

Towards Industrially Applicable Formal Methods: Three Small Steps, and One Giant Leap

John McDermid, Andy Galloway,
Simon Burton, John Clark, Ian Toyn, Nigel Tracey, Sam Valentine
High Integrity Systems Engineering,
Department of Computer Science,
University of York, UK.
(e-mail: {jam,andyg,burton,jac,ian,njt,sam}@cs.york.ac.uk.)

Abstract

In this paper we discuss issues in the development of formal methods for use in aerospace applications, reflecting our experience in working with both Rolls-Royce and British Aerospace. We discuss some of the key factors which we believe govern the application of discrete mathematics to aerospace applications, drawing comparisons with applied engineering mathematics in other domains. We give an overview of three projects (the three “small steps”):

- *The development of a domain-specific language for aircraft engine control system specification;*
- *The development of a formal semantics and tool support for state transition systems to facilitate analysis of specifications produced by systems engineers;*
- *The use of formalisms in support of test automation.*

We then discuss the “gap” we see between the needs of industry and the current focus of the formal methods research community by pointing out important facets of industrially applicable formal methods which are not receiving adequate attention. We refer to this as a “giant leap” due to the need for a cultural shift in the research community, and the need for a coherent approach to the identified research issues rather than piecemeal studies of the issues. Our conclusions are to be optimistic for the future use of formal methods in industry, albeit with concern that their potential will not be realised unless there is a shift in emphasis within the research community.

1 Introduction

The idea of using mathematics to specify and analyse programs has been with us for a long time – arguably from

the earliest days of digital computers around 50 years ago, or even from the days of Babbage. The term formal methods has been coined relatively recently. However the work of Floyd [5], Hoare [16] etc., which most would accept initiated the development of formal methods, is now about 30 years old. It is generally recognised that new ideas in engineering take in the range of 14-18 years to develop from initial concept to industrial maturity and into widespread use. In many cases, e.g. structured methods and CASE tools, this “time constant” appears to apply to computing (software engineering) as well as to other disciplines. On this criterion, formal methods should have been in widespread use for over a decade – but, despite a few significant success stories e.g. [2, 15, 19], they are not widely used. This should prompt the optimist to enquire “why haven’t they become more widely accepted?”, but it may prompt the pessimist to wonder if they will ever become widely used. The purpose of this paper is to investigate these questions – albeit from the standpoint of the optimist.

1.1 Systems of Interest

We consider the above questions from the point of view of the use of mathematics in other engineering disciplines, and our experience of trying to apply formal methods to complex computer-based systems, especially in avionics. Systems we have studied include aircraft engine controllers and flight control systems. These are real-time, safety-critical, fault-tolerant computer systems embedded in complex engineering products. Design and analysis of such systems pose many technical challenges, however they are just the sort of application for which the rigour of formal methods ought to be of benefit.

The technical characteristics of such systems which are relevant to this paper are:

- There is a fixed number of tasks;

- The computer equipment may be physically distributed in different places;
- There may be concurrent operation;
- Appropriate responses to external events are required within rigid time-limits;
- The cost of failure is very high therefore there are often stringent requirements on failure rates, e.g. the oft-quoted 10^{-9} failures/hour.

These characteristics influence the three projects we describe below, and present several challenges for formal methods. We believe that the difficulties we have encountered in applying formal methods in these domains is salutary, and indicative of changes in research emphasis which need to be adopted by the formal methods community.

1.2 Applied Engineering, Mathematics and Software

The following three factors seem to us to be some of the most significant in distinguishing formal methods from applied engineering mathematics in other domains:

- Domain specific notations and analyses – in other engineering disciplines the mathematics is adapted for the domain, and for analysis. For example, different mathematical models of transistor behaviour are used for radio frequency applications as opposed to use at modest frequencies, e.g. in audio circuits. Also, the mathematics is used to facilitate analysis of designs, and is rarely used for purely descriptive purposes. These notations give the mathematics considerable power and utility.

Most formal methods, e.g. Z [37] and CSP [17], are very general, and are perhaps best viewed as theories rather than methods. We believe that there is a need for domain specific notations to facilitate system specification and effective analysis;

- Covert mathematics – in other engineering disciplines the mathematics is hidden inside CAD tools. For example tools such as MatrixX enable control laws to be specified and analysed. More generally there are various finite element modeling tools which are used for stress analysis, aerodynamic modeling, thermal modeling, etc. These tools enable their users to design and analyse systems without the need for deep understanding of the underlying theories. With formal methods the mathematics is very overt, and often this is seen to be an impediment to the adoption of the techniques. Further, few tools support formal

analysis, except at a fairly shallow level, e.g. type-checking, without extensive user involvement. Formal analysis, at the level of proofs, is a high-skill activity which is the province of specialists. There is therefore a need for providing formal underpinnings to methods and notations acceptable to engineers, and to provide as much automated support for specification analysis as possible, including specification-based test automation;

- Analysis not construction – in other engineering disciplines the mathematics is used for analysis and checking of designs, but not for the construction of systems. This enables engineers to use their creativity to propose designs, and the analyses can be used to assess their effectiveness. This is particularly important where we are concerned with different facets of design, e.g. behaviour, mass, power consumption, and we need to propose a design which represents an acceptable trade-off between all these facets.

The above issues are related, and they cannot be addressed “in isolation”. The “small steps” we discuss in section 2 do not correspond directly to the above goals - indeed they all contribute to at least two of them. However we prefer to state these broad goals, rather than the specific aims of the three projects alluded to below, as they are more general. Further, they draw out more clearly what is necessary to establish formal methods that have the capabilities of conventional applied engineering mathematics.

It should be acknowledged that it seems to be harder to achieve these objectives for formal methods than it is in many more traditional engineering application domains. For example, in many cases there are closed form solutions to equations. In other cases, e.g. finite element analysis, there are approximations or iterative algorithms which can be used to obtain satisfactory solutions to problems which are not analytically or computationally tractable. There are analogies for formal methods, e.g. heuristics in model checking and tactics in theorem proving, but they do not seem to be so powerful as their counterparts in conventional engineering. In part this is due to the relative immaturity of software engineering, but also reflects the fact that computers are very flexible, and can be applied to many domains – where we need a new domain theory. In other disciplines, it is relatively rare to need to develop new domain theories, but not unprecedented – e.g. new techniques were needed for the analysis of curved reinforced concrete bridges.

1.3 Overview of the Paper

The above discussion is intended simply to “set the scene” for section 2 where we outline what we are doing to address those issues (the three small steps). In section 3

we identify a number of issues which seem to be particular to the use of formal methods of software development, and which we believe are not receiving adequate attention. We believe that these factors need to be addressed to enable the “giant leap” to bring formal methods into widespread use, at least for real-time embedded systems. Finally, in section 4, we draw some conclusions including our prognosis for the development of applied formal methods.

2 Three Small Steps

2.1 Developing Domain Specific Formal Notations for Engine Control Software

The Practical Formal Specification (PFS) project, funded by the UK Ministry of Defence, aims to provide the technology for formal development of engine control software. The project is supported by the Rolls-Royce Aeroengines group who, as a supplier to the Ministry of Defence, are required to use (or mandate of their subcontractors) formal techniques for the safety critical components of the software they provide [44].

Although some departments of Rolls-Royce have investigated the use of discrete formal methods (such as VDM [23]) in the past, such techniques are still not, in general, used in the development process for engine control software. Indeed, their use is still very much an area for research and development. To understand the reasons why formal techniques are not in more widespread use it is important to appreciate the domain in terms of both the process and the product it produces. Control software is developed within a multi-level concurrent engineering process where airframe requirements give rise to propulsion system level requirements, which in turn give rise to control system requirements. The control system itself is made up of many engineered components, including sensors, actuators, hydraulics, digital hardware and software. Software requirements hail from, and are affected by, the requirements and design of all the components in the engineering hierarchy.

Because of the complexity of the environment in which the software will eventually be placed software requirements are the product of dialogue between experts in different disciplines. Experts in control theory model the sensors and actuators and engine dynamics at different design points, and design transfer functions which when implemented in the software will allow an engine to meet the performance requirements whilst respecting safe engineering limits. Different engine situations (e.g. starting, normal or advanced control) often require different expertise. Other experts consider how to condition sensory information to remove noise and detect evidence of failures. A related safety and certification process prompts designers to introduce processor (and sensor) redundancy and engineers must

consider the situations under which control passes from one configuration to another.

The engineering process is multi-staged with design information, including software, being ‘base-lined’ at points in the overall development. Design changes often give rise to extra or amended requirements in the lower level engineering levels. Software requirements and software, at the lowest level, have to evolve.

The nature of the engineering process suggests that in order to be applicable, formal techniques will need to be robust to changes in requirements – it would not be economically viable to provide formal proof of a complete system every time the requirements are revised. Reusable, formally verified components and the use of reasoning at various levels of rigour (and hence cost) on a risk-of-change basis are both seen as ways of overcoming these problems. Compositionality and monotonic reasoning are also very important.

The engineering process needs to be foremost in mind. However, there are also problems with formal specification and verification even if one takes a static view of the software requirements and their subsequent design.

2.1.1 Formal Specification and Verification of Engine Control Software

Taking a static view of engine requirements, the most widespread view of a formal development (leading to sequential source code), is that it should begin with an abstract specification of ‘what’ is required of a system. Most methods then proceed by adding successive levels of detail about ‘how’ requirements will be implemented in practice. At each stage the more detailed level is shown consistent, with respect to a refinement relation, with the more abstract level. Indeed, this view of formal development is strongly advocated within Defence Standard 00-55.

Our experience with the PFS study has highlighted three main obstacles to the adoption of this kind of formal development. These are the inapplicability of formal abstraction mechanisms, the difficulty of eliciting and validating formal specifications, and the narrow viewpoint of the established formalisms. We will continue by discussing each point in more detail.

Inapplicability of Formal Abstraction Mechanisms

Abstraction is a vital consideration. Formal development aims to achieve higher integrity than more conventional software engineering approaches. However, the development process is inductive and unless one starts with a specification of requirements which is *valid* to a comparable order of integrity, the use of formal *verification* is not the significant factor in the integrity of the final software. Since *validation* is essentially a human activity (for which formal

and informal support exists) it is essential that initial statements of requirements are *abstract* and comprehensible.

In model-based methods (such as Z, VDM, B [1], Refinement Calculus [30]), which represent the most mature formalisms for the development of sequential source code, the main levers for abstraction are:

- Data Abstraction – specifying behaviour in terms of sets, relations etc. which are not directly implementable.
- Implicit Specification – specifying the relationship between before and after states, inputs and outputs without stating algorithms for how to compute them.
- Loose Specification – specifying a range of acceptable outcomes for a particular situation.
- Preconditioning – only prescribing behaviour for a subset of all the possible situations and not placing any constraints on behaviour in others.

In engine control software, the core requirements hail from continuous mathematical modeling (Rolls-Royce use the RRAP model) of the engine and actuator dynamics. The engineers use tools (such as MatLab, MatixX) to solve differential equations (transfer functions) and provide parameters for PID (proportional, integral, differential) compensators. The core requirements describe how sensory input should be mapped to actuator output over time to achieve the required engine dynamics. Once a set of compensators have been defined they are combined using selection logic that describes when the product of each compensator should be in control of the actuators. Input and output conditioning requirements are also necessary to validate and preprocess input signals and to convert demanded responses into actuator signals. Finally, fault detection and failure management requirements alter signal conditioning, and selection logic, providing gracefully degraded service as parts of the environment fail.

The software requirements, in general, do not involve the storage and maintenance of complex information structures – only integers – so data abstraction does not appear a widely applicable form of abstraction. Requirements are tightly specified, in the sense that only one outcome is stipulated for every situation. Also, because the controlled environment is understood in terms of the great many relationships between physical (and conceptual) quantities, the expression of requirements is also very explicit. Requirements are presented as a pipeline of low-level components (scalers, switches, adders, hystereses, differentiators, integrators) which propagate and transform sensory inputs through a myriad of derived values into actuator outputs. Of the low-level components, many are straightforward enough

to implement directly (although others are more applicable to abstraction in the usual formal sense e.g. constant function look-up). Knowing the dependencies between the values required to be computed (and thus their sequential ordering), and finding the relationships as explicit and low-level, at least the first three abstractions are very difficult to apply.

At the very least, where these kinds of abstractions do apply (e.g. the relationship between engine parameters, the engine performance requirements) discrete formal reasoning is intractable. Instead, continuous mathematics provides for formal development of control ‘algorithms’ which are high-integrity when used in their intended context.

Difficulty of Employing Formal Specification Just as important as abstraction for the comprehensibility of a requirements specification, and thus its validity, is the language in which the ideas are expressed. There has been much written in recent years e.g. [6, 35] about the drawbacks of using formal notations (such as those based on set-theory) to describe intended behaviour. The main points of this argument have been:

- Difficulty of eliciting and validating information presented in formal notations – Formal notations are often highly expressive (often one is only limited by the bounds of what one can describe in set-theory and predicate logic). When a piece of reasoning has been described in a formal notation by appealing to axioms and inference rules the results are, in general, machine checkable. However, possibly as a product of their expressibility and their suitability for reduction to syntactic rules, it is not uncommon even in communication between experts for mistakes to be made (remember we are talking about high levels of integrity, 10^{-9} !) [45]. Often, as with engine control software, the practitioners who describe requirements are experts in other disciplines and in the majority of cases have little experience in discrete formal notations.
- Process change to Formal Notations is too revolutionary – A possible solution to the problem of eliciting and validating requirements in formal notations would be to train domain experts in formal methods. However, our experience has shown that it is difficult to motivate experts (who are already extremely valuable) in one discipline to spend considerable time gaining expertise in another. The investment risk involved in gaining new expertise, plus the perception that the formal development is very different from a conventional development process, makes management less inclined to support a change in working practice. This can be exacerbated by the absence of a

clear process model and lack of convincing industrial tool support for formal notations.

Narrow Viewpoint of Established Formalisms Of the mature formalisms, different classes of method concentrate on different facets of computation. For instance, model-based notations (Z, VDM, B etc) are most commonly used to reason about the way information is stored and stepwise manipulated in sequential machines; process algebras (e.g. CSP, CCS [29], LOTOS [20]) are most commonly used to reason about the patterns of communication across a distributed topology; temporal logics (e.g. RTL [21], TRIO [9], mu-calculus [38]) are most commonly used to describe how actions and states are temporally ordered as the system evolves through time.

Engine control software is multi-faceted, there are important timing properties associated with the stability of the control laws, implementations may be distributed across several processors, and each processor must ultimately process sequential code. No one established formalism seems applicable to the *whole* domain. This prompts questions concerning how we use different formalisms together and how we ensure consistency across formalisms.

2.1.2 Domain Specific Languages

The PFS approach to solving the above problems has been to construct a Domain Specific Language. By Domain Specific Language (DSL) we mean a language with a syntax ideally tailored to expressing the basic concepts and structuring mechanisms of a particular application area. Thus, rather than being limited only by the bounds of what one can describe in, say, set theory, the practitioner is limited by the kinds of requirements concepts and structures of control software. The most important features of a DSL are:

- An intuitive concrete representation – The concepts and structures of the language, when represented in their concrete form, must be intuitive, unambiguous and easy to navigate. This may involve use of an intuitive lexicon, diagrams, tables or templates to represent the basic concepts. In PFS, the domain specific language is based heavily on hierarchical state machines (such as Statecharts [10]) and tabular notations such as those employed in SCR ('Software Cost Reduction') e.g. [13]. The DSL organises requirements as state-based components (which maintain behavioural or information state), reactive components (which have no state¹) and aggregations thereof.

¹In fact, because differentials are so important within control requirements the notion of 'stateless' is blurred to permit reference to immediately prior values.

- A layered formal semantic construction – In order for the domain specific information to be amenable to formal reasoning, the language must be given a formal semantics. For clarity (and therefore confidence in the semantics) this may involve several layers. Concrete representations (diagrams etc) are related to abstract representations (grammars, relations etc). Abstract representations may be subject to static semantic constraints. Language concepts may be related to more primitive 'core' language primitives. Finally core language primitives are interpreted as objects within a semantic mathematical system. In PFS, the formal semantics is inspired by work on the formalisation of graphical notations (see section 2.2).
- A set of reasoning goals – Given a formal semantics, proof obligations can be formulated. Proof obligations may be generated for model "healthiness" thus increasing confidence in the validity of a specification. Additional reasoning obligations may be generated in the light of a design artifact to demonstrate compliance to a specification given appropriate refinement relationships. Pragmatically, reasoning goals can then be handed to experts in formal reasoning.

Domain Specific Abstraction Domain specific notations permit domain specific abstraction. By constraining the domain of application it is possible to be *more* specific about what forms of abstraction are applicable. Firstly, given the relative importance or criticality of the functional components that make up an application in a particular domain, a specifier can employ *structural abstraction* to manage the introduction of detail. For example, in PFS the emphasis is on specifying core control requirements and working 'outwards', through signal validation and failure management, toward the sensor and actuator interfaces.

Additional abstractions can be provided within the notation itself. In PFS the use of hierarchical state machines permits the specifier to intuitively describe the desired behaviour, rather than stating behaviour in terms of the control variables that will eventually be used in the implementation. Also, the PFS approach focuses on the discrete aspects of engine control requirements. The notation prompts the specifier to provide 'place holder' information for the transformations being developed using continuous mathematics. Such requirements will be validated independently by their own mathematically 'formal' method.

A major consideration is how different levels of abstraction can be related formally. Again this can be domain specific, although PFS relies heavily on the fourth abstraction mechanism described above: stating assumptions and specifying within those assumptions. The emphasis is on explic-

itly stating the assumptions that a discrete requirement (or placeholder for a ‘continuous’ requirement) expects in its context. Assumptions might concern the range of inputs to a requirement, the rate of change or the expected relationship between inputs (especially with respect to arithmetic saturation etc.). The process of validating a set of requirements is formally supported by composing requirements together to form aggregates and propagating assumptions outward to the environment. Formal composition is based on the concepts of *guarding* and *governing* of requirements. Of these *governing* is the most straightforward – analogous to the concept of weakest precondition derivation in programming language semantics. *Guarding* takes into account requirements independence, that is, the conditions under which a requirement will not affect the observable behaviour of the software of which it is a part. Composition of requirements amounts to deriving the weakest precondition over a set of inputs to guarantee that, where a requirement affects the observable behaviour of the software, its assumptions are guaranteed by its context.

In general, given requirement composition in terms of *guarding* and *governing* the usual ‘weakened precondition’ refinement relation holds. The relationship supports ‘ideal’ to ‘real’ refinement in PFS. When an engineer initially describes a requirement, they assume an ideal context, e.g. no noise on signals, no failures, hardware responds as a quickly as software, signals will not induce arithmetic saturation. Such simplifications result in strong assumptions being documented for each requirement—assumptions that could not usually be guaranteed by the environment. The engineer subsequently has several ‘refinement’ courses. The assumptions can be systematically weakened, and behaviour added to deal with the additional situations. Alternatively, additional behaviour may be introduced to *guard* and *govern* the requirement, and thus guarantee its assumptions.

Elicitation and Validation The domain specific language overcomes the perceived problems associated with eliciting and validating requirements in formal notations. The engineers can communicate their ideas within an intuitively appealing language. The formal semantics for the notation, together with ‘healthiness’ (and other) proof obligations renders the elicited information amenable to formal reasoning. Effectively, rather than developing requirements in a formal notation for every product, much of the formal specification effort has been expended ‘up front’ by specifying a whole class of products. The engineers then instantiate the generic specification by the information they provide in the DSL. The remaining effort, the formal reasoning, may then be undertaken by an expert in formal proof.

The idea of domain experts working in an intuitive language and then handing the fruits of their labour over to a proof expert who needs to know nothing of engine con-

trol seems utopian. In fact, the separation of concerns can never be so clear cut. The specifiers must engineer the requirements so that validity proofs succeed. This means that they inevitably have to think in a formal way (albeit without the need to learn new mathematical notation). Experts in proof need to be able to feedback reasons why proofs have failed, this is undoubtedly easier with domain knowledge. It is hoped that this kind of synergy will help bring together hitherto disparate disciplines, and contribute to an evolutionary change in working practise toward the use of even more formalism.

2.1.3 Progress and the Future

The PFS project has made steady progress. After spending a considerable amount of time understanding the domain, a hypothesis was developed which iteratively evolved into the strategy outlined above. The requirements notation has recently been applied to a case study based on the engine starting requirements of a helicopter engine controller. The use of the notation with formal analysis will be the subject of a forthcoming case study involving flight control requirements. Some intermediate results from PFS were published at DCCS ‘98 [8].

The techniques are still under development. Work is continuing on the problem of how to best decompose software requirements ‘in the large’ into a hierarchy of sub-components for which the proposed approach (especially the outward propagation of assumptions) will add the most value. This work is being supported by domain modeling and reuse work being carried out both within HISE and Praxis Critical Systems. In addition, some of the formal underpinning remains under development.

A second PFS project is planned (due to start in April 1999) which will attempt to ameliorate some of the problems associated with the ‘narrow viewpoint’ of established formalisms. The proposed backbone of the second project is to extend the notation and formal underpinning provided under the first project to support the specification of requirements for (asynchronous) distributed systems. The technology produced is intended to be applicable at two levels. It will permit the specification and validation of distributed control systems (which are becoming more and more attractive e.g. for weight, fault tolerance). Also, it will permit the validation of concurrently integrated systems (such as integrated modular avionics solutions to the problem of controlling astable flight) whose components (or interfaces) have been specified using a PFS-style approach. The envisaged work will build upon research on integrating process algebras with model-based formalisms (e.g. [7, 39]) as well as research on symbolic model checking [14, 34]. The focus of the work will be on outward propagating assumptions within asynchronous communicating systems, demonstrat-

ing equivalence and verifying temporal properties specified in the μ -calculus.

2.2 Giving Precise Meaning to Graphical Notations

The BAe-funded Dependable Computing Systems Centre (DCSC) is addressing a broad range of problems in dependable computing applications. One strand of work is concerned with applied formal methods, and enabling systems engineers to use familiar notations but to underpin the notations with formal analysis. The focus of the work has been on graphical specification languages addressing the problem of the interactive and concurrent systems discussed here. They support the notions of a “state” which some “activity” is in, of “transitions” between those states, and of “events” which trigger those transitions, as well as allowing the description of conventional computer data and computations on that data, where the computations are linked to the states or the transitions.

Two such graphical notations are i-Logix’s Statemate, [10], [11], which is in use at a number of sites in British Aerospace, and ADL, [33], an experimental notation being developed there. For brevity we focus on the work with Statemate and Statecharts.

With Statecharts, the states are shown graphically by boxes of some shape, and the transitions by lines joining the boxes. The associated computations are described using textual annotations, written either in a fairly conventional computer language, in the case of Statemate, or in a formal specification notation, in the case of ADL.

For safety-critical systems there is often a requirement for proof that intended properties hold [44]. Current practice is to carry out such proofs as are necessary by whatever means are possible, usually falling short of full formality.

An ideal solution might be to use computers to do the proofs, which requires at least the following to have been achieved, formally and correctly:

- a) a statement of what the graphical notation and its associated text means;
- b) a description of the system design which has been expressed in it;
- c) a means of expressing desired properties;
- d) a means to verify the truth of those properties for the design described.

Work at the DCSC has been addressing these problems for several years. The overall aim is to enable, as far as practical, systems and software engineers to work in familiar graphical notations, but to be able to automate as much formal analysis as possible, i.e. to see how close we can get to the above ideal.

2.2.1 What the Notations Mean

Two complementary approaches have been pursued:

- a) an operational description;
- b) a description using RTL [21], [22].

The operational description has a structure which broadly corresponds with that of an animation tool for the notation. Thus there is a description of the particular system design under consideration, and a representation of the instantaneous status of the whole environment, including which state(s) each activity is in, which events have just occurred, and the values of all data. To this is added a description of the “step”, by which progress is made to the next state, at some later time.

Whereas the operational description works by using “snapshots”, the RTL description focusses on the history of particular events. The RTL description takes the form that a particular occurrence of a particular event takes place at a particular time. Entry to and exit from states count as events in this formalism.

An advantage of the operational description is that, being close to the animation tool in its structure, it can be verified as correct in terms of that tool. This is particularly relevant where, as with Statemate, the behaviour of the tool is the only real definition of the semantics of the notation.

The advantage of the RTL description is that it is formulated in a way which more directly corresponds to properties of probable interest, and so it may be easier to prove those properties.

An operational description of the “step”, the primitive execution step of a state machine, for a large subset of Statecharts has been written in Z [37], [40].

2.2.2 Describing the Specification

The Statemate tool allows its statecharts, and other graphical data, to be entered directly using a mouse-driven interface. Supplementary information, such as all names and the description of triggers and actions, is entered using the keyboard and recorded as text. Further information about the data environment is entered textually into the “data dictionary”. The whole of this information is referred to as the “System Under Development”, the SUD.

The Statemate tool provides an “Application Program Interface”, (API), which allows direct access to the internal form of the SUD. An interfacing tool, called ChartZ, has been written which takes this internal form and directly generates a description of the SUD in Z. The execution of the “step” can then be studied directly in terms of this description and that of the Z operational description of Statemate.

To use the RTL theory, it is necessary to recast the information about the SUD in terms of its occurrence relations.

This can be done manually, which is tractable but not reliable, or using the formal mechanisms developed, which will be much more reliable once completed.

Whichever approach is made to proof work, the process is greatly complicated if the data environment is too rich. In particular, before proving properties of interest it is essential to show that there are no “race” conditions, where there is no control over the relative timing of a “write” to some data item and another “read” or “write” to that same item.

This is an example of a general rule, that a disproportionate amount of proof effort can often be spent on apparently very improbable contingencies. This effect can be reduced if the notation is well designed, or if troublesome special cases can be dealt with by some rule of the syntax or the static semantics.

2.2.3 Formulating Desired Properties

At present, properties of interest are formulated using the same formal notation as that used to express the formal underpinning (i.e. Z or RTL depending upon the approach). In future it would be desirable to formulate such properties using graphical notations akin to those used to model the system. This is seen as an area of on-going research.

2.2.4 Carrying out Proof

Two proof tools have been studied, CADiZ [41] and PVS [3]. Of these, PVS is the more mature and, currently, the more generally powerful tool. This is mainly because a large library of lemmas and useful tactics has been developed over the years. It uses a Lisp-like notation, but provides the means to emulate other notations where desired. CADiZ is specific to Z, but again one may embed other notations into Z, specifically RTL in this context.

There has been much debate as to the best way to develop this work. Experimental results to date are inconclusive, since the only results which have been obtained come from experiments which have too many different characteristics, namely:

- a) work using the RTL formalism in PVS and using a fair amount of manual rewriting in preparing for the formal proof;

- b) work using the operational model and Z, working solely through the interface tool and CADiZ.

The results obtained from approach a) have been the more impressive so far. It is hoped eventually to create a viable process using the interface tool to extract the description of the SUD, an automatic generation of the appropriate RTL description, perhaps embedded in Z, and human-assisted automatic proofs of required properties, using whatever tool offers the best performance at that stage.

2.3 Testing and Formal Specifications

Work funded by the EPSRC and Roll-Royce is investigating approaches to test automation. Testing is still the principal means of gaining confidence in the correct operation of software and, for the class of application considered here, may account for 50% or more of total development costs. In consequence, improving the effectiveness and efficiency of testing may have a big impact on project costs. Testing should not, however, be viewed as an activity isolated from the rest of the development process. Increasing value of the testing process requires an integrated approach.

The punishment-oriented dictum “Pay now **or** pay **more** later” - the notion that effort invested in getting things right early saves the greater cost of correction later – is now accepted by software practitioners and is a major lever for the use of advanced specification techniques. The more optimistic “Pay now **and** pay **less** later”? – the notion that investment in the early stages of the life-cycle enhances later phase activities – has fewer adherents. In time, however, it may prove the most potent driver for the adoption of formal specifications. In particular, a synergistic approach to testing based on formal specifications promises great benefit.

Testing is supported by formal specifications. Test case design benefits from a clear understanding of functionality. More accurate test cases result in fewer flaws being propagated through the development life-cycle, reducing the amount of retest needed. A reduction in the number of propagated flaws and subsequent retest will allow stopping criteria based on bug discovery rate to be met earlier in the life-cycle. The principal benefit, however, lies in the potential for test automation. Formal specifications may permit the automatic derivation of a set of test case specifications that collectively exercise the system against rigorously identified adequacy criteria; some important research has emerged on this aspect. Less well-researched is the ability to automate the search for flaws. If this approach can be shown to scale up to handle complex software then the potential is considerable. Finally, this paper has already outlined how graphical specifications can be supported by formal notations. This formality also aids the testing process, providing another lever for their adoption.

Some of the above issues are addressed below, concentrating on the test automation aspect which Ould [31] has identified as the most crucial issue in testing. It is convenient, for reasons that will become clear, to address low level (principally code) specification issues first before more abstract system specification issues.

2.3.1 Assertions and Partial Specifications

Assertion checking is one of the most powerful techniques in software testing. The analyst inserts predicates about

the program state at various execution points. These predicates are couched in terms of an annotation language (typically embedded as comments of the implementation language but with restricted syntax). The annotations are effectively formal specifications of expected behaviour at the program level. Such annotations are at the heart of program proof. Established techniques (weakest precondition generation and symbolic execution) can be used to generate verification conditions that can be discharged to prove functional correctness. These assertions can, however, readily form the basis for much automated dynamic testing as is shown below.

Random testing by positive oracle. Large amounts of input can be randomly generated and outputs can be checked to verify that specific assertions hold, thereby obviating the need to pre-calculate expected results. Assertions may provide a full functional specification, i.e. a precondition and a post-condition, or else define only critical behaviour properties. A practical approach to formal specification does not necessarily entail a complete functional specification; the rigour of specification must be related to the criticality of the property at hand.

Aggressive randomised testing by negative oracle (falsification testing). Finding bugs is one of the purposes of testing but most current testing approaches are not specifically targeted at bugs. Rather, they are targeted at achieving particular coverage measures, in the hope that this will be sufficient to reveal flaws in the software. Such approaches will discover some flaws since the derivation of the criteria is influenced by empirical understanding of where bugs commonly occur. Randomised testing, as described above, is unlikely to be particularly effective at finding flaws, though no observations are made here about its merits compared to other established techniques such as equivalence partitioning. Instead a more targeted approach is advocated where automated support can *home in* on flaws [42]. Consider the following program:

```
Pre-condition True
x : integer range -10..10;
y : integer;
y := x * x + 2;
Post-condition (y > 2)
Negated post-condition (y ≤ 2)
```

An optimisation approach is used to find values of x that cause the *negated* post-condition to be true. With each input x we associate a cost that reflects how close it comes to making the condition true. Zero cost is associated with an input that causes the condition to be true. If x has values 10 and 3 then y ends up with values 102 and 11 respectively. We see that the x value of 3 comes closer to making the condition true (and so should have a smaller cost). An optimisation technique could cause x to vary from a start point

until a zero cost solution is reached, homing in to an x value of 0. In many cases the relationship between the cost and the input variables will be non-linear and it will be possible to get stuck in local optima. For this reason we believe that non-linear optimisation techniques should be used. Though they are not guaranteed to reach a result, assuming there is one, they provide very powerful heuristic search capabilities. The fully automatic techniques can home in on bugs in the program. Predicates can be reduced to disjunctive normal form (DNF) to enhance the search. Conversion to this form from more general, 'user-friendly', predicate expressions is well-established and fully automatic. Each predicate component in DNF defines one of the ways the program can fail and each can be the subject of a targeted search. The description above is couched purely in terms of the (negated) post-condition. In practice, other predicate assertions, such as pre-conditions or exception-conditions, can be included too.

Testing for exceptions, i.e. run-time errors, has not attracted a great deal of attention from researchers and remains poorly understood. For high integrity developments, formal verification conditions can be generated that, if discharged, guarantee freedom from exceptions. However, the proof of such conjectures requires a great deal of skill and effort. If the code is not exception free then the proof attempts are doomed to failure. For most projects such rigour gives way to ad hoc methods, typically regarding testing for exceptions as a by-product of other testing. There would appear little reason why a representative input for a functional equivalence class should be expected to raise an exception. The aggressive approach to fault finding described above can help.

2.3.2 Test Specification and Architectural Synthesis

Test specifications allow the rigorous derivation of test sets. Consider a model-based specification of an operation where the state space is defined as a collection of variables linked by some state invariant and the operation maps inputs and before states to outputs and after states. The operation itself may be phrased as the disjunction of several sub-operations, each defined for a subset of the input-state space. Each such sub-operation defines a test case requirement for one or more tests. The partition-based method has received research attention from the late 1980s [12] and attempts have been made to automate the process [36]. A thorough, well-defined criterion for requirements coverage can be also implemented based on the partitions.

Translating specification assertions into code level assertions will bring the benefits for testing outlined above. The feasibility of such translations depends upon the clarity or even directness of the mapping between the specification and implementation. The restricted structures of State-

charts may make automated architectural specification (e.g. creation of code templates for Ada packages) feasible and practical. Automatic code generation is for many the holy grail of software engineering. It is, however, fraught with difficulties. Automatic code template generation might provide a half-way house. It is clearly easier than automatic code generation but can bring testing benefits (particularly with respect to automation) by providing the mapping required to translate specification assertions into code level assertions. The restricted structure of engineering notations such as Statecharts should mean that special purpose heuristics can be developed to discharge various proof obligations that arise and also to prove that specific bindings of test data actually constitute formally correct test cases.

2.3.3 Proofs and Testing

Formal methods and testing have not been easy bed-fellows in the past. Proofs have been seen as being for the cognoscenti, testing for software engineering's working class. This social dichotomy is harmful; contrary to current beliefs each can in fact contribute to the other.

Invalid conjectures can waste a large amount of effort. Therefore before a long and arduous proof is embarked upon it is reassuring to have a good degree of confidence in the correctness of the program. Use of aggressive fault finding techniques can locate bugs (or hint at their absence) and so give the developer confidence that what they are attempting to prove is correct. Thus, targeted testing should be the first stage of the proof process, bringing the code at hand up to sufficient quality to merit the proof effort.

In addition, test cases themselves may be prone to error; the test designer may misunderstand the specification in the same way as the implementer and so will calculate the same erroneous results. Formal methods can help here. It is possible, for example, to prove that test inputs satisfy particular preconditions of an operation and that outputs from execution satisfy the postconditions (modulo appropriate refinement retrievals - i.e. a suitable mapping from specification data elements to program elements).

2.3.4 Basic Themes

Some basic themes relating formal methods and testing can now be summarised:

- **Rigorous test set specification.** Formal derivation of test set specifications makes clearer what tests are being performed and what the criteria for their creation has been. This has the potential to remove the ad-hoc nature from much of today's practical testing. Test set specification, based on the regular structure of formal representations of engineering languages such as

Statecharts, should also be amenable to a large degree of automation.

- **An underlying formality.** Graphical notations have proven popular with engineers. Earlier text has indicated that solid formal bases can be created to underpin some of these notations. Properties of interest can be specified or otherwise extracted and couched in a formal notation. The formal descriptions can be used as the basis for testing as outlined above.
- **Partial formal specification.** Formal specification does not necessarily mean full functional specification. Engineers should be encouraged to target their attempts at formality much more selectively. Partial specifications are likely to be the norm except in extreme cases.
- **Fault directed testing.** Automated stochastic search has the potential to find errors which would be missed by normal means inexpensively. Exception generation, a particular variant of falsification testing, is an obvious example of fault directed testing. Falsification testing should be performed to increase confidence in the program's correctness before a proof of correctness is attempted.
- **Testing and proof are complementary.** Testing can be used to easily (and automatically) demonstrate the existence of faults in programs, avoiding the wasted effort of ill-fated program proofs. Proofs can also provide confidence in the correctness of test cases. The restricted forms taken by descriptions of statecharts in formal notations may make possible the development of powerful heuristics for automatic test generation and proof tactics for demonstrating the correctness of the tests (with respect to certain specification properties).
- **Testing where proof fails.** In some cases formal proof may be very difficult or impossible. The automation of formal methods may be insufficient and manual intervention may be required to discharge proof obligations (e.g. when algebraic simplification is just not powerful enough). Additionally, particular language constructs may militate against the use of formal program proof. These may be regarded as cases in which testing effort may be fruitfully directed [43] as a means of gaining confidence in the programs.

3 One Giant Leap

The three projects described above represent, we believe, important steps towards providing more widely applicable

formally-based engineering methods for the domain of real-time, safety critical systems. The astute observer will also have noted that there is a common theme linking these steps - the need for, and use of, a formalisation of the semantics of state transition diagrams. Thus we hope that we can get synergy between these strands, or steps. However, even if we meet all our objectives with these projects we will, we believe, still be a long way from fully addressing the issues raised in section 1. We briefly describe some other issues of concern, then explain the use of the term “one giant leap”.

The following list, whilst by no means exhaustive, identifies some of the key issues which we believe need to be addressed to produce effective formally-based engineering methods - but which we see receiving scant, or at least inadequate, attention in the literature:

Modularity and interfaces – a key to structuring systems and managing complexity is the ability to modularise specifications and to impose and police interfaces between the modules. Put more strongly, without the ability to “divide and conquer”, it is not practical to deal with large scale systems. There is work on “compositionality” which, in part, addresses such issues – but much of this work is analogous to building cathedrals by gluing together matchsticks – one is reduced to reasoning at a very low level and thus the approaches simply do not seem to scale. We would like to see explicit work on the specification and analysis of interfaces which allows modular reasoning – Cliff Jones’ work on rely and guarantee conditions is one approach we have seen for a particular form of module interaction, but we believe there is a need for a much more complete and powerful theory.

Configuration and version management – in any software engineering process specifications go through many versions, and software configurations (the group of modules which make up the system) change over time. To be usable in an engineering context therefore we need means of bringing specifications under version and configuration control. Of course this can be done at the file level - but this is semantic free. We are aware of tool support which supports some configuration management, such as the B development environments. There is also work at the SVRC looking at this issue [24], but this is a fundamental engineering issue which is receiving too little attention.

Change management – to use the time-honoured phrase, in any engineering programme “the one constant is change”. We therefore need methods that help us with change:

- Identifying the impact of change;
- Propagating changes;
- Deciding when to reject changes.

The “killer” problem with formal methods and change seems to be the inability to repeat, cost-effectively, proofs following change. With tactics in theorem provers, etc. some small changes can be accommodated with minimal human effort. However, non-trivial changes tend to require a large – and disproportionate – amount of effort to “re-build” proofs. Unless ways can be found of overcoming such problems, fully formal development will remain expensive, and will probably be confined to very small, stable, parts of systems. We know of no work directly addressing this problem.

Review guidelines – in industrial software engineering reviews are the single most effective means of identifying errors in specifications and programs. Most review processes are supported by guidelines, or checklists, to help ensure a focused and effective process. Arguably some of the work undertaken by Dave Parnas and his colleagues addresses reviews (at least one of his aims is to make specifications more reviewable); the same might be said of the work by Mills and his colleagues on the “clean room” approach [28]. Of current research programs, that undertaken by Parnas’ group is the “exception which proves the rule”. So far as we are aware, there is no substantial research programme addressing what is the cornerstones of any industrially viable software engineering process, i.e. the provision of review guidelines for formal methods.

There are many other issues which we could cite – but the above is sufficient to make our point. Much of the “science” of formal methods is being studied – there is (technically excellent) work on process algebras, on theorem provers, on refinement, etc. But we see alarmingly little work on the engineering issues, some of which are outlined above.

It might be argued that all we have done is to identify some more “small steps” - but we have chosen to use the (emotive) term “one giant leap”. There are two main reasons for this. The first is cultural. We see too much research being undertaken in the spirit of “pure” scientific enquiry, not as a form of problem solving. As Parnas explained in a recent SigSoft article [32], a major characteristic of engineering research is to seek to solve real world problems. It seems to us that the culture shift needed for the formal methods community to address the above problems is a giant leap – in attitude.

The second reason is to do with the need to provide integrated solutions to the above problems. Unless we can provide plausible solutions, or ameliorations, to the above problems at the same time, within the same framework the results may not be industrially usable. To put it more concretely, if we have specifications we know how to review but which aren’t modular, and which we can’t change efficiently, then we may not be able to apply them on realistic

problems. To put it more crudely, this is a “giant leap” because it is a major technical problem which does not decompose well and which needs to be addressed holistically, or at least through a set of synergistic research projects (note the synergy between the three projects described above).

Our belief is that the lack of these “engineering attributes” have been a more significant impediment to the use of formal methods than the supposed difficulty of learning the underlying set theory, or the use of Greek symbols in Z. Again we can legitimately talk about a giant leap – to solve all these problems within a coherent framework is a major intellectual challenge – and one which we don’t see being studied in a holistic manner. Of course we do not seek to denigrate the whole of the work done to date in the formal methods community, or to say that none of it addresses the above problems. However we believe that the level of usage of formal methods in industry will not change dramatically unless there is a substantial shift in emphasis in the research community.

4 Conclusions

Whilst we have discussed what we perceive as problems in sections 1 and 3, we said in the introduction that we are taking the standpoint of optimists. There are several reasons for optimism. First, there are some impressive industrial applications of formal methods and some encouraging research results. In the former area we can cite SACEM [4] as one example. In the latter, recent progress on model checkers offers some promise for improved automation, perhaps including the handling of change. Second, we perceive some shift in focus of the research community – the existence of a series of conferences such as ICFEM is one indicator of this. Also, some widely respected figures, most notably Tony Hoare [18] have identified some of the limitations and fallacies (invalid assumptions) associated with the “classical” approach to formal methods. Whilst Hoare has been criticised, we hope that, because of his eminence, people will listen to his message and reconsider their research directions. Third, the approaches we are developing are being driven by the needs of the aerospace industry, and we see some industrial acceptance of our ideas – and other application domains, e.g. automotive, are following similar avenues.

However our optimism is tempered by realism. One of us has previously stated the view that formal methods are “over-sold and under-used” [25, 27]. After nearly ten years (the initial book chapter was written some time before it was published) we see little to change this view, although the cries of the salesmen seem rather fainter these days. Again, to repeat former observations, a fear is that the phrase “over-sold and under-used” may become an epitaph for formal methods [26].

Our view is that the development of domain specific languages, facilitating automated testing, is the biggest hope for achieving wider application of formal methods within the real-time safety critical systems community, in the near future. Our hope is that this does not represent the zenith of the use of formal methods and that success in this area will provide some of the needed encouragement towards realising the full potential of this technology.

5 Acknowledgements

The work described here owes much to our sponsors and has been influenced by discussions with a number of colleagues. We acknowledge their contributions, but should make clear that these people and organisations do not necessarily espouse the views set out above. The work most directly described above is funded by the Ministry of Defence through the Practical Formal Specification study, by Rolls-Royce through the University Technology Centre in Systems and Software Engineering, by British Aerospace through the Dependable Computing Systems Centre and by the EPSRC through grant GR/L42872. We have also benefited from collaboration with Daimler Benz in related areas, and some of the above ideas have been developed and consolidated through work on another EPSRC funded project (GR/K63566) investigating the formal basis for state and mode transition systems. In addition, acknowledgements go to Jim Armstrong, Phil Brooke, Trevor Cockram, Nick Cropper, Peter Lindsay and Shaoying Liu who have influenced our ideas – perhaps unwittingly. In particular, Jim Armstrong has undertaken the work with PVS discussed in section 2.2.4.

References

- [1] J.-R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [2] P. Behm, P. Desforges, and J.-M. Meynadier. MÉTÉOR: An Industrial Success in Formal Development (Abstract). In *Proceedings of B '98: Recent advances in the development and use of the B method*, number 1398 in Lecture Notes in Computer Science. Springer-Verlag, 1998.
- [3] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. In *Proceedings of WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, 1995. Available from: <http://www.csl.sri.com/wift-tutorial.html>.
- [4] B. Dehbonei and F. Mejia. *Formal Development of Safety-critical Software Systems in Railway Signalling*. Prentice-Hall, 1995.
- [5] R. W. Floyd. Assigning Meanings to Programs. *Mathematical Aspects of Computer Science*, 19, 1967.
- [6] M. D. Fraser, K. Kumar, and V. K. Vaishnavi. Strategies for Incorporating Formal Specifications. *Communications of the ACM*, 37(10), 1994.

- [7] A. Galloway and W. Stoddart. An Operational Semantics for ZCCS. In *Proceedings of ICFEM 97*. IEEE Press, 1997.
- [8] A. J. Galloway, T. J. Cockram, and J. A. McDermid. Experiences with the Application of Discrete Formal Methods to the Development of Engine Control Software. In *Proceedings of DCCS 98*. IFAC, 1998.
- [9] C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO: A Logic Language for Executable Specifications of Real-time Systems. *Journal of Systems and Software*, 12(2), 1990.
- [10] D. Harel and A. Naamad. The StateMate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4), October 1996.
- [11] D. Harel and M. Politi. Modeling reactive systems with statecharts: The StateMate approach. Technical Report Part No. D-1100-43, i-Logix Inc., June 1996.
- [12] I. Hayes. Specification directed module testing. *IEEE Transactions On Software Engineering*, 12(1):124–133, January 1986.
- [13] C. L. Heimeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3), 1996.
- [14] M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, (138), 1995.
- [15] M. G. Hinchey and J. P. Bowen, editors. *Applications of Formal Methods*. Prentice-Hall, 1995.
- [16] C. A. R. Hoare. An Axiomatic basis for Computer Programming. *Communications of the ACM*, 12, 1969.
- [17] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [18] C. A. R. Hoare. How Did Software Get So Reliable Without Proof. In M.-C. Gaudel and J. Woodcock, editors, *Proceedings of FME '96*, number 1051 in Lecture Notes in Computer Science, 1996.
- [19] I. Houston and S. King. CICS Project Report: Experiences and results from the use of Z in IBM. In *Proceedings of 4th International Symposium of VDM Europe*, volume 1 of *Conference Contribution*. Springer-Verlag, 1991.
- [20] ISO/IEC. *IS 8807: Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*. ISO/IEC, Geneva, Switzerland, 1988.
- [21] F. Jahanian and A. K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions*, SE-12(9), 1986.
- [22] F. Jahanian, A. K. Mok, and D. A. Stuart. Formal specification of real-time systems. Technical Report TR-88-25, University of Texas at Austin, June 1988.
- [23] C. B. Jones. *Systematic Software Development Using VDM (2nd Ed.)*. Prentice-Hall, 1990.
- [24] P. Lindsay and T. O. Supporting Fine-grained Traceability in Software Development Environments. In *Proceedings of the 8th International Symposium on System Configuration Management*. Springer-Verlag, 1998.
- [25] J. A. McDermid. *Formal Methods: Use and relevance for the development of safety-critical systems*. Butterworth-Heinemann Ltd, 1993.
- [26] J. A. McDermid. Over-Sold and Under-Used: An Epitaph for Formal Methods? (Abstract). In D. J. Duke and A. S. Evans, editors, *Proceedings of the BSC-FACS Northern Formal Methods Workshop*. Springer, 1997.
- [27] J. A. McDermid and L. Barroca. Formal methods: Use and relevance for the development of safety critical systems. *Computer Journal*, 35(6), 1992.
- [28] H. D. Mills. The New Math of Computer Programming. *Communications of the ACM*, 18(1), 1975.
- [29] R. Milner. *Communication and Concurrency*. Prentice-Hall, Hemel Hempstead, 1989.
- [30] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.
- [31] M. Ould. Testing - a challenge to method and tool developers. *Software Engineering Journal*, 39:59–64, March 1991.
- [32] D. L. Parnas. On ICSE's "Most Influential Papers". *ACM Software Engineering Notes*, 20(3), 1995.
- [33] S. Paynter, J. Armstrong, and J. Haveman. ADL - the activity description language. *Formal Aspects of Computing*, 1997.
- [34] J. Rathke and M. Hennessy. Local model checking for a value-based modal μ -calculus. Technical Report 96:05, Department of Cognitive and Computing Sciences, University of Sussex., 1996.
- [35] L. T. Semmens, R. B. France, and T. W. G. Docker. Integrated structured analysis and formal specification techniques. *The Computer Journal*, 35(6):600–610, 1992.
- [36] H. Singh, M. Conrad, G. Egger, and S. Sadeghipour. Tool-supported test case design based on Z and the classification-tree method. *First IEEE International Conference on Formal Engineering Methods*, November 1997.
- [37] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.
- [38] C. Stirling. An introduction to modal and temporal logics for CCS. *Lecture Notes in Computer Science*, 491, 1991.
- [39] K. Taguchi and K. Araki. The State-based CCS Semantics for Concurrent Z Specifications. In *IEEE Int. Conf. on Formal Engineering Methods '97*, pages 283–292. IEEE Computer Press, 1997.
- [40] I. Toyn. Innovations in the notation of standard Z. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *Z User Meeting*, Berlin, September 1998. Springer.
- [41] I. Toyn. *The CADiZ Web Pages*. 1998. URL: [http : //www.cs.york.ac.uk/~ian/cadiz](http://www.cs.york.ac.uk/~ian/cadiz).
- [42] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *Software Engineering Notes, Proceedings of the International Symposium on Software Testing and Analysis*, volume 23, pages 73–81. ACM/SIGSOFT, March 1998.
- [43] N. Tracey, J. Clark, and K. Mander. The way forward for unifying dynamic test-case generation: The optimisation-based approach. In *International Workshop on Dependable Computing and Its Applications (DCIA)*, pages 169–180. IFIP, January 1998.
- [44] UK Ministry of Defence. *Defence Standard 00-55 - The Procurement of Safety Critical Software in Defence Equipment*. 1997.
- [45] R. J. Vinter. *Evaluating Formal Specifications: A Cognitive Approach*. PhD thesis, University of Hertfordshire, Division of Computer Science, 1998.