A personal view of Formal Methods

B A Wichmann,
National Physical Laboratory,
Teddington, Middlesex, TW11 0LW, UK
E-mail: Brian.Wichmann@npl.co.uk

March 2000

The advantage of being correct is that you do not need to change your mind. J K Galbraith, BBC Interview, 10th January 1996.

1 Introduction

The original introduction of Interim Defence Standard 00-55 [13] created a controversy concerning the industrial application of Formal Methods which does not seem to have been resolved. Here, I am not concerned with the specifics of 00-55, but with the application of 'Formal Methods' in critical systems.

The IEE and BCS have a common working group on the issue of Formal Methods with the aim of attempting to obtain a better consensus. Without such a consensus, discussions on the issue seem to be inconclusive and lack focus. The starting point for the IEE/BCS working party was an IEE brief [8], which was felt by BCS to be inadequate. Rather than make such a negative comment, the desire was to produce something which would be more incisive — but this has not yet happened, although a serious attempt has been made. This paper is a personal contribution to the debate which I hope will eventually allow the Institutions to produce effective guidance on this topic, that is when and to what extent, 'Formal Methods' should be used.

This paper is concerned with using methods like VDM-SL and Z to specify discrete systems of a general nature. More specialized methods which are formal but have a smaller range of application have often been more successful in their application. For instance, the use of SDL in the specification of network protocols has resulted in techniques for the automatic generation of test cases which has been very successful in the quality assurance of network software. These more specialised applications are not considered here.

If I am asked for a single phrase to encapsulate my views on Formal Methods, I would quote John McDermid's remark: 'Oversold and under-used' [14]. Unfortunately, this remark merely highlights the dilemma: how can formal methods be sold more effectively so they are not under-used? In this paper, I consider this problem almost entirely from the point of view of producing safety systems involving software. I do not have personal experience in other high integrity areas, such a security, although I suspect the situation is similar.

As part of the IEE/BCS working group activities, a workshop was arranged at IEE with participation by invitation. The attendance was good, the discussion lively, but I did not feel that the resulting write-up [21] did result in the advice which I think

industry should expect from the Institutions. This is *not* a criticism of the organisers, nor of the excellent write-up by Muffy Thomas, but just the way the event turned out.

All too often, the discussion on Formal Methods seems to be of a method seeking an application, while in industry the requirement is to produce the most cost-effective solution to a problem. Hence the key issue is to determine under what circumstances does the application of Formal Methods provide a cost effective development route.

The confusion about the applicability of Formal Methods is reflected in the standards on the production of critical systems. This debate started with 00-55, but the international consensus standards have different views as follows:

00-55. This standard is for the procurement of UK defence software which is safety critical. It takes a very prescriptive view of applying Formal Methods, with animation and formal code verification.

DO-178B. This standard [16] is guidance for civil avionics software to enable aircraft containing safety related software to be certified by the appropriate Government agencies.

It takes an ambivalent view in which the use of Formal Methods is not required, but can be applied to gain additional assurance. In the USA, the Formal Methods expert, John Rushby undertook a study of the implications of DO-178B for both FAA and NASA. Not surprisingly, John Rushby's excellent study is more positive on Formal Methods [19]. DO-178B does call for substantial effort to be placed in analysis, and it is quite clear that many mathematically-based methods have a direct bearing on this.

IEC 61508. This generic standard [7] is gaining acceptance in many industries outside the civil avionics/nuclear sectors which do not have their own standard.

It classifies numerous software engineering methods in an informative annex. For the more critical systems, Formal Methods are "Highly Recommended" which requires that the method is used, or a justification made for not doing so. In essence, the use of Formal Methods is almost entirely in the hands of the independent assessor.

ITSEC. This security standard [9] is written from a very different perspective than the above. As the title indicates, the substance is that of *evaluation*, rather than security itself.

For levels above E3, Formal Methods are effectively mandated.

MISRA. These automotive guidelines [15] have been produced recently by an industrial collaborative effort.

In general, the guidelines take the view that Formal Methods is an immature 'emerging technology' [1] which cannot necessarily be applied in the automotive industry. This, in my view, is unduly negative and some of the specific points raised in the guidelines are considered later.

Obviously, an international consensus standard requires agreement. There is no doubt that we do not have a consensus in this area, and therefore no clear advice or set of requirements can be expected. Indeed, it is hard to justify *requiring* the use of Formal Methods without a clearer statement of the objectives to be satisfied and the total development context, which is difficult to specify adequately in a standard.

[1] See section 5.5.

2 The traditional view

The traditional means of applying formal methods, at least in the UK, is to start by producing a formal specification in a suitable mathematical notation, almost always VMD-SL or Z. This is the approach specified in 00-55, and hence is even taken by some to provide the definition of Formal Methods. The term *traditional* is used in the note to specify this method of applying Formal Methods.

The immediate problem with this approach is the validation of the formal specification[2]. This is a non-trivial problem and all those that apply this method need to give very serious attention to the validation and how the validation process can be justified.

We have four potential methods of validation (which are not incompatible) as follows:

Use of tools. Tools can process a formal specification. However, their ability to detect subtle errors is often very limited. Tools which check syntax and static semantics are clearly useful, but the 'difficult' bugs tend to arise from deep semantic properties of the problem domain or the program specification[3]. It is highly unlikely that automated tools will ever be able to provide much assistance in this area.

Hence I conclude that *only* using tools as a validation method is inappropriate.

Animation. The standard 00-55 calls for an animation of the specification and a process of validation via that animation [4]. There are several problems here.

Firstly, in many cases, the most elegant specification is implicit and cannot be directly executed. If such a form of specification is used, then another refinement stage is need to produce an executable specification. Incidentally, there is a danger with animation that developers will choose to produce an executable specification which is harder to validate by other means, in order to remove the need for another development step. An executable specification can also suffer from implementation bias, potentially encouraging an implementation method which may not be appropriate for the final system. This point is covered well in Jones' book [10], but developers need to ensure that their process has a review step to detect the problem.

Secondly, animation is merely validation by program execution (ie, testing). This is severely limited in the ability to detect errors unless the specification is very simple, or tools are used to ensure that all aspects of the specification are animated. For data-driven systems, it is clearly essential that realistic test data is constructed, which can be a significant undertaking [5].

Proof. In this context, we are seeking confirmation that the specification adheres to key properties. For instance, if a key safety requirement can be specified using the entities within the formal specification (of the program), then one can, in

[2] John Rushby in [19] states: '.. the certifier and applicant will need to reach agreement on the technical basis for validating formal specifications.'

[3] One bug in the specification of the STV algorithm [22] arose from using a 'set' when a 'bag' should have been used — not easy to detect.

[4] A means of checking software by executing a prototype, which in 00-55, is derived from a formal specification.

[5] An interesting error has occurred recently in the STV implementation that was used for the 1995 Synod elections. The full details are yet to be published, but an ambiguity in the English was not detected during production of the VDM-SL (can one expect to do this?). The error *would* have been detected with test cases at about the branch level, but in this case, only statement coverage level testing had been undertaken.

principle, prove that this requirement will be satisfied by an implementation (of that specification). If this proof cannot be undertaken, then either the problem is 'hard', the proof tools are inadequate, or the specification is inadequate in not ensuring this key requirement will be satisfied. In general this is necessary, but not sufficient; there is no guarantee that functional refinement preserves the safety properties.

In practice, key requirements usually involve part of the external environment to the program (say, the railway layout for a signalling system), and thus it is not possible from the program specification alone to ensure the system satisfies these key requirements.

Software Inspection. By Software Inspection, I mean a controlled manual review process, a good example of which is given in [5], and is commonly known as Fagan inspection. A central problem with a formal specification is that the intended user community cannot typically review the mathematical part of the specification. This implies that their review will depend upon the natural language explanation, with all the attendant ambiguities. One approach to this problem is to re-write (at least in part) the informal specification from the formal specification.

In some cases, users have taken the trouble to learn VDM or Z in order to be able to review key specifications in depth. Failing this, there is a need to perform more detailed validation if high confidence is required in the specification.

Note that the software inspection process compares input and output documents. In this case, it would be an informal specification against a formal one. One would expect that defects would be recorded in the informal specification which should be corrected.

In some contexts, the use of this traditional approach will strengthen the assurance of the specification-to-code step, but the weakest link is the original specification (of the software component). Hence the optimal approach would be to put additional effort into the specification stage (which may involve Formal Methods, but not of the traditional type).

The training issue of the use of Formal Methods cannot be ignored. Reading a formal specification is much easier than writing one, and this should be reflected in the staff profiles.

Any claims made by organisations for the benefits of Formal Methods must be seen in the context of their staff expertise and development skills. Unfortunately, it is not easy to understand the approach taken by companies, still less to compare one company with another.

Some academics maintain (with some justification) that no software engineering development technique has been justified by suitably controlled statistical testing and therefore no claims can be made for the use of such methods. This criticism can easily be made against claims for the use of Formal Methods. I do not think it is reasonable to expect such controlled testing of software engineering methods, since the cost would be huge, and no other branch of engineering accepts the need for testing design methods (as opposed to testing designs which software engineering already involves). In civil engineering, claims are made that the structure is designed to last for 100 years, even when the materials used are novel, and I think this entirely reasonable. Of course, the claims must be treated as such, rather than statements of fact, and must be open to professional review.

One of the fundamental problems we have with safety systems is that the reliability requirements are much higher than can be justified by conventional testing [11]. In this context, Formal Methods appear to offer a magic wand, since a proof of an algorithm can verify the absence of some forms of bugs. The over-selling comes in at this point, which often ignores the following:

- Any proof of the code will assume the specification is flawless.
- Many other requirements must be satisfied before the code proof can be directly applied to a real system, such as ensuring the compiler, operating system or processor chip is fault-free.
- The software used to conduct proofs is very complex in itself, which therefore requires justification (see the topic of 'Tool qualification' in [16]).
- There is no need to strengthen the strongest link in the chain. In fact, there is plenty of evidence that the specification is often the weak link in the chain. (See below for more on this.)

One underlying principle is that of 'divide and conquer'. Hence one would like the production of a formal specification to be a review point of some significance (at least after its validation). Indeed, an attractive property of 00-55 is that the development is naturally divided into two: producing the validated formal specification, and then producing the final system based upon that. One can therefore reasonably ask if the customer would be prepared to 'sign-off' the Formal Specification to mark a contractual milestone. Unfortunately, there is substantial reluctance to do this, because typically the customer does not have the expertise necessary to judge the adequacy of the Formal Specification or its validation. This problem is fundamental: how can a customer gain confidence in a process that he/she cannot validate?

Another aspect of the 'divide and conquer' principle is that one would like the production of the formal specification to be a good fraction of the total effort required. If the Formal Specification is very easy to produce, then all the hard work must come later. Alternatively, if producing the formal specification is too like the main development work, then one has merely moved the effort to another part of the life-cycle (with perhaps little benefit). Below, we give examples of both extremes:

If one specified the RSA public key encryption algorithm [18] formally, the result would be trivial and of little interest (it is just one formula) — all the hard work is refining that into an implementation. Indeed, I am not sure that that particular refinement can be formally verified, since it requires proof of Knuth's multi-length arithmetic routines (or similar routines). This issue illustrates another problem. In the area of mathematics, including numerical analysis, classical mathematical proofs are used, rather than the mechanical verifiers used in computing. Reliance is only placed upon such mathematical proofs after papers have been reviewed in appropriate journals. Reviewing is a labour-intensive process, and hence mechanical verification seems the obvious route. Knuth's book contains mathematical proofs of his algorithms, but I believe that mechanising these proofs would be a significant undertaking. In any case, many numerical algorithms have not been mechanised, which implies that formal verification of computer programs using floating point is not currently practical.

As another example, an algorithm to conduct STV elections has been formally specified [22]. In this case, the specification is virtually identical to an implementation (it is an implementation in VDM-SL). Hence the refinement is trivial, and all the implementation would has been transferred to the specification!

Although these two examples above are extremes, in advocating the use of formal methods in the traditional manner, I would expect to have a clear understanding of where in this spectrum the application is supposed to lie.

An impressive traditional application of Formal Methods is that undertaken by GEC-Alsthom for railway signalling using the B-Method [2]. It is clear that the approach taken has been to apply the method rigorously due to the criticality of the application, in spite of difficulties encountered in undertaking program proof. My view is that it is difficult to see how this method can be generally recommended due to not only the problems of proof (which need not be undertaken for less critical applications), but because the system is designed for proof, and so not undertaking this results in a less natural development. As an example, the GEC system has to compute the energy of the train, which due to the nature of the B-Method is done using integers. This requires an extra refinement step from the application domain, unlike a similar calculation performed within the Boeing 777 aircraft which uses Ada fixed point.

Another equally impressive example of the use of Formal Methods, but with little proof, is that of the CDIS project [6]. Proof was used to validate a key local area network protocol, but this could probably have been undertaken even if the rest of the system was not formally specified. Superficially, this example is a better indication of general industrial use, since code proof must be the exception rather than the rule, even for higher integrity systems. However, I think there are some reasons why Praxis' experience might not provide the basis of a general recommendation:

1. The initial design was undertaken by staff with good VDM-SL expertise, although initially only a small fraction of the implementation team had reading knowledge of VDM-SL.

2. The overall structure of the specification was not top-down, which would be hard to avoid with their design approach.

3. The overall percentages of resources spent on design, code and test seemed little different from more conventional development.

In my view, significant improvements can be made to the effectiveness of applying Formal Methods by applying at the right level and with reduced scope. For instance, in the STV case, Formal Methods were not applied to the input-output or data validation issues.

My advice on this traditional application of Formal Methods is as follows:

1. If the weakest link in the development is likely to be the quality of the informal specification, then you *should* either produce a formal specification of the software, or demonstrate by other means, such as an Inspection, that all likely faults in the specification have been removed.

   The important point here is that early on in the project, one should be able to place substantial reliance upon the informal specification, either by a review, or by its transliteration into a mathematical form. Obviously, you need confidence in the mathematical formulation — it is very easy to formally specify the wrong thing!

2. The specification resolution process should identify key properties of the software, and issues which are secondary to the main software. If practical, a formal specification should be produced of the area involving the key properties, while leaving the secondary issues.

3. Any formal refinement of the code should be undertaken on a highly selective basis. This process can be very expensive, which implies that the developer would typically need to charge additionally for this, which in turn implies support from the customer. (An interesting case for the justification of program proof is with the B-Method for the Paris Metro, which enabled a safety case to be made to allow the trains to be closer and hence increasing the throughput significantly.)

4. If there is any likely requirement for formal code refinement, then the development process must be reviewed in detail against this, to ensure a smooth transition from the current development strategy. Serious consideration should be given to developing software within a highly constrained environment, such as that provided by SPARK [3, 23]. SPARK prohibits aliasing, and thus allows the use of proof tools which would be impossible in other contexts.

One might add: *beware of program proof — it is far too easy to prove the wrong program correct!*

Before leaving this traditional approach to the use of Formal Methods, we must mention one misconception. It is often stated that Formal Methods cannot handle concurrent systems[6], when what is really meant is that neither Z nor VDM-SL can model concurrent systems[7]. In fact, LOTOS, Estelle and SDL can all handle concurrent systems and are widely used for this purpose.

We now turn to other uses of Formal Methods which appears to provide a better opportunity in terms of the cost-benefit ratio.

3 Analysis versus design

In my view, the key issue for some systems is 'the provision of strong engineering arguments that a critical system will behave in an intended manner' [4]. This situation can arise for a number of reasons:

- The system is safety-critical, ie, a software failure could cause injury or death.
- The system has a high security rating is evaluated to, say above E3 in ITSEC [9].
- The customer demands high assurance (and one assumes is prepared to pay for this).
- The system is business-critical in that a serious software fault could put the business into liquidation.
- There is a high recall cost in relation to the selling price.

Since I was initially a mathematician, I find it inherently attractive to use mathematics to support the 'strong engineering arguments'. On the other hand, one needs to be clear that the mathematical reasoning applied directly supports the arguments, rather than being peripheral. I have known situations in which Formal Methods have been applied to the 'easy bit' which has few risks, and *not* applied to the area having the highest risk of undetected errors.

[6] See [15], section 3.3.2.20

[7] Extensions of some mathematical methods can handle concurrency, such as RAISE which extend VDM-SL.

Seen in this light, it is clear that analysis of programs to determine that they will operate as intended is vital. In consequence, the entire design process must be arranged to make this feasible. Since one needs to know that the 'arguments' hold without exception, static analysis is the obvious tool for high assurance. Unfortunately, the form of analysis needed for critical code is almost always 'deep' in that it requires semantic analysis of the program. For a discussion on the forms of static analysis, see [24].

Two quite different problems arise at this point:

Contracting. As an example here, consider the Sizewell B primary protection system. This was written by Westinghouse using a design which was 10 years old when the system went live. The customer responsible for the system was Nuclear Electric, but they had to convince the Nuclear Installations Inspectorate that the 100,000 lines of software was 'safe'. Even though the reliability requirements for the software were modest, obviously the NII needed some convincing. The 'strong arguments' included an analysis using MALPAS [17] of the software, which was very expensive to undertake. If those requirements had been clear 10 years earlier, and it had been possible to relay those issues through the contractual chain, then no doubt the design would have been different.

Hence, issues like a long life-cycle, system designers subcontacting to software experts and the lack of incisive standards in this area, makes it difficult to deliver convincing 'strong arguments' cheaply.

System versus software. Software does not kill, but systems can. Hence the risk analysis needs to be at the system level. However, in very many systems, the complexity and hence the major risk lies in the software. There is a danger that by considering the software in isolation, it may not be possible to demonstrate the acceptability of the system as a whole.

NPL is currently involved in marketing a tool which can be used to directly support the safety case for some systems [20]. For instance, SAAB have used this software (Prover) to model part of the system and the control logic to show that certain unsafe states cannot occur. To achieve this, one needs to identify the critical parts, and model these in propositional logic. This can be quite demanding, but in cases such as that SAAB aircraft undercarriage, the output directly supports the safety case. An interesting application of this tool is in the design validation of PLC systems [1]. Since PLC logic is quite simple, the modelling of the software in the context of the system using propositional logic is quite feasible. Proof of properties of this model then becomes a practical approach to design validation.

Another area of tool support to demonstrate key properties is that of timing analysis. Here, the safety requirement involves a non-functional requirement, i.e, timing, which should be capable of being traced through to the object and checked by a suitable tool.

It seems to me that undertaking some forms of analysis can be a much more effective means of exploiting Formal Methods than the direct application to the software design process which was considered above. However, retrospective analysis, as with Sizewell, cannot be recommended.

We now have a dilemma: given a specific requirement, how can we design the system to ensure that mathematically-based reasoning can directly support the 'strong arguments'?

In the use of mathematical-based reasoning, we need to consider the cost-benefit analysis and the other means of obtaining the necessary assurance. For instance, part of the attraction of tools like Prover is that it can be used to show properties hold which cannot be confirmed by testing, since the input space is too large.

4 Some conclusions

In attempting to arrive at some general conclusions, I re-read John Rushby's excellent report [19]. My conclusions are very similar, albeit with a UK slant.

1. I think Software Engineering needs to be using mathematically based methods (of specification and analysis) much more than it is at the moment. It is hard to see how high assurance can be provided effectively without this.

2. Due to the lack of skills in the more formal methods, one cannot expect widespread use of Formal Methods throughout the entire life-cycle. This does not mean that Formal Methods cannot provide substantial benefit to even the majority of projects, since the application can be to critical sub-systems and properties.

3. Formal Methods should be applied to areas where conventional techniques are inadequate. For instance, a system involving concurrency can easily have a design fault which would be very difficult indeed to locate by testing — hence another approach is needed. Modelling systems using a Finite State Machine is often quite simple, does not involve complex mathematics, and can provide the basis for an analysis of the delivered software.

4. The attraction of Formal Methods when checked by proof is that informal 'reviews' can be replaced by rigorous analysis. However, this can only apply to those parts of the process which are formal. Unplanned external factors can easily undermine the dependability of a system — formal proof of a railway signalling system comes to nothing if the system is not protected against vandals.

5. The traditional Z/VDM-SL approach seems to have inhibited the use of Formal Methods in other ways[8]. Mathematical methods are ideal for the validation of abstract models of systems; but to be effective, the models need to be much simpler that the 'code'. Of course, once such a model has been validated, there must be an argument to show that the 'code' does follow that model, thus demonstrating that the system has the required properties.

[8] John Rushby says: 'Rather (than the traditional approach), the goal would be to establish that certain properties hold, and certain conceptual faults are absent, in formal models of the basic mechanisms necessary for safe operation of the system.'

References

[1] A Borälv, H Årgen. Formal Verification of Programmable Logic Control. Master's Thesis. Uppsala University, Sweden. (Draft, August 1995).

[2] M Carnot, C DaSilva, B Dehbonei and F Mejia. Error-free Software Development for Critical systems using the B-Methodology. Third International Conference on Software Reliability. IEEE. October 1992. pp274-281.

[3] B A Carré and T J Jennings. SPARK — The SPADE Ada Kernel. University of Southampton. March 1988.

[4] D Craigen, M Saaltink, S Michell. Ada 95 Trustworthiness Study: A Framework. To be published.

[5] T Gilb and D Graham. Software Inspection. Addison-Wesley 1993. ISBN 0-201-63181-4.

[6] A Hall. Formal Methods in the Development of an Information System for Air Traffic Control. IEEE Software, (To appear).

[7] IEC 1508: Draft. Functional safety: safety-related systems. Parts 1-7. Draft for public comment, 1995.

[8] Formal Methods in Safety Critical Systems. Public Affairs Report No 9. June 1991. IEE.

[9] "Information Technology Security Evaluation Criteria", Provisional Harmonised Criteria. Version 1.2. 1991. (UK contact point: CESG Room 2/0805, Fiddlers Green Lane, Cheltenham, Glos, GL52 5AJ.)

[10] C B Jones, "Software Development : A Rigorous Approach", Prentice-Hall, 1980.

[11] B Littlewood and L Strigini. Validation of Ultra-High Dependability for Software-based Systems. Comm ACM. Vol 36, No 11, pp69-80.

[12] Draft Guidelines for the use of Formal Methods. Ministry of Defence / National Physical Laboratory. April 1994.

[13] Interim Defence Standard 00-55, "The Procurement of Safety Critical Software in Defence Equipment", Ministry of Defence, (Part1: Requirements; Part2: Guidance). April 1991.

[14] J McDermid. Formal Methods: Use and relevance for the development of safety critical systems, in Safety Aspects of Computer Control, Ed P A Bennett, Butterworth Heinemann, 1993.

[15] Development Guidelines For Vehicle Based Software. The Motor Industry Software Reliability Association. MIRA. November 1994. ISBN 0 9524156 0 7.

[16] Software Considerations in Airborne Systems and Equipment Certification. Issued in the USA by the Requirements and Technical Concepts for Aviation (document RTCA SC167/DO-178B) and in Europe by the European Organization for Civil Aviation Electronics (EUROCAE document ED-12B). December 1992.

[17] Rex, Thompson and Partners Ltd : MALPAS User Guide, MALPAS Release 4.1; RTP/4009/UG, Issue 3 : January 1988.

[18] Rivest, R L, Shamir, A and Adleman, L. A method of obtaining digital signatures and public key cryptosystems. Comm ACM, Vol 21. No 2 pp120-126. February 1978.

[19] J Rushby. Formal Methods and the Certification of Critical Systems. SRI International. November 1993. SRI-CSL-93-07.

[20] Martin Säflund. Modelling and Formally Verifying Systems and Software in Industrial Applications. The Second International Conference on Reliability, Maintainability and Safety (ICRMS'94), 1994. (Available from NPL).

[21] Muffy Thomas. Formal methods and their role in developing safe systems. To be published.

[22] P Mukherjee and B A Wichmann. STV: A case study of the use of VDM. IMA conference on the Mathematics of Dependable Systems. 1993.

[23] B A Wichmann. Strategy on the Use of SPARK. NPL Report 227/94. June 1994.

[24] B A Wichmann, A A Canning, D L Clutterbuck, L A Winsborrow, N J Ward and D W R Marsh. An Industrial Perspective on Static Analysis. Software Engineering Journal. March 1995, pp69-75.