 21(2):99–106, February 1995.

145. J. Jacky (1997) *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press.
146. C.B. Jones (1991) *Systematic Software Development Using VDM*, 2nd edition. Prentice Hall International Series in Computer Science, Hemel Hempstead and Englewood Cliffs.
147. M. Joseph, editor (1996) *Real-Time Systems: Specification, Verification and Analysis*. Prentice Hall International Series in Computer Science, Hemel Hempstead and Englewood Cliffs.
148. R.A. Kemmerer (1990) Integrating Formal Methods into the Development Process. *IEEE Software*, 7(5):37–50, September 1990.
149. J.C. Knight & D.M. Kienzle (1993) Preliminary Experience Using Z to Specify a Safety-Critical System. In [44], pp 109–118.
150. J. Knight & B. Littlewood (1994) Critical Task of Writing Dependable Software. *IEEE Software*, 11(1):16–20, January 1994.
151. L. Lamport (1978) Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.
152. * L. Lamport (1989) A Simple Approach to Specifying Concurrent Systems. *Communications of the ACM*, 32(1):32–45, January 1989.
(See page 331 in this collection.)
153. L. Lamport (1994) The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, May 1994.
154. L. Lamport (1994) LaTeX*: A Document Preparation System User's Guide and Reference Manual*, Addison-Wesley, Reading, Massachusetts.
155. H. Langmaack (1997) The ProCoS Approach to Correct Systems. *Real-Time Systems*, 13:253–275.
156. K.C. Lano (1996) *The B Language and Method: A Guide to Practical Formal Development*. Springer-Verlag, Formal Approaches to Computing and Information Technology (FACIT) series, London.
157. K.C. Lano (1996) *Formal Object-Oriented Development*. Springer-Verlag, Formal Approaches to Computing and Information Technology (FACIT) series, London.
158. K.C. Lano & J. Bicarregui (1998) Formalising the UML in Structured Temporal Theories. In H. Kilov & B. Rumpe, editors, *Proc. Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications)*, TUM-I9813, Institut für Informatik, Technische Universität München, Munich, Germany, pp 105–121, June 1998.
159. K.C. Lano & H. Haughton, editors (1994) *Object-Oriented Specification Case Studies*, Prentice Hall, Object-Oriented Series, Hemel Hempstead.
160. P.G. Larsen, J.S. Fitzgerald & T. Brookes (1996) Applying Formal Specification in Industry. *IEEE Software*, 13(3):48–56, May 1996.
161. H.C. Lauer & R.M. Needham (1979) On the Duality of Operating System Structures. *Operating Systems Review*, 13(2):3–19, April 1979.
162. D. Learmount (1992) Airline Safety Review: Human Factors. *Flight International*, 142(4238):30–33, 22–28 July 1992.
163. B. Le Charlier & P. Flener (1998) Specifications are Necessarily Informal: More myths of formal methods. *Journal of Systems and Software*, 40(3):275–296, March 1998.
164. G. LeLann (1981) Motivations, Objectives and Characterization of Distributed Systems. In *Distributed Systems – Architecture and Implementation*, Springer-Verlag, pp 1–9.
165. N.G. Leveson (1991) Software Safety in Embedded Computer Systems. *Communications of the ACM*, 34(2):34–46, February 1991.
166. N.G. Leveson (1993) *An Assessment of Space Shuttle Flight Software Development Processes*. National Academy Press, USA.

167. N.G. Leveson (1995) *Safeware: System Safety and Computers*, Addison-Wesley, Reading, Massachusetts.
168. * N.G. Leveson (1995) Medical Devices: The Therac-25 Story. In [167], Appendix A, pp 515–553.
(See page 447 in this collection.)
169. N.G. Leveson & C.T. Turner (1993) An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, July 1993.
170. N.G. Leveson & P.G. Neumann (1993) Introduction to Special Issue on Software for Critical Systems. *IEEE Transactions on Software Engineering*, 19(1):1–2, January 1993.
171. B.P. Lientz & E.B. Swanson (1980) *Software Maintenance Management*, Addison-Wesley, Reading, Massachusetts.
172. B. Littlewood & L. Strigini (1992) The Risks of Software. *Scientific American*, 267(5):38–43, November 1992.
173. Luqi & J.A. Goguen (1997) Formal Methods: Promises and Problems. *IEEE Software*, 14(1):73–85, January 1997.
174. J.L. Lyons (1996) *ARIANE 5: Flight 501 Failure*. Report by the Inquiry Board, European Space Agency, 19 July 1996.
URL: *http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html*
175. D.D. McCracken & M.A. Jackson (1982) Life Cycle Concept Considered Harmful. *ACM Software Engineering Notes*, 7(2):28–32.
176. J.A. McDermid, editor (1992) *Software Engineer's Reference Book*. Butterworth-Heinemann.
177. D. MacKenzie (1992) Computers, Formal Proof, and the Law Courts. *Notices of the American Mathematical Society*, 39(9):1066–1069, November 1992.
178. D. MacKenzie (1995) The Automation of Proof: A Historical and Sociological Exploration. *IEEE Annals of the History of Computing*, 17(3):7–29, Fall 1995.
179. K.L. McMillan (1993) *Symbolic Model Checking*. Kluwer Academic Press, Boston.
180. S. Magee & L.L. Tripp (1997) *Guide to Software Engineering Standards and Specifications*. Artech House, Boston.
181. J.J. Marciniak, editor (1994) *Encyclopedia of Software Engineering*. John Wiley & Sons, Wiley-Interscience Publication, New York (2 volumes).
182. B. Meyer (1985) On Formalism in Specification. *IEEE Software*, 2(1):5–13, January 1985.
183. H.D. Mills (1993) Zero-Defect Software – Cleanroom Engineering. *Advances in Computers*, 36:1–41.
184. H.D. Mills, M. Dyer & R.C. Linger (1987) Cleanroom Software Engineering. *IEEE Software*, 4(5):19–25, September 1997.
185. R. Milner (1989) *Communication and Concurrency*. Prentice Hall International Series in Computer Science, Hemel Hempstead and Englewood Cliffs.
186. Ministry of Defence (1991) *The Procurement of Safety Critical Software in Defence Equipment* (Part 1: Requirements, Part 2: Guidance). Interim Defence Standard 00-55, Issue 1, MoD, Directorate of Standardization, Kentigern House, 65 Brown Street, Glasgow G2 8EX, UK, 5 April 1991.
187. C. Morgan (1994) *Programming from Specifications*, 2nd edition. Prentice Hall International Series in Computer Science, Hemel Hempstead and Englewood Cliffs.
188. L. Moser, Y.S. Ramakrisha, G. Kutty, P.M. Melliar-Smith and L.K. Dillon (1997) A Graphical Environment for the Design of Concurrent Real-Time Systems. *ACM Transactions on Software Engineering and Methology (TOSEM)*, 6(1):31–79, January 1997.
189. S.J. Mullender, editor (1993) *Distributed Systems*. Addison-Wesley, Reading, Massachusetts.
190. R.M. Needham (1993) Cryptography and Secure Channels. In [189], pp 531–541.

191. P.G. Neumann (1989) Flaws in Specifications and What to do About Them. In *Proc. 5th International Workshop on Software Specification and Design, ACM SIGSOFT Engineering Notes*, 14(3):xi–xv, May 1989.
192. P.G. Neumann (1994) *Computer-Related Risks*. Addison-Wesley, Reading, Massachusetts.
193. J.E. Nicholls (1991) A Survey of Z Courses in the UK. In J.E. Nicholls, editor, *Z User Workshop, Oxford 1990*, Springer-Verlag, Workshops in Computing, pp 343–350.
194. J.E. Nicholls, editor (1992) *Z User Workshop, York 1991*. Springer-Verlag, Workshops in Computing.
195. N. Nissanke (1997) *Realtime Systems*. Prentice Hall International Series in Computer Science, Hemel Hempstead and Englewood Cliffs.
196. P. Oman, editor (1994) Computing Practices: Hardware/Software Codesign. *Computer* 27(1):42–55, January 1994.
197. G. O'Neill (1992) Automatic Translation of VDM Specifications into Standard ML Programs. *The Computer Journal*, 35(6):623–624, December 1992 (Short note).
198. T. O'Riordan, R. Kemp & M. Purdue (1988) *Sizewell B: An Anatomy of the Inquiry*. The Macmillan Press Ltd, Basingstoke.
199. * K. Orr, C. Gane, E. Yourdon, P.P. Chen & L.L. Constantine (1989) Methodology: The Experts Speak. *BYTE*, 14(4):221–233, April 1989.
(See page 57 in this collection.)
200. * J.S. Ostroff (1992) Formal Methods for the Specification and Design of Real-Time Safety-Critical Systems. *Journal of Systems and Software*, 18(1):33–60, April 1992.
(See page 367 in this collection.)
201. S. Owre, J.M. Rushby & N. Shankar (1992) PVS: A Prototype Verification System. In D. Kapur, editor, *Automated Deduction – CADE-11*, Springer-Verlag, Lecture Notes in Artificial Intelligence, volume 607, pp 748–752.
202. P. Palanque & F. Paternò, editors (1997) *Formal Methods in Human-Computer Interaction*. Springer-Verlag, Formal Approaches to Computing and Information Technology (FACIT) series, London.
203. D.L. Parnas (1972) On the Criteria to be Used in Decomposing Systems. *Communications of the ACM*, 15(5):1053–1058.
204. D.L. Parnas (1993) Predicate Logic for Software Engineering. *IEEE Transactions on Software Engineering*, 19(9):856–862, September 1993.
205. D.L. Parnas (1995) Teaching Programming as Engineering. In [40], pp 471–481.
206. D.L. Parnas (1995) Using Mathematical Models in the Inspection of Critical Software. In [125], Chapter 2, pp 17–31.
207. D.L. Parnas (1996a) Education for Computing Professionals. In [65], Chapter 3, pp 31–42.
208. D.L. Parnas (1996b) Teaching Programming as Engineering. In [65], Chapter 4, pp 43–55.
209. D.L. Parnas (1997a) Software Engineering: An Unconsummated Marriage. *Communications of the ACM*, 40(9):128, September 1997.
210. D.L. Parnas (1997b) Precise Description and Specification of Software. In V. Stavridou, editor, *Mathematics of Dependable Systems II*, Clarendon Press, Oxford, pp 1–14.
211. D.L. Parnas (1998a) "Formal Methods" Technology Transfer Will Fail. *Journal of Systems and Software*, 40(3):195–198, March 1998.
212. D.L. Parnas (1998b) Successful Software Engineering Research. *SIGSOFT Software Engineering Notes*, 23(3):64–68, May 1998.
213. D.L. Parnas, G.J.K. Asmis & J. Madey (1991) Assessment of Safety-Critical Software in Nuclear Power Plants. *Nuclear Safety*, 32(2):189–198.
214. D.A. Penney, R.C. Hold & M.W. Godfrey (1991) Formal Specification in Metamorphic Programming. In S. Prehn & H. Toetenel, editors, *VDM'91: Formal Software Development Methods*. Springer-Verlag, Lecture Notes in Computer Science, volume 551.

215. J.H. Poore & C.J. Trammell, editors (1996) *Cleanroom Software Engineering: A Reader*. Blackwell Publishers.
216. J.P. Potocki de Montalk (1993) Computer Software in Civil Aircraft. *Microprocessors and Microsystems*, 17(1):17–23, January/February 1993.
217. R.S. Pressman (1997) *Software Engineering: A Practitioner's Approach*, 4th edition. McGraw-Hill, New York.
218. RAISE Language Group (1992) *The RAISE Specification Language*. Prentice Hall, BCS Practitioner Series, Hemel Hempstead.
219. F. Redmill & T. Anderson, editors (1993) *Safety-critical Systems: Current Issues, Techniques and Standards*, Chapman & Hall, London.
220. A.W. Roscoe (1998) *The Theory and Practice of Concurrency*. Prentice Hall International Series in Computer Science, Hemel Hempstead and Englewood Cliffs.
221. Royal Society Study Group (1992) *Risk: Analysis, Perception and Management*, The Royal Society, 6 Carlton House Terrace, London SW1Y 5AG, UK.
222. Royal Signals and Radars Establishment (1991) *The Official Handbook of MASCOT, Version 3.1*, Computing Division, RSRE, Malvern, UK.
223. W.W. Royce (1970, 1987) Managing the Development of Large Software Systems. In *Proc. WESTCON'70*, August 1970. Reprinted in *Proc. 9th International Conference on Software Engineering*, IEEE Press, 1987.
224. W.W. Royce (1998) *Software Project Management: A Unified Framework*. Addison Wesley Longman, Object Technology Series.
225. A.R. Ruddle (1993) Formal Methods in the Specification of Real-Time, Safety-Critical Control Systems. In [44], pp 131–146.
226. J. Rumbaugh, I. Jacobson & G. Booch (1999) *The Unified Modeling Language Reference Manual*. Addison Wesley Longman, Object Technology Series.
227. J. Rushby (1995) *Formal Methods and their Role in the Certification of Critical Systems*. Technical Report SRI-CSL-95-01, SRI International, Menlo Park, California, USA, March 1995.
228. H. Saiedian & M.G. Hinchey (1996) Challenges in the Successful Transfer of Formal Methods Technology into Industrial Applications. *Information and Software Technology*, 38(5):313–322, May 1996.
229. * L.T. Semmens, R.B. France & T.W.G. Docker (1992) Integrated Structured Analysis and Formal Specification Techniques. *The Computer Journal*, 36(6):600–610, December 1992.
(See page 533 in this collection.)
230. E. Sekerenski & K. Sere (1998) *Program Development by Refinement: Case Studies Using the B Method*. Springer-Verlag, Formal Approaches to Computing and Information Technology (FACIT) series, London.
231. M. Shaw & D. Garlan (1996) *Software Architectures: Perspectives on an Emerging Discipline*. Prentice Hall, Englewood Cliffs.
232. M.L. Shooman (1983) *Software Engineering: Design, Reliability and Management*, McGraw-Hill, New York.
233. I. Sommerville (1996) Safety-critical Software. In *Software Engineering*, 5th edition, Addison-Wesley, Harlow, England, Chapter 21, pp 419–442, Part 4: Dependable Systems.
234. I. Sommerville & P. Sawyer (1997) *Requirements Engineering: A good practice guide*. John Wiley & Sons, Chichester.
235. R.A. Spinello (1997) Liability, Safety and Reliability. In *Case Studies in Information and Computer Ethics*. Prentice Hall, Upper Saddle River, New Jersey, Chapter 8, pp 190–232.
236. J.M. Spivey (1992) *The Z Notation: A Reference Manual*, 2nd edition. Prentice Hall International Series in Computer Science, Hemel Hempstead & Englewood Cliffs.
237. R.M. Stein (1992) Safety by Formal Design. *BYTE*, p 157, August 1992.

238. R.M. Stein (1992) *Real-Time Multicomputer Software Systems*. Ellis Horwood, Series in Computers and their Applications, 1992.
239. S. Stepney (1993) *High Integrity Compilation: A Case Study*. Prentice Hall, Hemel Hempstead.
240. N. Storey (1996) *Safety-Critical Computer Systems*. Addison-Wesley, Harlow, England.
241. W. Swartout & R. Balzer (1982) On the Inevitable Intertwining of Specification and Implementation. *Communications of the ACM*, 25(7):438–440, July 1982.
242. A.S. Tanenbaum & R. van Renesse (1988) A Critique of the Remote Procedure Call Paradigm. In R. Speth, editor, *Research into Networks and Distributed Applications – EUTECO 1988*, Elsevier, pp 775–783.
243. M.C. Thomas (1993) The Industrial Use of Formal Methods. *Microprocessors and Microsystems*, 17(1):31–36, January/February 1993.
244. M. Thomas (1996) Formal Methods and their Role in Developing Safe Systems. *High Integrity Systems*, 1(5):447–451. Workshop report 20/3/95, IEE, London, UK. URL: *http://www.iee.org.uk/PAB/Safe_rel/wrkshop1.htm*
245. I. Toyn, D.M. Cattrall, J.A. McDermid & J.L. Jacob (1998) A Practical Language and Toolkit for High-Integrity Tools. *Journal of Systems and Software*, 41(3):161–173, June 1998.
246. A.B. Tucker, Jr., editor (1997) *The Computer Science and Engineering Handbook*. CRC Press.
247. K.J. Turner, editor (1993) *Using Formal Description Techniques: An Introduction to Estelle, LOTOS and SDL*. John Wiley & Sons, Chichester.
248. The University of York (1998) *Modular MSc in Safety Critical Systems Engineering, 1998/99: Diploma, certificate & short courses*. Department of Computer Science, The University of York, Heslington, York YO1 5DD, UK, 1998. URL: *http://www.cs.york.ac.uk/MSc/SCSE/*
249. S.H. Valentine (1992) Z- -. In [194], pp 157–187.
250. D.R. Wallace, D.R. Kuhn & L.M. Ippolito (1992) An Analysis of Selected Software Safety Standards. *IEEE AES Magazine*, pp 3–14, August 1992.
251. P.T. Ward & S.J. Mellor (1985) *Structured Development for Real-Time Systems*, Yourdon Press, New York.
252. M.W. West & B.M. Eaglestone (1992) Software Development: Two Approaches to Animation of Z Specifications using Prolog. *Software Engineering Journal*, 7(4):264–276, July 1992.
253. B.A. Wichmann (1998) Can we Produce Correct Programs? *The Computer Forum*, IEE, London, UK, 16 October 1998. URL: *http://forum.iee.org.uk/forum/*
254. S.P. Wilson, T.P. Kelly & J.A. McDermid (1995) Safety Case Development: Current Practice, Future Prospects. In R. Shaw, editor, *Proc. Twelfth Annual CSR Workshop*, Bruges, Springer-Verlag.
255. * J.M. Wing (1990) A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23(9):8–24, September 1990.
(See page 167 in this collection.)
256. L.A. Winsborrow & D.J. Pavey (1996) Assuring Correctness in a Safety Critical Software Application. *High Integrity Systems*, 1(5):453–459.
257. J.C.P. Woodcock & J. Davies (1996) Using Z: Specification, Refinement and Proof. Prentice Hall International Series in Computer Science, Hemel Hempstead & Englewood Cliffs.
258. J.C.P. Woodcock, P.H.B. Gardiner & J.R. Hulance (1994) The Formal Specification in Z of Defence Standard 00-56. In [35], pp 9–28.
259. J.C.P. Woodcock & P.G. Larsen, editors (1993) *FME'93: Industrial-Strength Formal Methods*. Springer-Verlag, Lecture Notes in Computer Science, volume 670.

260. J.B. Wordsworth (1996) *Software Engineering with B*. Addison-Wesley, Harlow, England.
261. E. Yourdon & L.L. Constantine (1979) *Structured Design*, Prentice Hall, Englewood Cliffs.
262. E. Youdon (1989) *Modern Structured Analysis*, Yourdon Press/Prentice Hall, New York.
263. E. Youdon & C.A. Argila (1997) *Case Studies in Object-Oriented Analysis and Design*. Yourdon Press/Prentice Hall, New York.
264. A.M. Zaremski & J.M. Wing (1997) Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methology (TOSEM)*, 6(4):333–369, October 1997.
265. P. Zave & M. Jackson (1997) Four Dark Corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methology (TOSEM)*, 6(1):1–30, January 1997.
266. P. Zave & R.T. Yeh (1982) An Operational Approach to Requirements Specification for Embedded Systems. *IEEE Transactions on Software Engineering*, SE-8(3):250–269, March 1982.
267. H.S.M. Zedan, editor (1990) *Distributed Computer Systems*. Butterworths, London, 1990.
268. Z Standards Panel (1995) *Z Standard: Z Notation Version 1.2*, ISO Panel JTC1/SC22/WG19 / BSI Panel IST/5/-/19/2, Committtee Draft (CD), 14 September 1995.
URL: *http://www.comlab.ox.ac.uk/oucl/groups/zstandards/*

*References marked * are reprinted in this collection.*

Author Biographies

Jonathan Bowen studied Engineering Science at Oxford University. Subsequently he worked in the electronics and computing industry at Marconi Instruments and Logica. He then joined Imperial College in London and was employed as a research assistant in the Wolfson Microprocessor Unit.

In 1985 he moved to the Programming Research Group at Oxford University where he initially undertook extended case studies using the formal notation Z on the Distributed Computing Software project. He has been involved with several successful proposals for European and UK projects in the area of formal methods involving collaboration between universities and industry. As a senior research officer he managed the ESPRIT REDO and UK IED SAFEMOS projects at Oxford and and the ESPRIT ProCoS-WG Working Group of 25 European partners associated with the ProCoS project on "Provably Correct Systems".

Bowen has given many presentations at conferences and invited talks at academic and industrial sites around Europe, North America and the Far East. He has authored over 130 publications and has produced eight books in the area of formal methods, especially involving the Z notation. He is Chair of the Z User Group, previously served on the UK Safety Critical Systems Club advisory panel, and was the Conference Chair for the Z User Meetings in 1994, 1995 and 1997. He has served on the program committee for many international conferences, and is a member of the ACM and the IEEE Computer Society. In 1994 he won the Institute of Electrical Engineers Charles Babbage premium award for work on safety-critical systems, formal methods and standards [46, see also pages 485–528 in this volume].

Since 1995, Bowen has been a lecturer at The University of Reading, UK.

Mike Hinchey graduated from the University of Limerick, Ireland *summa cum laude* with a B.Sc. in Computer Science, and was awarded the Chairman's prize. He subsequently took an M.Sc. in Computation with the Programming Research Group at Oxford University, followed by working for a Ph.D. in Computer Science at the University of Cambridge. Until 1998 he was a member of faculty in the Real-Time Computing Laboratory in the Dept. of Computer and Information Science at New Jersey Institute of Technology.

He is an Associate Fellow of the Institute of Mathematics, and a member of the ACM, IEEE, American Mathematical Society and the New York Academy of Sciences.

Hinchey has lectured for the University of Cambridge Computer Laboratory, and taught Computer Science for a number of Cambridge colleges. He is Director of Research for BEST CASE Ltd., and also acts as an independent consultant. Previously, he has been a research assistant at the University of Limerick, a Systems Specialist and Network Specialist with Price Waterhouse, and a Research Engineer with Digital Equipment Corporation.

He serves on the IFAC Technical Committee on Real-Time Software Engineering, and is editor of the newsletter of the IEEE Technical Segment Committee on the Engineering of Complex Computer Systems. He is Treasurer of the Z User Group and was Conference Chair for the 11th International Conference of Z Users (ZUM'98). He has published widely in the field of formal methods, and is the author/editor of several books on various aspects of software engineering and formal methods.

Hinchey is currently a professor at the University of Nebraska at Omaha, USA and the University of Limerick, Ireland.

Index