EUROPEAN WORKSHOP ON INDUSTRIAL COMPUTER SYSTEMS

TECHNICAL COMMITTEE 7

(Safety, Reliability and Security)

Guidance on the use of Formal Methods in the Development and Assurance of High Integrity Industrial Computer Systems

Parts I and II

Edited by

S O Anderson, R E Bloomfield and G L Cleland

Working Paper 4001

14, June 1998

DISCLAIMER OF WARRANTIES AND LIMITATION OF LIABILITIES

THIS DOCUMENT WAS PREPARED BY THE MEMBERS OF THE TECHNICAL COMMITTEE 7 OF THE EUROPEAN WORKSHOP ON INDUSTRIAL COMPUTER SYSTEMS (EWICS TC 7). NEITHER THE MEMBERS OF EWICS TC 7, THEIR COMPANIES, NOR ANY PERSON ACTING ON BEHALF OF ANY OF THEM;

(A) MAKES ANY WARRANTY OR REPRESENTATION WHATSOEVER, EXPRESS OR IMPLIED, (I) WITH RESPECT TO THE USE OF ANY INFORMATION, APPARATUS, METHOD, PROCESS, OR SIMILAR ITEM DISCLOSED IN THIS REPORT, INCLUDING MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, (II) THAT SUCH USE DOES NOT INFRINGE ON OR INTERFERE WITH PRIVATELY OWNED RIGHTS, INCLUDING ANY PARTY'S INTELLECTUAL PROPERTY, OR (III) THAT THIS REPORT IS SUITABLE TO ANY PARTICULAR USER'S CIRCUMSTANCE; OR

(B) ASSUMES RESPONSIBILITY FOR ANY DAMAGES OR OTHER LIABILITY WHATSOEVER RESULTING FROM YOUR SELECTION OR USE OF THIS DOCUMENT OR ANY INFORMATION, APPARATUS, METHOD, AND PROCESS, OR SIMILAR ITEM DISCLOSED IN THIS DOCUMENT.

EWICS TC 7

Formal Methods Sub-group

The Formal Methods sub-group of EWICS TC7 has compiled this document. Contribution, both great and small have been made by many who have both attended the quarterly TC meetings, and who have reviewed drafts and written new text between meetings. Active members of the Formal Methods sub group during the period of preparation of this guideline are:

Dean Ayres	AEA Technology	U.K.
Stuart Anderson	University of Edinburgh	U.K.
Reinhold Bariess	Univ. Darmstadt	Germany
Robin Bloomfield	Adelard	U.K.
John Brazendale	Health and Safety Executive	U.K.
Andy Coombes	University of York	U.K.
G L Cleland	University of Edinburgh	U.K.
Jan Gayen	T.U.Brauschiwig	Germany
Christopher Gerrard	Gerrard Software	U.K.
Peter Daniel	GEC-Marconi Security	U.K.
Peter Froome	Adelard	U.K.
Geoff Duke	ICS PLC	U.K.
Edward Fergus	Health and Safety Executive	U.K.
Wolfgang Ehrenberger	Fachochschule Fulda	Germany
Peter Froome	Adelard	U.K.
Janusz Gorski	Technical University of Gdansk	Poland
Bill Johnson	Du Pont	U.S.
Fiona Maclennan	Lloyds Register	U.K.
Vic Maggioli	Feltronics	U.S.A.
Michael Marhoefer	Siemens	Germany
Meine van der Meulen	Simtech	Netherlands
Elisabeth Noe-Gonzales	Merlin Gerin	France
Keith Purves	ERA Technology	U.K.
Johannes Reiner	Federal Test and Research Centre, Arsenal	Austria

Contents

Part I

1 About this document

1.1 Scope

This is a guidance document on the use of formal methods in the development and assurance of industrial computer systems. It is a companion document to Part III—the Formal Method Directory. It is intended to provide an introduction for the inquisitive, and an appraisal of the current state of these methods and assistance to those wishing to use these techniques. The document is not a textbook nor does it pretend to be an exhaustive survey: tutorial material and surveys are referenced where available. It is hoped that it is sufficiently detailed to be useful.

1.2 Intended audience

This document is intended for project managers, developers, users, assessors and procurers of high integrity computer systems. It assumes a background in existing practice (quality management systems, structured methods) as outlined in other EWICS TC7 Guidelines [20, 21, 22], and an awareness of existing and emerging standards such as [41, 30, 29].

1.3 Structure of the Document

The Guideline is in three Parts. The first is this section and contains details for facilitating the use of the Guideline. Part II provides guidance and Part III is a supporting directory of formal methods.

1.4 How to use the Document

All readers should familiarize themselves with Part I of the document. Those seeking an overview of the issues should concentrate on Section 3 in Part II, Section 4.1 and Section 5 in Part II and consult Part III, the Directory, selectively. There is no substitute for actually reading what is offered so readers are urged to complete Parts I and II of these Guidelines. Part III is a Directory and by its very nature will probably be rarely read from beginning to end.

1.5 Relationship to other documents

The document provides practical advice for those wishing to use or evaluate formal methods in an industrial environment. As such it is a companion to standards that require or recommend the use of formal methods such as those from the IEC [30, 29], the UK MoD [40, 31] and in the security field the Information Technology Security Evaluation Criteria [1].

1.6 Conformance Clause

This document is intended to provide *guidance*. It is not in a form where conformance has any significance; any claim for conformance should be treated with care.

Part II—Guidance on the use of Formal Methods

2 Background

Software systems are being increasingly relied upon within critical applications, for example in process control, nuclear protection, fly-by-wire aeroplane systems and car engine management. Large computer systems are also used to organize important activities in society, like social services, medical care, banking etc., and in such systems there is a necessity to assure safety, security and integrity.

The increasing requirement for the dependability of computer system calls for the use of rigorous methods in the development and validation of such systems. Formal methods, that is methods with formal mathematical basis are the most rigorous methods, and are being recommended where the requirement to safety and/or security is very high. Emerging national and international safety standards and the European Harmonized Security Evaluation Criteria all recommend or mandate the use of formal methods for these critical systems.

In spite of the arguments and requirements for the use of formal methods, and the fact that formal methods have been available within the research community for at least a decade, such methods are still not widely used in practical applications. One of the reasons for the poor rate of adoption is the lack of appropriate and timely information to assist those involved with procurement, management, design, testing and certification of critical systems. This document provides an introduction to formal methods including their impact on the software and systems engineering process; a discussion of their advantages and problems; and advice on the selection of techniques. Part III contains a directory of relevant techniques consisting of a brief description and evaluation of these techniques, as well as sources of further information such as tool suppliers or organisations which can provide support for the method.

It is hoped that this document reflects current best practice, however the field is moving rapidly and it is hoped that users of this document will provide feedback information to the editors (or indeed contribute further information) so that the document can maintain accurate and timely information.

3 Introduction to formal methods

3.1 What are Formal Methods?

The term “formal methods” has come into common use (and abuse) during the past decade. In this guideline we take a fairly liberal interpretation of the term. We include, for example, not only “mainstream” formal methods such as Z, VDM, CSP and CCS, but also other programming and system design paradigms which are underpinned by discrete mathematics, for example code generation and transformation, techniques and tools for static analysis of programs, and programming languages with sound semantics. Not included are structured methods, such as JSD or Yourdon, dataflow approaches, or object-oriented programming although we are interested in the integration of formal methods with these methods. The UK MoD Interim defence Standard on Safety-Critical Software [40] defines a formal method as

> *A software specification and production method, based on a mathematical system, that comprises: a collection of mathematical notations addressing the specification, design and development phases of software production; a well-founded logical system in which formal verification and proofs of other properties can be formulated; and methodological framework within which software may be verified from the specification in a formally verifiable manner.*

This is a rather ambitious definition. As we describe below such methods are attractive, but in practice most formal methods in common use do not address the full spectrum of design, some supporting specification phases, some the construction phases, and others the analysis of systems.

Typically, formal methods have three components:

- a notion of "program"[1],
- a means of expressing properties of the computation of programs, and
- some method for establishing whether a program has some property.

Each of these is rigorously mathematically defined. This is a fairly liberal description of a formal method. It covers things like the type inference system of some modern programming languages which "prove" programs have a well-formed type, to entirely general purpose theorem proving systems or proof assistants, or paper and pencil methods. The quality of evidence one gets from a formal method varies according to the method but generally it is highly accurate, is convincing to trained workers, and often has a rather narrow coverage because it abstracts from many detailed features of systems.

There is also a wide range of formality and rigour applied in the use of formal methods. A *formal proof* might consist of justifying a conjecture by deriving it from the basic axioms of the mathematics upon which the logical system in use is based. A *rigorous argument* looks more like an outline of how a proof might proceed, but would not supply all the intervening detail. Additionally, rigorous argument may draw upon a rich body of known results, but without the need for formally integrating them into a proof. Note that this is how engineers and mathematicians usually work—they customarily have a commonly understood context which obviates the need to descend into detail which obscures the main subject. However computer based proof tools do not have the insight of mathematicians, and cannot interpolate unstated detail between steps. Computer based proofs are therefore usually very detailed, and can be extremely obscure, although their validity is potentially less contentious than a rigorous argument.

The mathematical basis of formal methods may be an existing part of mathematics—for example the Z specification language has set theory at its heart—or can be developed anew for the method—the Calculus of Communicating Systems has a theory presented in an algebraic style, but specific to the Calculus.

---

[1] The notion of program should be interpreted very liberally. It need not conform closely to "conventional" programs. Along with the notion of program will come a well defined notion of what it means to compute with that program.

There is a large and growing variety of formal methods, of varying age and maturity. It must be realised that there is no "standard" formal method. Each technique has particular strengths and weaknesses and it may be that in the course of systems development it is appropriate to use a number of methods at different stages of the process. Just as in traditional engineering, no single theory encompasses all aspects of design and development. Many theoretical approaches are used—either explicitly or implicitly. "Stock answers" are often brought to bear and, even if these appear not to involve mathematics directly, they usually have a long history of mathematical analysis which has consolidated into a precise understanding of the nature and behaviour of a particular aspect of the component or structure. The interaction and compositional properties of such components are well understood and the properties of the whole (strength, weight, current, speed, loading, cost, time to construct, etc.) can all be predicted with much more accuracy than is now possible with comparable computer systems development.

The need for and impact of this use of theory is also well understood by managers, engineers' licensors and procurers. Comparable use of methods or theories is currently not so diverse in the area of computer systems engineering and it is desirable that engineers cultivate a broad awareness of possible methods which are relevant to different aspect of systems design and develop the ability to assess the usefulness and applicability for particular applications of competing or complementary approaches.

Just as there is no standard formal method, there is (as yet) no standard way of applying any particular formal method. As case study material and tools are developed to support the engineering domain, this may change. However until then it is likely that engineering application of formal methods will require to be open in nature with the partial success or failure of certain approaches expected and the results of this failure used to inform more successful application.

As with other engineering disciplines, a body of "good practice" should provide guidance to engineers as to effective use of techniques within different contexts. Unfortunately with formal methods this material is sparse, of variable quality, and not well indexed. In due course this body of information will grow, in particular it is hoped that a range of texts and real-world case studies will be produced.

We now go on to present the capabilities and limitations of formal methods, and guidance for their uptake.

3.2 The Nature of Formal Methods

Software and systems engineering have a number of different life cycle models, however most break down into a number of phases of activity such as: requirements, specification, design, implementation and test.

A common view of the use of formal methods in this regime might expect the following. The customer interacts with the supplier closely over requirements. The supplier writes a formal specification, which is then successively refined as implementation detail is added until an implementation results. This is then subjected to test. The use of formal methods at the various stages of refinement can mean that the final implementation in some sense is proven to satisfy the top-level specification. The 'proof of satisfaction' is one of the documents which may be required by the certification authority as part of any licensing procedure.

Traditionally this kind of approach has been seen as the goal of formal methods research and there is still considerable academic investigation into approaches which aim to achieve this. However the level of "correctness" delivered must be interpreted within a broad engineering framework—the meaning of the word to the customer may be very different to that assumed by the designer. Absolute correctness is unattainable—it is not a concept that is familiar to (or sought by) engineers in other disciplines, they do however make extensive use of mathematics to model, design and analyse.

In this hierarchy the requirement and specification layer generally say mostly *what* the system is required to do. The lower layers (and there is usually more than just one step at each level) contain increasing amounts of implementation detail *how* the system is to achieve its function. Correspondingly the amount of information at each level increases as we descend, usually very rapidly.

Let us comment on and elaborate this model. There are several observations to be made. Note that many of these are not necessarily restricted to formal methods, but are often true of software engineering in general.

Requirements are never right.

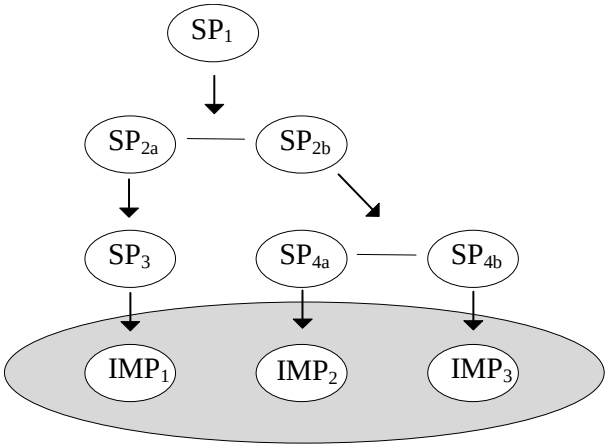
The symptom of the supplier delivering what he was asked for and the customer only then realising that the original requirements were wrong or ambiguous is common. This is one of the major problems in systems and software engineering. Close interaction between the customer and the supplier is desirable at all stages of development, but identifying problems as early as possible in the life-cycle is highly desirable. Use of formal methods at the requirements stage can increase the clarity of understanding and so reduce the scope for misinterpretation with the corresponding saving of wasted development effort, saving both time and money.

One approach is to use a formal method that supports executable specifications. This does indeed allow experimentation with statements of requirements and can result in a significant "feel good" factor for the customer as they have something concrete (i.e. runable) in their hands early on. However, this can mean that implementation level decisions (how rather than what) are made too early. Inappropriate use of such languages can restrict freedom at the design stage and can remove the possibility of simplification or generalisation of the system. It may be preferable to develop a prototype than to use an executable specification language. However any prototype should be developed from the point of view of informing the requirements analysis rather than guiding the implementation.

It is important that formal languages for requirements or specification are used in the appropriate style and not as programming languages. Avoiding this trap can be difficult if one is moving existing programming staff into the use of formal specification languages.

Real world problems are complicated.

In some cases it is possible nowadays to "verify" code against specifications for some systems of perhaps up to 20,000 lines of code. However most systems nowadays are much bigger than this and to apply the above paradigm to large systems we need to support the idea of decomposition as well as refinement:



Here a top level specification (SP_1) is decomposed and refined into two subsidiary specifications (SP_2 and SP_3). Note that there are two kinds of relationship here. Horizontal lines are a glue that binds components together and the vertical connection represents the refinement we have talked about already. Thus each level is guaranteed to implement the specification it "descends" from. Ultimately we end up with a number of executable modules (IMP_{1-3}) which implement SP_1.

In this model the implementations delivered at the bottom are linked formally only at levels above this. This means that we need an integrated theory of refinement and decomposition as well as an implementation language which supports the communication behaviour of these "modules" with respect to the higher level specification of decomposition. In practice this means the semantics of both the specification language and the implementation language must be related formally so that the behaviour of the implementation is adequately proven to satisfy the higher level specification.

An alternative approach is to use a single "wide spectrum" language. These are languages which support both specification and implementation activities. The term "wide spectrum" is used to refer to the spectrum of activities in system development. In some cases they provide mechanisms for verifying the code against specification, or support formal refinement of code from specifications. Although these exist in prototype form this area is still the subject of much research.

One benefit of the approach in figure 1 is the separation of concerns possible under this model. It may be possible to isolate critical aspects of system behaviour in a small number of components. The use of formal methods to ensure integrity of these components can then be more focused. Traditional techniques may then be adequate for the remainder, although great care must be taken over interfaces.

Formality is internal to the model above.

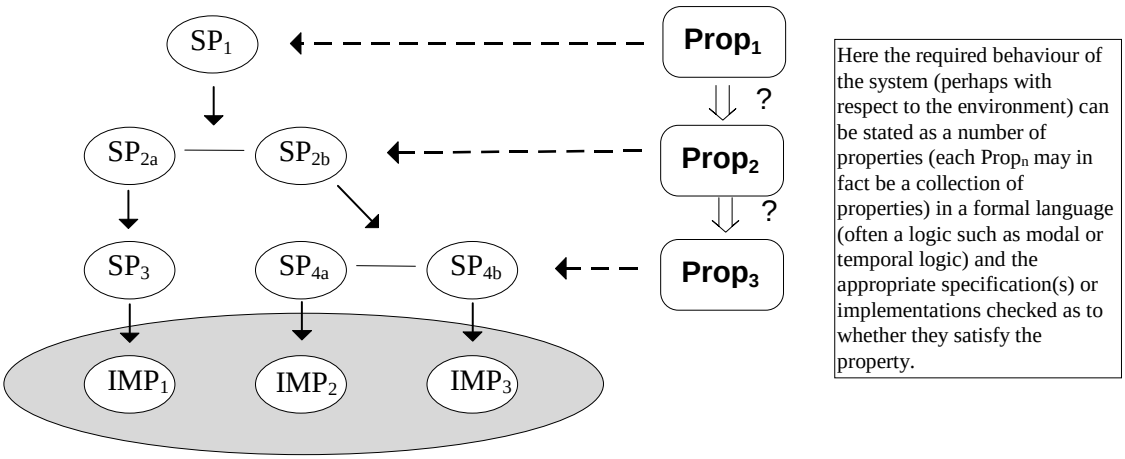
Even if the formal proof of correctness of implementation against specification exists this still only represents a part of the final system. The implementation will be expressed in some high level programming language. This will require to be compiled to object code, probably with the linking in of library components. This then runs on some hardware. It is possible to verify components such as compilers, library function and hardware, but these would generally use different techniques and to reason about the "correctness" of the composition of all of these is a considerable task requiring significant intellectual and economic investment. These kinds of activities usually require strategic sponsorship, and involvement of academic groups, although we can expect results of this work to become viable (technically and economically) in the industrial context during the next decade.

Other problems arise in that the resulting system has to interface with the environment which includes unpredictable entities like people or analogue components often in an asynchronous environment. Here the main problem is one of *validation* of the system with respect to its

environment. Here again formal methods can help. One can model interfaces, and use these to explore the operating conditions of the plant, thus exposing the requirements on the operator, as well as the system. One can exercise a formal specification of a system with respect to an environmental model such as a fault tree analysis (which itself may be expressed in a formal language).

Even if code is verified with respect to a specification, it must then be compiled to execute on target platforms, nowadays the simplest of which have executives, and increasingly, full blown operating systems which support the application code. Nowadays these are intricate and highly specialised, and generally not amenable to formal verification. Unlike application code, however, compilers and operating systems may have a significant history of successful operation, and this can often be used to justify their use. Ultimately the system is executed on physical components (wire, glass, silicon etc.) which (at least in this context) are also analogue in nature. The behaviour of all of this combined is certainly impossible to model formally so one must ultimately call a halt to the process of full verification and decide what concrete benefit the application of formal methods can actually give.

In practice one must choose from a range of approaches and it worth extending the model above with the idea of property oriented specification:



Typically these properties will state some aspect of dependability required of the system, such as safety, security, reliability and performance. In a fully formal model of this kind one would expect properties for lower levels of specification to include higher level properties plus some more which are relevant to that level of implementation. For example, one would expect performance related properties only to be relevant for verification and validation purposes at the lower levels of specification or implementation.

Such approaches may require the replacement of much of existing systems engineering practice.

It may be appropriate for a limited number of systems (or clearly delineated components) which are small enough and critical enough to consider “full” verification in conjunction with good testing practice. For most applications however this will not be appropriate. It would require a major change to systems engineering practice. Formal methods are by and large based upon technological approaches, whereas the successful management of the development of computer systems is as much a social problem. Much of the systems engineering research in the past two decades which has successfully been deployed in industry is focused around the management of people, activities and communication. “Traditional” formal methods do not fit well with these

approaches with their strong focus on the technical object being produced (the program) rather than the teams doing the production (the programmers, designers, testers etc.).

Application of formal methods *is* possible in ways which will integrate and provide complementary strengths to existing methods. By using this approach it is possible to evolve existing practice to integrate these new approaches. It also handles the risk problem well in that the consequences of failure are localised and can usually be mitigated by reverting to already well known and understood good practice.

3.3 Benefits in the use of Formal Methods

As described above formal methods can be used to provide a high degree of assurance of a system's correctness with respect to a specification. While in some applications the assurance that "proof of correctness" itself is the main goal, in many other applications the primary benefit of formal approaches to system design is not the "proof", but the increased understanding of the problem which the *process* of proving or formally specifying provides, and the increase in confidence in and support for the design process (and final product).

In addition to the benefits of formal specification or proof of correctness there are many other ways in which use of formal approaches can be beneficial. Note that these are not achievable with every formal method, or with each application, but the benefits can include:

To raise engineering understanding of the problem or application.
The process of thinking formally about a system will almost certainly force clearer thinking, and hence understanding, about the nature and purpose of the system being built. This benefit alone can justify the use of formal methods.

To raise customer confidence in the product.
Delivering a faulty product can do enormous damage to your customer base. While formal methods are unlikely to remove all errors, they can be used to analyse critical aspect of system behaviour, and potentially eliminate specific classes of errors.

To raise confidence in the development method.
This then supports the supplier in making clearer claims about the quality/cost/ development time and effort of a product.

To document the design process unambiguously.
This can help all those involved: the project manager, designer/engineer, certifier/validation and verification team, maintainer, customer.

To reduce maintenance effort and cost.
Much of the life-cycle cost of current systems is not in the original design, but in system support, modification and maintenance. The clear documentation produced when formal methods are used results in significant savings in this phase of the life-cycle. The cost of rectifying a fault once a system is in operation is orders of magnitude greater than correcting it at the design stage. In addition, some formal methods provide active support during maintenance stages by identifying the scope of alterations and thus limiting the re-testing required of the modified system.

To identify mistakes and omissions early
Formal approaches tend to detect problems early in the design cycle when they are therefore much easier and cheaper to correct.

To shorten the time to market.
Despite the fact that more effort needs to be invested in the specification phases when no deliverable code is produced, the resulting system should have fewer errors. Having to re-engineer a product after being released and found to be faulty can cost enormous amounts in lost market share. Experience has shown that it is straightforward to produce code quickly and efficiently from formally produced specifications. Significant gains in programming team productivity is therefore possible.

To market your product/service/company
An ability with formal methods in your organisation may confer a significant competitive advantage in certain markets e.g. security or safety. Indeed in some sectors it may not be possible to tender without ability in formal methods.

To protect yourself/company
Product liability laws may leave you exposed to costly litigation in the event of disaster or non-performance, if you cannot demonstrate that you have adopted the best design and validation techniques available and appropriate.

To aid the certification process
The use of formal methods is increasingly being advised (and in some cases mandated) in standards and guidelines. Their use can therefore be a major aid in convincing a certifier that the system is safe to deploy.

3.4 Limitations to the use of Formal Methods

It is important to realise limitations in the use of formal methods, as well as their benefits. Outlined below are some of the disadvantages or potential problems which may arise and should be taken into account in making a decision on whether to use formal methods.

Formal methods can be expensive to do
Perhaps more accurately—formal methods *will* be expensive the first time you try them. It is important to amortise start-up costs across a spread of projects and time and to start out with simple and easily achievable aims, ramping up as technical experience grows. This said, the use of experienced consultants in the early stages on use of formal methods will usually be a low risk and cost effective route to adoption.

There is no formulaic approach
Case study material is sparse. This will improve in time and with experience, but you must expect to view projects from an applied *research* perspective rather than established engineering *practice*. Appropriate risk management techniques must be used. The use of appropriately experienced consultants for early ventures in the field can reduce both cost and risk.

Training time
This varies by method, but it takes at least several months for even a well motivated engineer to become familiar with the use of formal methods in design. (Attaining a *reading* capability can be much quicker though)

Difficulties of communications
Using formal methods to communicate can increase accuracy, but does require a change in attitude, and some training. This can be a particular problem with customers and management who do not have the *technical* incentive to become familiar with them. Integrating formal methods into existing development methods, and proper use of interspersed text and engineering diagrams and notations, can mitigate the problem.

The complexity/size problem
Although formal methods are improving their range of application the problem of scaling up to larger examples can be problematic. Abstraction is a common approach in many engineering disciplines for handling such complexity. In systems engineering however our understanding of approaches to abstraction is not well developed. Again the use of skilled consultants or staff can help here.

Formal methods are weak on performance
Research is improving in this area, for example in the development of timed calculi, and in integrating with more traditional performance approaches, but it will be some time before these mature.

Poor formal infrastructure
While principles of applications of formal methods are well understood, maximum benefit requires adequate formal infrastructure. For example almost all programming languages do not have a *usable* formal semantics. This makes the behavioural analysis of code difficult, even though formal methods might have been used earlier in the development cycle. Similarly no *usable* verified compilers exist, so how can one be sure of the "correctness" of object code.

Cultural uncertainty surrounding formal methods
Formal methods are a moving target. Which ones work in what domain? What is next year's method going to be? Will I have to completely retrain?

The validation problem
Formal methods apply to abstractions. Real systems are too complicated to analyse fully. (This is true in traditional engineering as well though.) Can I validate my model with respect to the expected behaviour of the whole system? What level of requirement/specification is appropriate?

Lack of tools
Although the situation is improving tool coverage is still poor, and doing formal methods by hand is time consuming, error prone and usually will require a higher level of expertise. It's a bit like trying to write a program without a compiler. Even simple tools such as syntax and type checking tools can strongly support the use of formal methods.

Formal methods do not substitute for creative thinking
In fact even more is required to bring together traditional design capability with the new approach. It is important that existing expertise and practice are properly utilised with the introduction of formal methods.

3.5 The Investment/Benefits Balance

The relationship between investment and benefit across domains is not always clear. Here we try to characterise the range of investment required to achieve a given benefit from the use of formal methods.

Customer Even with no investment there may be an increase in perceived assurance of product or of process. The most likely investment route here is to buy in external expertise. By investing to the level where a customer can read specifications they begin to be able to *measure* assurance levels and to *contract* for a precise deliverable.

Developer The common perception is of high engineer training costs. It is interesting to contrast this with industrial expectations of uptake of other software technologies. A two week course is often adequate for a reasonable grounding in the Ada language, but a project manager would not expect such a course to immediately confer an ability to program seriously in the language. This ability grows with use and guidance from peers and mentors over a period of months. So it is with formal methods—adoption costs should be amortised over several projects, but may result in a significant increase in productivity, a possible reduction in the skill base required to conduct the work, and an increase in the scope and scale of the method supported by the tool.

Assessor/IVV By developing a reading and validating capability an assessor can verify that a process satisfies relevant standards or guidelines. By investing in tools this will provide additional ability to assess product quality in a localised way. In general to assess product claims an assessor must invest to the same level as the developer, whereas process claims may be assessed with a lower comparable level of expertise.

Regulator By buying in expertise they get access to accepted best practice which then is represented in standards and guidelines.

3.5.1 Principles of successful Applications of Formal Methods

The benefits outlined above are not readily realised. There are “no free lunches” when it comes to software engineering and the successful application of formal methods requires planning, commitment and resources. As with any new technology the rationale for adopting it has to be established; the likelihood of achieving the benefits assessed; and a judgement made as to whether the resource is available and whether costs are justified. We list here the main principles considered necessary for the successful use of formal methods. These are further elaborated in Section 5.

3.5.2 Summary of principles

Ask yourself the following questions:

Appropriateness of problem

- Is there a match between problem and method(s) to be used which will realise the benefits?

- Scalability, complexity issues—will the method cope with the size of the problem, (e.g. are there abstractions which are tractable, but are still a fair representation of the problem)?

- Is there a clear identification of expected benefits? (Are they justified by the cost?)

Technical basis

- Do I have or can I get technical people with skills, both in the methods to be used *and* in the problem domain?

- Have I assessed the scope and possible limitations of the methods in the domain?

- Is there appropriate tool support?

- Is there quality information on the method? In particular an understanding of what has and hasn't been achieved on other projects.

Correct management approach.

Do I have:

- an understanding of the implications of use of new technology on my organisation and business?

- the organisational maturity to admit new technology without adversely affecting existing process quality?

- customers who support the use of formal methods and are prepared to cooperate and understand the approach we are to adopt?

- a realistic appraisal of benefits and costs linked to risk management?

- a plan for evolutionary adoption of formal methods?—Walk before you run.

4 The Impact of Formal Methods on Software and Systems Engineering

4.1 Introduction

The aim of this section is to give an overview of the impact of formal methods on Software and Systems Engineering (SSE). We first consider this from the viewpoint of the impact on the evidence produced during SSE before looking at the impact on the activities. This is done first in general and then for the IEC 61508 [29] safety life cycle. In addition the impact on some classes of system component, system quality and safety standards is considered. The aim is not to be exhaustive but to give an idea of the main effects of the use of formal methods.

4.2 Evidence

4.2.1 The role of evidence in SSE

Software and Systems Engineering (SSE) is a complex process. This makes characterising it difficult. Rather than fix on some particular characterisation of SSE this section introduces a broad categorisation of activities undertaken in SSE, based on their use of evidence. Any engineering activity requires decision making. Such decisions are motivated and justified by evidence. Thus the emphasis of this section is on the role of evidence in SSE. The impact of formal methods is primarily in providing new ways of obtaining particular kinds of evidence about systems and designs. After considering the impact in general, the effect on particular aspects of SSE is discussed.

The choice of this characterisation is an attempt to unify the underlying concepts evident in most approaches to SSE without being overly prescriptive. Ultimately the concern of the systems/software engineer is with the *behaviour* and *suitability* of the finished system. Thus the engineer is concerned to define what is the required behaviour, give the means to observe the behaviour, and provide evidence that the constructed system will display the required behaviour in all circumstances. Concentrating on behaviour means the distinction between systems and software engineering is no longer significant since hardware and software make equal contributions towards system behaviour. Although the development of software may proceed semi-autonomously (and the design and development process may benefit in efficiency terms) ultimately the total system design is the chief concern of all (developer, tester, user, regulator, etc.).

We emphasise the role of *evidence* in the development of computer systems. In common with other engineering disciplines, SSE involves constructing a system and making, en route, the case that it fulfils its intended purpose. This requires the management and presentation of evidence. The standards literature acknowledges that evidence plays an important role in SSE. For example, the notion of *objective evidence* is used extensively in the definition of the Ontario Hydro standard [39]. There, *objective evidence* is given the definition:

> *Written justification to support a claim which is verifiable by a third party, using the information provided.*

Use of evidence pervades SSE. The process of generating evidence and using it to justify decisions is a recurrent theme. Evidence is heterogeneous, coming from different sources and with different levels of confidence. Combining different evidence can be difficult. Effective management of a large body of evidence is fundamental to the success of any engineering project. This involves:

- Establishing a body of evidence sufficient to support the needs of a project. This involves identifying evidence needed and the qualities required of that evidence.
- Determining how to go about generating the necessary evidence (e.g. by using appropriate methods, staff and tools.
- Identifying paths to improving the evidence generating capabilities of staff and equipment engaged in SSE.

These considerations are central to good SSE practice. Generating evidence to make an adequate case may be done in a variety of ways. There is no best approach, but the past decade has seen steadily increasing requirements on the quantity and quality of evidence required to justify safety-critical programmable systems. This is likely to continue as our understanding and awareness of safety improves.

In SSE evidence is used to support two different kinds of activity:

> *Internal:* In the process of constructing a system the representations of problems and solutions undergo radical transformation. These transformations are marked out in, for example, the traditional waterfall diagram. The transition from technical requirements to design, and from design to code are radical transformations but they are internal to the production process and are to a large extent under the control of the developer. Of course the choice of the forms of internal evidence may be constrained by the external factors—but within these constraints great freedom is possible.
>
> *External* These forms of evidence attempt to relate the internal technical artefacts to the external (and usually informal) requirements. A critical component of SSE is that satisfactory external evidence is generated and that internal evidence is consistent with the external evidence. In particular it is important that external evidence is comprehensible to customers, users and regulators.

Internal evidence is the domain of the technical expert, but should be capable of independent assessment.

In dealing with external evidence one is dealing with matters which concern a wide range of communities some of whom may have little specific technical expertise. For such groups the requirement is for a minimum of unnecessary theorising. The primary concern of such groups is a convincing demonstration that the delivered system will behave as expected for its intended lifetime.

4.3 Putting Evidence Together

Within the SSE process evidence is generated, stored, managed and utilised with the objective of informing decision making during design and constructing a case that a system meets its requirements as part of review.

> *Communities of Use:* A critical aspect is consideration of the interlocking communities who will interpret the evidence generated by the project. There is a growing body of sociological evidence which points to the problems which arise when evidence is interpreted in radically different contexts [32]. The adoption of formal methods may exacerbate this problem because experts will tend to miss subtle misunderstandings made by non-experts and this can lead to a loss of assurance.
>
> *Properties of Evidence*: The quality of a collection of evidence relates to a number of properties we can enumerate (though probably not exhaustively). The following list contains some important properties of evidence and how these properties relate to formally derived evidence.
>
> *Conservativeness*: Evidence often undergoes radical transformation in the construction process, in particular in relation to representation and to the degree of abstractness. For

example, inadequate requirements analysis may lead to requirements which are not preserved by the transformations of construction and therefore may lead to blind alleys in the construction process. This may require remedial requirements analysis late in the construction process. Formally derived evidence is often rather abstract and thus is quite robust under various transformations.

Coverage: You need to check that the evidence is adequate to provide answers in sufficient detail for each question you have to answer. For example in most cases it would be entirely inappropriate to focus purely on the human interface of a system and its performance and forget the functionality (or vice versa). Formal methods usually provides good coverage of functional aspects but are weak on a variety of other areas e.g. usability or performance.

Accuracy: Accuracy may vary from question to question—how much do we really need to know? Formal methods often give highly accurate answers. For example code analysers like MALPAS [11] or SPADE [14] provide highly accurate analyses of specific program properties, so accurate that they are proven theorems. By contrast, testing is rather flexible in how much accuracy it yields and may be very costly to achieve high accuracy.

Timeliness: This is particularly relevant in the case of system construction. Many decisions are taken in the course of constructing a system. Quick answers may allow a more exploratory approach to the construction process. Such an exploratory approach may be essential to the construction of high quality systems requiring considerable novelty but may be less essential in more routine tasks. Planning and review seem to need less rapid results. An examples is the use of type checkers to provide accurate and timely information on the well-formedness of specifications and so stop the propagation of errors into later stages of the project. Also, some specification languages and their support tools can allow tests for required behaviour (e.g. the Concurrency Workbench for CCS [2], ObjEx for OBJ [23]). Again such information may provoke early review of a design. Tom Gilb [25] advocates an evolutionary approach to the construction and delivery of systems—this (if such an approach is possible for your project) allows very rapid user feedback on the direction of the system development

Appropriateness: Evidence should be directed towards a clear goal. It is probably worthless to completely specify and formally verify a particular computer-human interface, but it may be appropriate to show certain critical properties hold of the interaction, e.g. no blind alleys, that warnings are clearly displayed, that enough operator information is displayed, that the system does not livelock.

Convincingness: The authority of evidence is clearly important. In our culture mathematical evidence has a special role since our experience tells us it is particularly reliable. The traditional arguments for design and coding practices which generate small comprehensible modules are motivated by convincingness.

Independence: Diversity of evidence is important. Seeking agreement where there could have been conflict in evidence provides corroboration.

Intelligibility: If evidence cannot be interpreted it is utterly worthless. When designing an HCI it may be worthwhile having an executable specification so it is intelligible to the

customer. While in the case of, say, telecommunications, having a description devoid of jargon may make the evidence unintelligible to a group of technical experts—at least at the level of detail at which they want to work.

Efficient use of Evidence. Evidence can serve a variety of purposes, and in combination it can be used to infer additional evidence. There is a tension between dissemination and control since wide dissemination can lead to both designers and validators being swamped by evidence. Good control needs to be exercised and the identification of relevant evidence for different groups and purposes needs to be identified. Synergistic relationships between various elements of evidence should be identified and exploited in the planning of a project. An example might help here: In distributed systems some actions are independent and can be arbitrarily interleaved but will still lead to the same system state. Formally this property is called *confluence.* Having a proof of confluence of some part of a system a tester knows that a whole range of input sequences are equivalent and can test only one rather than sampling from the set. Of course, proofs can be in error and more testing might also help to raise the confidence in the proof, or provide a counter example. (This is of course another form of synergistic inter-evidence relationship.)

4.4 Impact of Formal Methods on evidence

Evidence generated by formal methods is claimed to have special characteristics which derive from their mathematical basis:

certainty: rigorous, or formal mathematical proofs claim to show that some properties *certainly* hold for a model of the system. However, such statements are always qualified by some *assumptions* about the real system. If assumptions are not made explicit they can be a source of serious failures.

generality: mathematics is the language of generality *par excellence*. Rigorous or formal proofs often claim to consider the behaviour of the system under all (or at least most) possible inputs. The ability to reason about a wide range of possible circumstance simultaneously is a powerful motivation for adopting formal methods.

Formal methods can provide evidence of very high quality when applied properly. As staff acquire capability in formal methods normally the quality of the evidence, predominantly its accuracy and convincingness, should improve. The availability of very good evidence from one source often lessens the need for evidence from another and older practices may be superseded. Adopting formal methods to generate internal evidence can have a large impact on the skills and training requirements of system developers and testers. In the best circumstances the language of formal methods can be used as a lingua franca within and between the design, test and validation groups, allowing a free flow of information. In other circumstances it can act as a barrier to such flows where only a small sub-group are familiar with the methods. The adoption of formal methods can widen the gulf between the forms of internal and external evidence, thereby increasing the difficulty of validation. The lack of such data for formal methods is one of the key factors inhibiting their uptake. Gaining experience can be difficult, and costly, but finding a route to adoption of formal methods that limits risk and allows some evaluation of their worth without committing to full-scale use is one aim of this guideline.

Through careful evaluation the evidence needed to plan the use of formal methods effectively will be acquired. There is no single “ideal” route and different approaches suit different situations. The recent NIST report [18, 19] provides some empirical evidence relating to

industrial uptake. In the design phase, formal methods' capacity to generalise and to abstract away from detail gives the designer a powerful tool. The designer can choose to analyse problems at an appropriate level, ignoring low-level detail. The evidence generated by this use of formal methods has the feature that it remains valid even when some detailed design changes are taken at lower levels of abstraction. Advocates of formal methods argue that this layered and selective generation of evidence during construction supports a flexible design process that can accommodate design changes more easily than conventional methods. They also argue that formal evidence at the design level promotes *reuse* of designs because evidence is generated at a high level of generality thereby establishing its validity across a range of possible systems. However, formal methods are unlikely to become the sole approach to demonstrating the acceptability of a system. They will become one powerful technique in the design and test teams' repertoire. The arrival of formal methods need not revolutionise system development but as their use spreads they will gradually change the balance of effort expended on the range of techniques used in system design and validation.

Using formal methods can involve significant effort but it appears they can bring significant benefits and can contribute usefully to other existing methods.

4.5 SSE Activities

4.5.1 Overview of SSE

SSE is composed of three different types of activity: *planning*, *construction* and *review*. Activities falling into each type have different requirements for evidence and use evidence in different ways.

> *Planning*: These activities establish the framework within which a project proceeds. Establishing the scope and feasibility of projects requires an accurate description of the problem and mechanisms for evaluating the feasibility of solving the problem. Planning is similar to construction but requires more empirical evidence: estimates of costs, time, scale of commitment etc. are essential to establishing the feasibility of any project. The kinds of evidence required to make accurate estimates are empirical, based on previous experience of the problem area, mix of skills in the team, experience of other groups.
>
> *Construction:* These are the primary parts of SSE. [Typically construction includes: requirements capture, design, coding, testing, and modification.] The orientation of construction is the generation of a completed artefact with the required behaviour. Evidence is generated and utilised in construction primarily to direct decisions on how to proceed with the next step. For example, in the design phase there is a myriad of possible choices for the design of each component. The task of the designer is to generate and use available evidence to constrain choice and inform decision taking.
>
> *Review:* These activities are more administrative, involved in checking that construction is on target to build a system with the required attributes: quality, safety, reliability, maintainability etc. Though these are important during construction, it is usually review activities which are responsible for showing the system has the required attributes. Usually review attempts to be all embracing in that it attempts to review itself.

4.5.2 Formal Methods and SSE Activities

Formal methods impact on the SSE activities in the following ways:

Planning: Establishing the scope and feasibility of a project is similar to the construction process. By contrast, the process of estimation involved in planning seems to rest almost entirely on empirical evidence. This can only be acquired from experience of many projects. It seems unlikely that formal methods can help in this empirical activity. Adopting formal methods in SSE can change the process radically and thus change the empirical data required to plan. Thus though formal methods can do little to contribute to planning their adoption can change the structure of plans since the balance of activities may change.

Construction: Adopting formal methods in construction activities gives a source of high quality evidence in the area of system functionality and behaviour. In outline the potential contribution of formal methods at construction activity is:

Requirements: During construction we must first capture requirements and convert them into a specification. The perennial problem here is that of validation. The correspondence between "real" requirements and the specification remains problematic. Formal specification methods allow the designer to construct an abstract model of the final system and express properties required of the system. Formally generated evidence derivable from formal specifications provide consistency checks which ensure specifications can, in principle, be satisfied. Some approaches [23] can be animated to provide empirical checks. Some other approaches allow the formal checking of the outcome of traditional validation exercises, e.g. fault tree analysis [8]. Generally formal methods are strongest in the area of functional requirements though there are now good theories of concurrent systems [2, 28, 42] and the analysis of real-time is an active area of research.

Design: Design is a difficult process, engineers find a satisfactory solution to design problems by drawing on a wide range of experience and deploying diverse methods. Early advocates of formal methods believed that the entire design process should be structured by the formal structure of the system. This led to attempts to enforce the use of a single formal method during design. The result was a great deal of frustration on the part of designers used to bringing a range of approaches to bear on a design problem. In some circumstances, however, use of a single method can result in better final products which can be demonstrated to meet their specification.

However, it is sometimes impossible to arrive at a satisfactory design using such an approach. Successful adoption of formal methods in design may require that their use is selective, that a variety of methods are used, and their use is phased in over a number of projects. This clearly has a strong impact on the training of design staff. The generality of formal methods may also be of use in design since it may be possible to reuse components if their design is suitably generalised.

Implementation: The main impact here is in the design of languages and the ability to prove the implementation correctly implements the design. Programming languages with a formally defined semantics provide the basis for this and to the construction of highly assured compilers, translators and checking tools. Examples of this are Lustre [27], Occam [43], Eiffel [34] and Standard ML [35, 36]. More traditional languages such as C, Pascal, Modula-2, or Ada do not have such clearly defined semantics, however "clean" or "safe" subsets of these languages have been engineered, where each program has a well-defined meaning independent of a particular compiler [4]. Even assembler can be used in a formal context through annotation of the code with assertions. Assertions are logical

statements that capture the essential purpose of a program. The outcome of design using formal methods is a precise statement of some of the specification and perhaps proof that the design meets the specification. The specification will impact on test design since there is a precise statement of at least some of the behaviour expected of the system. Also, some proofs will be of the form: if the system behaves this way in one situation it will behave similarly for some class of situations. This type of evidence can be useful in designing test cases.

Modification: The main impact on modification is that some precise specification of some system components will have been generated during earlier construction activities. This means that the assessment of the impact in change of specification is facilitated and the consequences of changes can be tracked more easily. An example of this is the IBM CICS project [17].

Review: Here we are looking for confirmatory evidence that a number of well described objects are compatible or that they conform to some pre-specified standard. In other words we are no longer in the position of requiring evidence (information) to make a decision; we are now in the position of checking that some previously taken decision conforms to the standards we require. The problem of validation remains difficult because we must demonstrate that the models (formal or informal) we have used in the development system conform to the real world. Here the precision of formal models can help because this does ease the identification of errors and problems. Verification of important properties of systems can usually be achieved using checking tools. Examples of checking tools are MALPAS, SPADE [37], also LUSTRE has some associated tools [27]. The problem is the risk associated with *post-hoc* verification. If serious design errors have propagated then it will be impossible to verify a property without a complete redesign. However, if the aim of the project is to verify some given properties more informal approaches e.g. code reviews can make the informal checking of formal properties one of their priorities.

4.5.3 Impact of Formal Methods on the safety lifecycle

The overall safety life cycle of IEC 61508 Part 1 is becoming an agreed international description of the typical phases in the development of a safety system. This is a brief description of the kind of role which can be played in these phases by formal methods.

1. *Concept*: Formal models can provide a means of modelling components of the overall system and defining the interfaces between such components. In particular one can work to characterise the environment, plant and control at a very high level.

2. *Overall System Definition:* The power of formal methods to *abstract* away from irrelevant detail can provide the means to precisely characterise the overall system in a concise way. Abstract models of system are usually comprehensible by the non-expert, particularly when they can be animated.

3. *Preliminary Hazard and Risk Analysis:* Much of the analysis here is probabilistic and little work in formal methods has addressed these issues. However, there is some work on the formalisation of standard approaches [8, 12] to hazard analysis. Often one can use mathematical reasoning to establish the assumptions required for some condition to arise then use informal probabilistic reasoning to estimate the probability that the assumptions hold simultaneously.

4. *Develop Overall Safety Requirements:* Determining tolerable levels of risk for particular hazards is largely based on empirical experience and thus formal methods have little to say directly about this. However, the outcome of this process is the overall safety requirements specification and it is possible to use formal methods to state the functional aspects of this in a precise manner which provides a sound basis for later work.

5. *Develop Overall Safety Strategy:* Again risk reduction strategies are largely empirical however it may be possible to use mathematical reasoning to evaluate the feasibility of different strategies. Reuse is also claimed to be ore straightforward using formal methods, advocates contend that one can reuse reasoning about particular architectures or techniques across different situations.

6. *Overall Validation Planning:* Formal methods have little to offer to planning activities but planning to incorporate formal methods in validation activities can have a significant impact. In particular, some safety requirements are formalisable as properties of the implemented system and proving these properties hold of the completed system can aid in validation

7. *Safety Related Systems: E/E/PS (Electrical/Electronic/Programmable System) Realisation*: Formal methods are strongest in the area of functional specification, design and implementation. In particular the use of programming languages with well-defined formal semantics and logical systems to allow reasoning about programs in such languages opens up a route to assurance through proving properties of the implemented system. More generally, formal models of the system and its software can be used to explore and motivate design decisions

7.1 *E/E/PES Safety Requirements Specification* Both functional and some safety requirements can be captured using formal notations. Having formalised specifications of requirements one can reason about them and check, for example, that the system is in principle implementable, or perhaps that in order to achieve some safety requirement one needs to add some assumption about the behaviour of the environment or system.

7.2 *Validation Planning* If one intends that proofs should play a role in validation one needs to leave sufficient time since this can be an arduous process and one also needs to ensure suitably trained staff are available.

7.3 *Design* Formal methods can support design. In particular in some formal methods, notably VDM and some algebraic approaches, a formally motivated design methodology, usually called refinement, has been developed. It seems [3] that some projects have used this approach for some parts of design but that used in full it can be too arduous. One can use formal methods to support design through the construction of abstract system models which can be used to help in comparing the merits of different design decisions.

7.4 *Verification* If a refinement approach fully adopted then with each refinement taken in the design process one generated proof obligations which amount to requirements for formal verification of the design decisions. In large scale developments very many such obligations are generated and it seems the approach taken is to identify those obligations which are seen to be particularly critical and discharge those obligations by carrying out a proof.

7.5 *Development of Operation and Maintenance Procedures* Use of formal methods in the specification of a system may result in the development of a particularly concise statement of the intended functions of the system. This in turn could have a significant impact on the structure of the operation and maintenance procedures. The requirement for formality may have significant impacts on the maintenance procedures to ensure the integrity of the system.

7.6 *Implementation* The biggest impact here is likely to be in the choice of programming language. Ideally one should choose a language whose semantics is formally defined and which has a formally verified compiler. There are no such compilers commercially available but a number of subsets of commonly used languages have formally defined semantics and compilers whose simplicity raises the level of confidence in their correctness.

7.7 *Integration of PES Hardware and Software* The approach taken with formal methods is to formally model the hardware as part of the design activity and validate the hardware model with the real hardware. This should ease integration problems.

7.8 *E/E/PES Safety Validation* Safety properties which are formalisable (e.g. that some action cannot be taken by a system, or that some action will always eventually follow another) can be proved (or disproved) to

5 The Exploitation of Formal Methods

5.1 Overall approach

This section considers the exploitation of formal methods and expands on the principle outlined in Section 3.5.1 for their successful use. The importance of the correct management approach and commitment can not be stressed too much. Just as in any endeavour, the uptake of new technologies need to be planned, resources secured and the project well managed. However the exploitation of formal methods requires a more systematic approach to exploitation than many other new techniques. This is for a variety of reasons but mainly reflects the need for a greater investment—education and training, tools, organisational—than is customary in the adoption of new workstations or programming languages. The investment can be significant for the following reasons:

- different technical skills
- changes to the system and software engineering processes
- changes to culture
- transition costs are larger than those typically borne by a single project
- time scales of adoption can be longer than a single project

The essence of the correct approach to formal methods exploitation is to understand the motivation for use of formal methods (see Section 5.2), to appreciate the implications for the

organisation and project (see Section 5.2.1), and manage the risks of using a new technology (see Section 5.3). For future company benefit and reference, the process by which decisions are made should be documented and become an internal deliverable on the project.

5.2 Reasons for using formal methods

Although it is rare to see this as an explicit document it is advisable to provide a rationale for the use of formal methods setting out the benefits, costs and risks. See Section 3.3 for possible benefits, limitations and costs of use of formal methods.

5.2.1 Implications for organisations

The contemplation of any change in the systems development process within an organisation is a serious issue, and justifies much thought and planning. This is especially true when considering a move from a structured development technique to a formal approach. The reasons for this are twofold:

The adoption of formal methods requires a change in development working practices. For instance, it is widely reported that the use of formal techniques leads to a longer specification phase. It is very tempting to try to curtail the specification phase because it is longer and more expensive than previous experience with structured methods suggests. However, the serious adoption of formal methods must be accompanied by a management commitment to the resultant change in life cycle in order that the full benefit of the method can be gained. This requires, for example, an acceptance that the specification phase will account for a higher proportion of the total development effort.

The successful use of formal methods requires a skill base which is different from that required for the successful use of structured techniques. A developer skilled in structured techniques needs such traits as clarity of thought and precision of expression, together with a knowledge of the development method and an appreciation of how it may be applied to large problems. To use formal techniques, the above skills must be supplemented by knowledge of aspects of discrete mathematics such as predicate calculus or set theory. Perhaps most important is a trait known as *mathematical sophistication*. This is not to say that a fully-trained mathematician is required, but an ability to apply mathematical techniques to solve problems, and to quickly grasp new mathematical ideas and notations. The difference between “readers” and writers should also be understood. To write formal specifications requires experience in depth, and this capability is only required in one or two members of a team, however the communication of formal concepts amongst project members requires a basic reading ability. This can be acquired at relatively low cost, but management does have to consider the question of motivation of staff who may be resistant to such a move.

The first of these issues can be addressed by ensuring that everyone involved in the development is aware that working practices will differ from those previously experienced, and appreciates the benefits of the change.

The second issue raises the question of how to obtain the required expertise. Broadly this can be done in two ways:

- By purchasing skills or forming a partnership with other companies, research organisations, or Universities with the required skills.

- By developing a new skills base within an organisation. This can be done either by recruiting new staff with the required skills, or by providing training for current employees.

When recruiting staff with formal methods skills, it is important to remember that reading a book on computer system design does not enable one to write a fly-by-wire system. Experience is at least as important as formal training, and so a new graduate with formal methods training will not possess all the skills required to work in an industrial situation. However, such a graduate could play a valuable role as an in-house formal methods expert, to work with a team of others who have had less formal methods training, but more experience in practical development. Without such an expert, an in-house team has no-where to turn to when technical problems occur. Also, formal proofs are much more efficiently conducted by someone with a mathematical background and training in this type of work. So the best approach is probably to gain a blend of highly-trained formal methods experts to work with experienced development practitioners who have received a basic training in formal techniques. Such training would preferably be in the form of a course which is tailored to the needs of the organisation, rather than a generic course, which may be aimed at the wrong level.

Formal methods cannot make up for deficiencies in the process organisations adopt to conduct the developments. While it is still a matter of debate how mature an organisation's process has to be to gain benefit from a particular method, it is unlikely to realise many benefits on industrial scale problems unless an appropriate QMS is in place (e.g. ISO-9000).

5.3 Manage the risks

There are a number ways of managing the risk of a new technology. An evolutionary path for adoption is important: do not run before you can walk (see Section 5.3.1). Also learn from others experience (see Section 5.3.2) both in getting the technology right and in addressing the important people factors. (Hopefully this guideline is a first step in learning from others experience). Also, the method has to be appropriate for the problem and the problem suitable for the method. Part III of the Guideline addresses this in detail and preliminary considerations are given below in Section 5.3.3. The choice of method is also influenced by the quality of tool support available and this is discussed in Section 5.3.4.

5.3.1 Evolutionary adoption

There are a number of ways in which formal methods can be applied in an evolutionary way:

Coverage of problem domain Choose the formal methods which can be applied to just a part of the problem where the greatest returns are anticipated.

Degree of rigour. Most formal methods can be used with a varying degree of rigour either by easing the amount of proof required and providing rigorous rather than fully formal proofs. (See Section 3.1 and [40] for more information on rigorous and formal arguments.) In addition they might be used just as specification languages or, for model based approaches like VDM, they might only be applied to describe the state of the system rather than the functions or operations associated with it.

Extend current notations The formal method could be combined with existing structured method to elaborate and underpin parts of the structured method e.g. to provide a specification of the data transformations in a dataflow diagram.

Underwrite existing notations. Formal methods can be used to define existing notations and some of their benefits can be realised indirectly.

Provide more powerful notations. Most programming languages currently in widespread use lack a fully mathematically rigorous semantic definition. That is, the behaviour of a particular program is left open to the interpretation of the implementer of the language and the vagaries of the operating environment. Recently we have begun to see the emergence of languages which are fully defined. Examples are Standard ML [36] developed at Edinburgh University; the object oriented language Amber [13] developed in DEC and the Eiffel object oriented language. Such languages not only have the benefits of unambiguous execution models, but also offer powerful new programming structures which aid the specification and construction of software. Ultimately the existence of a formal semantic will also allow the verification of program behaviour against specifications.

In practice the evolutionary adoption will involve many of these aspects.

5.3.2 Learn from others

Lastly the prospect of successful application of formal methods is significantly increased if you can copy what others have done. Without a significant body of examples to build upon, the application of formal methods remains experimental. Only when a significant body of examples have been done and documented can a more engineering approach to the use of formal methods be made. Although case study material is beginning to appear, the area is not fashionable, academics by nature tending to prefer the further development of fundamental ideas and industrial groups are not prepared to take the business risk of experimenting with new ideas, or using formal methods in only simple ways which are well understood. Effective case study development needs appropriate and timely interaction between both academic and industrial groups. This probably need strategic intervention and support for national or international funding agencies—time scales and risks are too great for industry to see an acceptable return on investment in a reasonable time scale. The techniques directory (Part III) indicates both known published case study material and groups and consortia which are active in experimenting or exploiting a particular approach.

There is a growing literature on the uptake and impact of formal methods on industrial system development. In the UK the National Physical Laboratory has sponsored a survey of the uptake and use of formal methods in the UK [3]. In the US the NIST has commissioned an international survey of the use of formal methods in industrial applications [19], this provides an in-depth view of the use of formal methods across 12 projects.

5.3.3 Appropriateness of method/problem

The selection of an appropriate method for the problems of interest is vital to the success of the project. This is not a straightforward choice as problems and methods have a number of different strengths and weaknesses and indeed one of the strong messages from this document is that there is no single method for all your needs. The strengths and weaknesses do not just concern technical issues relating say to the theoretical bases of the method but to equally important aspects such as their maturity.

Part III is a directory of formal methods providing a description of each method and a brief appraisal on the following points:

- the maturity of the a method e.g. the availability of textbooks standardization activities, user base.

- the availability of study material, training courses, and consultancy on the use of the method.

- the strength of the method, describing areas of application in which the method has a good record of application and a description of the kinds of properties which can be formalised within the method.

- the level of Industrial Experience which has been gained with the method.

- the availability of tools to support the method.

5.3.4 Tools

Increasingly tool support is being seen as essential in the successful industrial use of formal methods. This is probably true, but care must be taken to get tools and the need for them in the right context. Much success in formal methods has been conducted without tool support. However this has often been done by academic groups, with significant scientific understanding. The case studies thus conducted are usually either so simple as to be irrelevant to; or represent a considerable abstraction from the real world. Where detailed and realistic problems have been tackled many errors remain in the published analysis. Doing formal methods without tools is like writing programs without a compiler.

Commercially supported usable tools are beginning to become available—see the techniques directory in Part III.

As indicated in the preamble to this section, tools can do a number of things. At their simplest, they can provide support for syntactic and semantic checking of expressions in the language of the method. This can eliminate many errors and support consistency checking. Availability of tool support at this level should probably be a prerequisite for deciding on the use of a particular method.

At the next level comes support for manipulating expressions in the language and in doing related proofs. This requires much greater insight, and by and large it is still a subject of academic study. However in a limited number of areas some success is apparent and we can hope for major advances in industrial grade systems in the coming years.

In the past there has been considerable activity aimed at fully automating the proof aspects of formal methods. It is however clear now that although there have been some success (for example in the area of induction) is it unlikely that generalised automatic proof tools for formal methods will emerge in the short term. As in other engineering disciplines the activity of using formal approaches to the design and analysis of computer systems requires a high level of engineering insight. The main thrust in proof tools is now to provide a *proof assistant* which structures the proof in such a way that the mathematical (and hopefully engineering) insight of the human “operator” guides the system. Conversely computer based tools are good at the basic housekeeping tasks such as checking the consistency, completeness or correctness of the proof. Again a close analogy with programming languages can be made: compilers provide strong

support for syntax and (limited) semantic checking of programs, but it still requires engineering insight to develop "good" programs.

Some tools provide support for graphical interfaces for animation of the system. These have the potential for supporting the exploration and understanding of what for real systems will almost certainly be an extremely complicated set of equations or formulae in the unstructured text which is the normal underlying representation. However, despite being very good for giving impressive demonstrations, when used on real problems these system can often isolate the knowledgeable user from much of the power of the system which is available using basic command-line interfaces.

It is important to realise the limitation of tool support. Tools do not replace the process of thinking about design. The aspirations of the mid '80s to develop "black box" tools which would automatically produce "correct" code from specifications have not been realised. The process of writing good programs was then and remains, primarily a creative process backed up by engineering judgement. If anything the use of tools and formal methods will require even greater skill, understanding and creativity amongst designers.

Because of their nature, tools will remain expensive. Commercially supported tools are expensive to buy. Academic tools (although cheap or free) are not normally developed to the level required for industrial deployment and the staff training overhead can be considerable. They tend not to have guaranteed support and because they are usually research based they will "improve" from year to year changing syntax and features. The introduction of standards for formal methods should go some way to addressing this problem however.

6 References

[1] http://www.itsec.gov.uk/—all the ITSEC documents are downloadable from this site

[2] R. Milner, Communication and concurrency, Prentice-Hall, London, ISBN 0-13-115007-3, 1989.

[3] S. Austin and G.I. Parkin. Formal methods: A survey. National Physical Laboratory, Queens Road, Teddington, Middlesex TW11 0LW, March 1993.

[4] J. Barnes, High Integrity Ada: The SPARK Approah, Addison-Wesley, Longman, London, 1995

[5] L.M. Barroca and J.A. McDermid. Formal methods: Use and relevance for the development of safety-critical systems. *The Computer Journal*, 35(6):579-599, December 1992.

[6] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. Information and Computation, 60:109-137, 1984.

[7] W. Bevier. Kit and the short stack. Journal of Automated reasoning, 5(4), 1989.

[8] R.E. Bloomfield, J.H. Cheng, and J. Gorski. Towards a common safety description model. In J.F. Lindeberg, editor, SAFECOMP '91, 1991.

[9] T. Bolognesi and E. Brinksma. Introduction to the specification language LOTOS. In van Eijk, Vissars, and Diaz, editors, The Formal Description Technique LOTOS. Elsevier, 1989.

[10] R.S. Boyer and Y. Yuan. Automated proofs of object code for a widely used microprocessor. Technical report, Computer Science and Mathematics Departments, U. Texas at Austin, 1989.

[11] Bramson, B. D. Malverns Program Analysers. 1985. RSRE Research Review

[12] G. Bruns and S. Anderson. Validating safety models with fault trees. In SAFECOMP 93 Proceedings of the 12th International Conference on Computer Safety, Reliability, and Security Springer, 1993.

[13] L. Cardelli. Typeful programming. Technical Report 45, DEC SRC, May 1989.

[14] Carre, Bernard. Program Analysis and Verification. in: Sennett, Chris. High-Integrity Software. London: Pitman; 1989: 176-195

[15] B.A. Carre and C.W. Debney. Spade-pascal, version 4.0. Copyright Program Validation Limited.

[16] CEU. European Harmonized Security Evaluation Criteria, 1991.

[17] B.P. Collins, J.E. Nicholls, and I.H. Sorensen. Introducing formal methods: the CICS experience with Z. In B. Neumann et al., editors, Mathematical Structures for Software Engineering. Oxford University Press, 1991.

[18] D. Craigen, S. Gerhard, and T.J. Ralston. Formal methods reality check: Industrial usage. In Formal Methods Europe Symposium (FME'93). Springer-Verlag, LNCS, 1993.

[19] D. Craigen, S. Gerhart, and T.J. Ralston. An international survey of industrial applications of formal methods. Technical report, US National Institute of Standards and Technology, 1993. To appear.

[20] Felix Redmill (editor). Dependability of Critical Computing Systems:1. Elsevier, 1989.

[21] Felix Redmill (editor). Dependability of Critical Computing Systems:2. Elsevier, 1991.

[22] Peter Bishop (editor). Dependability of Critical Computing Systems:3 -Techniques Directory. Elsevier, 1992.

[23] Christopher Gerrard et al. Obj—an introduction. Technical report, Gerrard Software Ltd, UK, 1993.

[24] M. Francis, S. Finn, E. Mayger and R.B. Huges. Reference manual for the lambda system, version 4.2. Copyright Abstract Hardware Limited, 1993.

[25] T. Gilb. Principles of Software Engineering. Addison-Wesley, Wokingham, England, 1988.

[26] M. Gordon. HOL—a machine oriented formulation of higher order logic. Technical report 68, University of Cambridge, Computer Laboratory, 1985.

[27] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. Proceedings of the IEEE, 79(9): 1305—1320, September 1991.

[28] C.A.R. Hoare. Communicating Sequential Processes. Prentice Hall International, 1985.

[29] IEC 61508-1 Functional safety of electrical/electronic/programmable electronic safety-related systems-Part 1: General requirements

IEC 61508-2 Functional safety of electrical/electronic/programmable electronic safety-related systems-Part 2: Requirements for electrical/electronic/programmable electronic systems,

[30] IEC 61508-3 Functional safety of electrical/electronic/programmable electronic safety-related systems—Part 3: Software requirements

[31] Defence Standard 00-56, Issue 2, Ministry of Defence, Directorate of Standardization, Kentigern House, 65 Brown Street, Glasgow G2 8EX, UK Safety Management Requirements for Defence Systems, 1996.

[32] D. MacKenzie. Computer-related accidental death: An empirical exploration. Science and Public Policy, 21 (1994), 233-48.

[33] B. Meyer. Object-Oriented Software Construction. Prentice-Hall, London, 1988.

[34] B. Meyer. Eiffel: The Language Prentice-Hall, London, 1991

[35] Robin Milner and Mads Tofte. Commentary on Standard ML. MIT Press, 1991.

[36] Robin Milner, Mads Tofte, and Robert Harper. The Definition of Standard ML. MIT Press, 1990.

[37] Ian M. O'Neill. Notes on theorem checking and a guide to the spade proof checker. Technical report, University of Southampton, December 1988.

[38] I.M. O'Neill, D.L. Clutterbuck, P.F. Farrow, P.G. Summers, and W.C. Dolman. The formal verification of safety-critical assembly code. In Proceedings of Safecomp '88. Pergammon Press, November 1988.

[39] Ontario Hydro, 700 University Avenue, Toronto, Ontario M Standard for Software Engineering of Safety Critical Software, doc. 982 c-h 69002-0001, rev. 0 edition, December 1990.

[40] (Part 1: Requirements, Part 2: Guidance). Defence Standard 00-55, Issue 2, Ministry of Defence, Directorate of Standardisation Kentigern House, 65 Brown Street, Glasgow G2 8EX, UK. Requirements for Safety Related Software in Defence Equipment, 1997.

[41] Radio Technical Commission of America. Software Considerations in airborne systems and equipment, RTCA/DO178B, 1993.

[42] W. Reisig. Petri Nets. EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1985.

[43] A.W. Roscoe. A denotational semantics for occam. In S.D. Brookes, A.W. Roscoe and G. Winskel, editors, Seminar on Concurrency, number 197 in LNCS, pages 306-329, Berlin, July 1984. Carnegie Mellon University, Springer Verlag.