Dear Sir, Yours faithfully: an Everyday Story of Formality

Peter Amey

Publication notes

Invited keynote address at the Twelfth Safety-critical Systems Symposium, Birmingham, UK, 17-19 February 2004.

Published in “Practical Elements of Safety”
Felix Redmill, Tom Anderson (Eds)
Springer
ISBN 1-85233-800-8

Dear Sir, Yours faithfully: an Everyday Story of Formality

Peter Amey

Praxis Critical Systems, 20, Manvers St., Bath BA1 1PX, UK
peter.amey@praxis-his.com

Abstract. The paper seeks a perspective on the *reality* of Formal Methods in industry today. What has worked; what has not; and what might the future bring? We show that where formality has been adopted it has largely been benefical. We show that formality takes many forms, not all of them obviously "Formal Methods".

1 Introduction

This could have been the shortest paper in the history of the SCSC. Formal Methods were briefly promoted by a small group of academic zealots in the 1980s. There were no useable tools, the methods didn't scale to real problems and industry—after careful evaluation of the evidence—quickly abandoned them. Formal methods are now dead (more dead even than Ada, if that's possible). End of story.

Not true, certainly, but a pastiche that is not a million miles from the perceptions, and prejudices, of much of our industry. These misconceptions and myths were well expressed by Anthony Hall over 10 years ago [Hall 1990]; many of his observations are as fresh and relevant as if they were written yesterday. The truth about Formal Methods, as always, lies somewhere between the extremes expressed by pundit and critic. Formal methods have not proved to be a panacea (although this claim was usually a straw man erected by critics rather than something seriously advanced by enthusiasts). Neither have Formal Methods died. In fact they are widely used; sometimes in a rather niche way; sometimes so routinely that they no longer attract attention, and sometimes in disguise.

I hope to show that formality, whether or not it comes with the label "Formal Methods" is alive and well and has a future. Indeed, if our industry is to rise to the challenges of the 21st century, mathematical rigour, in some form, will have an essential role to play (as it has in *all* other engineering disciplines).

For the rest of this paper I propose to abandon the capitalization of Formal Methods. In any case, this promotion to proper noun status is, I think, harmful in that it somehow labels normal good engineering as "special", "unusual" or just plain "different". I will use the term *formal* simply to mean *underpinned by mathematical rigour* (whether or not the mathematics is visible to the user). A formal method (without the capitals) simply means a *method underpinned by mathematical rigour*.

2 An Historical Overview

The need for precise ways to interact with the fast but stupid machines we call computers was recognised early in their life. In a 1948 University of Manchester paper, Alan Turing noted "one could communicate with these machines in any language provided it was an exact language" and "the system should resemble normal mathematical procedure closely, but at the same time should be as unambiguous as possible" [Hodges 1992]. Despite this early wisdom, the initial priority was for productivity rather than precision. It was only at the beginning of the 1970s that it became clear that the growing power of hardware and the availability of high-level languages such as FORTRAN, meant that it was possible to construct systems that exceeded our capabilities for specification, verification and validation. In his 1972 Turing Award lecture, Dijkstra observed: "The vision is that, well before the 1970s have run to completion, we shall be able to design and implement the kind of systems that are now straining our programming ability at the expense of only a few percent in man-years of what they cost us now, and that besides that, these systems will be virtually free of bugs". I think, in defence of Dijkstra's reputation, we should make it clear this was a *vision* not a *prediction!* We should also be clear, as observed recently by Martyn Thomas [Thomas 2003], that 30 years after that lecture, we have not even come close to achieving Dijkstra's vision.

The Formal Methods (capitals intentional here) movement that really got under way in the 1980s was just one of the responses to the growing software crisis. Other responses included the development of Ada by the US DoD, to replace the polyglot chaos that they then endured (and are now rapidly returning to); the adoption of static analysis by UK MoD certification agencies; and the production of new, tighter standards such as Def Stan 00-55. My overwhelming recollection of each of these events is the hostility they generated in much of the software industrial base. I particularly recall the launch event and panel session at RSRE Malvern for Interim Def Stan 00-55 where industry representatives queued up to explain that it wouldn't work, couldn't be made to work and shouldn't even be tried while at the same time remaining silent on how else we might advance the software engineering cause.

These perversities may be explained by the "guru effect". Since we have imperfect processes for producing software, we often rely on the exceptional skill of a very few people to get things to work. These are people who, amongst other things, happen to be able to work effectively with our imperfect processes. These people soon get a reputation for their skill. When alternative approaches are mooted, to whom does the manager turn for advice and guidance? These same experts. Are these experts likely to recommend the adoption of techniques that may undermine their status as company guru? No. They may recommend change but only to something they are comfortable with or which they find cool or interesting. In this sense, the company-saving guru may also be its biggest obstacle to progress.

3 No Evidence?

Before describing the current situation and laying down some hostages to fortune for the future, I would like briefly to address the suggestion that industry decided not to

adopt formal methods because a careful business analysis revealed little advantage in it. If only that were true. I would even be happy to stop promoting precision, rigour and formality if it were. The truth is that:

- our industry is highly prone to rejecting or adopting technologies without *any* kind of analysis of their merits (witness the rush to object oriented methods or to the UML); and
- such evidence as is available shows the adoption of formal methods to be very largely advantageous.

As a motivator there follows a brief selection of formal methods successes. Note that these are *business* successes as well as *technical* successes.

3.1 CDIS

The CCF (Central Control Function) Display Information System is rather an old project now, it was delivered by Praxis to the UK CAA (Civil Aviation Authority) in 1992. It rates a mention here because of relative novelty of the approach (at the time) and the excellence of the results achieved. An abstract VDM model was produced along with more concrete user interface definitions. Code was produced from progressive informal refinements of the abstract VDM model. The high-availability, dual LAN was specified using CCS (Calculus of Communicating Systems). See [Hall 1996] for a more detailed descrition of this project. End-to-end productivity was the same or better than on comparable projects using informal approaches but the delivered software had a defect rate of about 0.75 faults per kloc, approaching an order of magnitude better than comparable ATC (Air Traffic Control) systems. This extra quality was free. The CDIS system has been exceptionally trouble free in over 10 years service.

3.2 Lockheed C130J

The Lockheed C130J mission computer software is a good example of formality by stealth. The software specification was written in a tabular, functional form using the Software Productivity Consortium's CORE notation [SPC 1993]. The code itself was written in SPARK [Barnes 2003] and SPARK's annotations were used to bind the specification and code tightly together. The CORE specification looks remarkably unthreatening but actually has clearly-defined semantics allowing the automatic generation of test cases from it. SPARK is also rigorously defined and leaves no hiding place for vagueness and ambiguity; indeed, a key benefit from its use was the way it forced coders to challenge anything unclear in the specification they were being asked to implement.

It might be thought that Lockheed were prepared to accept the obvious pain that CORE and SPARK must have inflicted because of the extreme criticality of the software under development. Actually, this view is quite wrong: there was no pain. In fact the code quality improved (by one or two orders of magnitude over comparable flight critical software) and the cost of development fell (to a quarter

of that expected)[Amey 2002]. The savings arose from the reduction in the very expensive testing process associated with achieving Level A assurance against DO-178B [RTCA 1992] and, in particular, the virtual elimination of retesting caused by the detection and correction of bugs. Again we find that formality applied to the development of systems both raises quality and reduces cost.

3.3 SHOLIS

The Ship Helicopter Operating Limits Information System provides existential proof that Interim Def Stan 00-55 was a practical standard and could be followed without exorbitant cost and with good results (even with tools and with computing power that fall well short of their current equivalents). SHOLIS is a safety-related system that determines whether a particular helicopter manoeuvre is safe for a given ship in a particular sea and wind state. SHOLIS was specified in Z, coded in SPARK and correctness proofs (technically *partial* proofs since termination was not proved) used to bind the code and specification together. A survey of the project was carried out in conjunction with the University of York [King 2000]. The project illustrates both the power and the limitations of the formal methods used. On the positive side the proof process was both tractable and cost-effective. In particular, proof effort on the specification, before any coding effort had been expended, revealed many subtle flaws that could have emerged later and been more expensive to correct. Overall, proof—in terms of errors eliminated per man hour expended—was more effective than traditional activities such as unit test. In fact the project provides good evidence for abandoning comprehensive unit testing on projects where strong notations such as SPARK and are used—a significant saving. No errors have been reported in the SHOLIS system in sea trials and initial deployment.

Less positively, the system did experience (a few) late-breaking problems. Some of these were found during acceptance testing and involved requirements that were outside the formal model of the system represented by the specification. For example, there was a requirement for the system to tolerate the removal and replacement of arbitrary circuit boards during operation, something that clearly cannot be specified in Z. The main lesson here is to understand what is within and what is without (outwith for Scottish readers) the formal system model and to take adequate verification and validation steps for the latter.

3.4 MULTOS CA

Praxis Critical Systems developed the Certification Authority (CA) for the MULTOS [MULTOS] smart card scheme on behalf of Mondex International. The approach taken is detailed in [Hall 2002a] and [Hall 2002b]. Unlike some of the other example projects the MULTOS CA is *security critical* rather than *safety critical* and was developed to meet the requirements of ITSEC E6, a security classification broadly equivalent to SIL4 in the safety world [ITSEC 1991]. The system was COTS based and incorporated C++ (for the user interface GUI); C (for interfaces to specialized encryption hardware); an SQL database; Ada 95; and SPARK (for the key security-critical functions).

The system specification was produced in Z and, with the full agreement of the customer, was made the definitive arbiter of desired system behaviour. This agreement was significant because Praxis offered a warranty for the system where any deviation from the specified behavour would be fixed at Praxis's expense; this would have been much more difficult to agree if there was no rigorous description of the system's expected behaviour and therefore no common ground on which to agree whether a warranty claim was justified.

A particularly striking feature of this project is the way that errors were usually detected very close, in time, to the point where they were introduced (see [Hall 2002b]). There were very few cases where an error introduced early in the lifecycle (for example, in the specification) remained undetected until late in the lifecycle (for example, integration testing). This is a key property that distinguishes systems developed formally (which can be *reasoned about*) from those developed in a more ad hoc manner (and which can only be *tested*). A consequence of this early error elimination—the very essence of the term "correctness by construction"—is high productivity and low residual error rates. The MULTOS CA delivered an end-to-end productivity—including requirements, testing and management—of 28 lines of code per man day which is high for a SIL4 equivalent project. This high productivity was coupled with a residual error rate, corrected under warranty, of 0.04 defects per KSLOC (about 250% better than the space shuttle software! [Keller 1993]).

Of particular note is the fact that the customer, who now maintains the system, has adopted and maintained the formal specification because of its obvious value; perhaps as a financial organization they are showing greater wisdom and less prejudice than the software world?

3.5 TCAS

Most, if not all, large commercial aircraft are now equipped with the Traffic Collision Avoidance System (TCAS). Interestingly, the official specification for the TCAS II system, sponsored by the US Federal Aviation Administration, is a formal one, written in RSML (Requirements State Machine Language). Prior to its adoption, the formal specification was produced by the Safety-Critical Systems Research Group at the University of California, Irvine. A parallel effort by an industry group to produce an English specification was abandoned because of the difficulty of coping with the complexity of the TCAS function using an informal notation.

The adoption of the RSML specification has had some important benefits. In particular, it has been possible to check it for mathematical completeness and consistency [Heimdahl 1996]. The widespread use of the specification has also spawned a number of supporting tools including code generators, test-case generators and simulators; none of these would have been possible using an informal, English language specification.

The TCAS example is very instructive: it shows that a formal specification can be written for a system whose complexity defied expression in natural language and that formal specification was usable by reviewers and implementors who were not experts in the specification techniques used.

3.6 HDLC

My final example illustrates the use of model checking in the area of hardware and in communication protocols—in this case a High-level Data Link Controller (HDLC) being produced by Bell Labs in Spain. The controller was initially produced using traditional hardware techniques including VHDL backed by extensive simulation. At a late stage, when the design was considered almost finished and when the builders were confident of its correctness, the formal verification team at Bell Labs offered to run some additional verification checks on the design [Calero 1997]. The checks were carried out using the FormalCheck model checking tool [DePalma 1996]. Very quickly, an error was detected that had eluded all the hours of simulation to which the design had been subjected. At best the error would have reduced throughput, more likely it would have caused lost transmissions. The model checking also helped propose a correction and this correction was itself validated using FormalCheck. Clearly much nugatory effort was avoided and model checking now forms part of the standard design process at the site concerned.

Incidentally, this is far from being an isolated example of the commitment to formal verification of hardware designs using model checking. Intel, for example, are also committed users. See for example [Schubert 2003] whose paper includes the significant words: “The principal objective of this program has been to prove design correctness rather than hunt for bugs”.

4 But *Everyone* Uses Formal Methods!

The above examples show that the adoption of formal methods, in a variety of forms, is highly cost-effective and can deliver a better quality product at a lower cost (and probably to the greater satisfaction of its creators). Why then has industry not adopted such rigour with the same enthusiasm that it shows for objects, UML and automatic code generation? Despite the positive evidence we are constantly told that *no one* is using formal methods now.

Well actually, *everyone* uses formal methods, or a least a formal notation, as part of their development process; this comes as a surprise to many software engineers but is nevertheless true. The end product of any software development process, however chaotic, is a mathematically rigorous and precise description of some behaviour. That precise description is machine code and its precise meaning is defined by the target processor which provides operational semantics for the language used. The debate is not therefore *whether* to use formal notations and formal methods but *when* to use them. In the worst case scenario, we achieve precision (another word for formality) only when it is too late to help us. We have a formal specification of our system but one that can only be *animated* rather than *reasoned about*. We are forced to animate it, by testing, with all the disadvantages that brings, because we have no other choice.

By contrast, earlier adoption of more formal notations has considerable merit. We can start with specifications: the process of specification is finding a precise way of recording some desired behaviour or property. We routinely regard the end

product of this process, the specification document, as being the most important aspect of specification; however, the intangible benefits that accrue *during its production* are at least as valuable. It is during the specification process that we can discover the ambiguities, inconsistencies, lack of completeness and other flaws that would eventually emerge as bugs. The more rigorous our approach to specification the more quickly and more obviously these problems emerge. As Hall puts it "It is hard to fudge a decision when writing formal specifications, so if there are errors or ambiguities in your thinking they will be mercilessly revealed: You will find you cannot write a coherent specification or that, when you present the specification to the users, they will quickly tell you that you have got it wrong. Better now than when all the programming money has been spent!" [Hall 1990].

We must therefore seek to understand why more formal approaches to software development have not generated the unstoppable momentum of UML or visual programming and, from this, find ways of injecting formality into the earlier stages of the software lifecycle where it will do most good.

5 The Situation Today

I find it useful to draw analogies with other engineering disciplines, especially aeronautical engineering which was my original profession. By the time that aeronautical engineering had advanced beyond the craft stage it had acquired a mathematical basis. For example, reasonable predictions of stress in components could be made, but only by very highly skilled engineers working very hard with tools like slide rules (that seem very primitive to us now). Today, stress calculations can be made more easily by less senior engineers using more powerful tools. The crucial thing here is that the mathematics has not been abandoned as too hard but encapsulated in a more accessible and productive form.

I believe there is a strong analogy with formal methods but with a rather less satisfactory outcome. Using the early methods such as VDM, B or Z, a highly skilled engineer (akin to our senior stress man) could make a very rigorous prediction of the behaviour of a software component. Unfortunately, because this was perceived as hard, the response was not, as in the aeronautical world, to encapsulate the mathematics but to give up doing the calculations at all. To stretch the analogy, we might liken UML to an engineer's sketch pad. We can scribble an approximation of what we want to build but we can't analyse or measure it. We are reduced to building it and then testing it to see if (or more usually, where) it breaks. This is well short of the aeronautical equivalent where we may well still test a component but in the expectation that it will not break because our calculations tell us it won't. Citing difficulty as the reason for the failure to adopt a more rigorous approach to software is especially galling when you consider how much simpler is the mathematics of, say, formal specification than that of stress or of compressible aerodynamic flows!

(In case anyone thinks this passing criticism of UML is unfair, let me quote Jos Warmer of Klasse Objecten in the Netherlands: "In many cases, people simply have their own interpretation, which is implicitly known by other people in their team, project or department. The environment and background of the reader/writer of the

model determines its meaning. The consequence of this is that UML is a standard notation, but without a standard meaning."[Warmer 2003]).

5.1 Industrial Need

The failure to attempt a more rigorous approach to software development would not matter if the performance of our industry was wholly to be admired. Regrettably that is not the case. On any objective set of measures our industry paints a story of failure. Cancelled systems, late delivery, poor performance and cost overruns are the norm. Martyn Thomas [Thomas 2003] quotes figures from the Standish Group and from the BCS that show:

The Chaos Report 1995	
Projects cancelled before delivery	31%
Projects late or over budget or which deliver greatly reduced functionality	53%
Projects on time and budget	16%
Mean time overrun of projects	190%
Mean cost overrun of projects	222%
Mean functionality of intended system actually delivered	60%
BCS Review 2001	
Success rate of 1027 projects	12.7%
Success rate of 500 *development* projects	0.6%

hardly a glowing testimony to a thriving industry!

The expectation of failure has become so entrenched that it has corroded the entire basis on which contracts are offered and won. Those asking for a system to be built ask for more than they either need or expect to get. Those bidding for the work offer an unrealistically low price in the expectation that they will be able to deliver less than the contract requires or that they will be able to force the price up when there is an inevitable requirements change. The entire process is dishonest, leaves both parties dissatisfied, further lowers future expectations and discriminates against those professional organizations that are genuinely able to quote for and deliver the requested system. In the end I think this cycle of dissatisfaction actively reduces the amount of work available and may even be part of the cause for the downturn in the software business; after all, given the high expectation of failure, why would a potential customer seek to have anything built unless it was unavoidable? I suspect many systems that could be replaced or updated to the benefit of their owners soldier on precisely because of a lack of confidence that the replacement would be better or even work at all.

5.2 The State of the Art, or The State of the Practice?

What has the above told us?

– Our industry does not have a good track record for delivering dependable systems at predictable cost;

– our industry has largely rejected the adoption of more formal methods; and
– formal methods, when tried, have had an overwhelmingly positive effect on dependability and cost.

Put in that rather stark form it does suggest that formality has been unreasonably neglected and that it should be revisited. Clearly much of what is trumpeted as "state of the art" is actually no more than "state of the practice". We are not failing because of the inadequacies of computer science or because of the lack of applicable techniques, we are failing because we are not using existing methods of proven utility. As Edsger Dijkstra put it so elegantly in 1973: "Real-life problems are those that remain after you have systematically failed to apply all the known solutions".

6 The Future

So how can we move from the morass of sexy but semantic-free languages, pointless but pervasive processes and multiplicities of meaningless metrics onto some logically firmer ground? I think there are several possible routes.

6.1 Traditional Formal Methods

Traditional Formal Methods as typified by Z and VDM continue to be used and research in this area continues. Currently there is interest in the modelling of systems that exhibit a mixture of discrete and continuous behaviour. Another hot topic is the construction of notations that encompass both model-based and process-based behaviour.

The main problem with traditional Formal Methods remains one of perception and prejudice. I have had some interesting conversations with potential clients which have been proceeding very well until I have said something like: "we recommend constructing a formal model of ..." at which point the aghast client manages to combine all of Hall's seven myths into a single sentence that usually finishes with "couldn't we use UML?". In a similar vein, I know of a particularly savvy organization that produced a Z specification for a system but found that most of the implementors they offered it to wanted extra money because of the difficulty of dealing with this unusual artefact; personally I cannot think of any easier or better start to a project than to have a customer who knows exactly what they want and who can express it precisely!

The MULTOS CA project tells us that non-specialists can be readily helped to understand formal specifications and that the precision of such a specification brings contracting and commercial benefits as well as the more obvious technical ones. The production of a formal specification also potentially increases the power of a system procurer since once such a specification exists it should be possible to get the system constructed by one of several developers. By dividing a new project up into separate requirements capture, specification and construction phases risk can often be reduced. Non-formal approaches make it much more likely that the entire process has to be let to a single organization as a single contract and removes

this more incremental option. For this reason alone, system procurers should not be afraid of bidders who suggest using a formal approach.

Unlike a lot of heavyweight, tool-centric approaches, the adoption of formal methods does not have to be done in a single big bang. Often it can be integrated into an existing process. It can be used to specify the most critical part of a system even if it is not adopted for everything. My experience is that even sketching out a few key safety invariants in a rigorous manner can bring useful benefits.

6.2 Formality by Stealth

This is the area that I think offers the greatest promise. It continues the aeronautical engineering analogy offered earlier by seeking to encapsulate mathematical rigour in a user-friendly wrapper. When a mechanical engineer uses a CAD program to perform a stress calculation on a model the tool will use rigorous mathematical techniques such as finite element analysis to produce the result which it will then display in a pleasant range of colours. Similarly, an aerodynamicist may explore some new wing shape using a computational fluid dynamics system. Again, he will get a visualization of fluid velocities and pressures but without direct visibility of the families of partial differential equations that the system is solving to produce them.

The growing popularity of model-based design notations such as UML and their associated tools provides an opportunity (and a risk) here. If we can develop such tools so that they become analagous to the finite element analysis and computation fluid dynamic tools mentioned earlier then we can, potentially, move formality to an earlier stage of the development lifecycle. One trend that might drive things in this direction is automatic code generation. Since machine code is formal it follows that any attempt to generate code from a diagram imposes a semantic meaning on that diagram. Unfortunately, if there is an ambiguous source language between the diagram and the machine code then the connection is rather tenuous and the imposed semantic meaning rather weak. In this case we are reduced to generating code, testing it to see what it does and tweaking the diagram if it isn't quite what we want. The situation is a little different, however, when the generated source code is unambiguous (see [Amey 2002] for a discussion of ambiguity in programming languages). Here the implied meaning of the diagram is directly deducible from the generated source code and so the diagramming language suddenly stops being semantic free. We have seen this effect with a number of tool vendors who have sought to generate SPARK from their design tools: the immediate effect is to force them to be much more precise about what their various diagrams actually mean. (This is exactly the same effect experienced on the Lockheed C130J where coders found themselves unable to express ambiguous specifications). Potentially then, we can find a sound semantics for, say, UML and then apply formally-based analysis to a model expressed in it. That is the hope, and the UML2 initiative, which has raised the emphasis on strong semantics, is a promising sign. There remains, however, the danger is that we won't even try to exploit stronger semantics for the UML. Instead we will tolerate the vagueness of the diagramming notations used on the grounds

that code generation will let us get to test quickly and that testing will reveal any problems; precisely the kind of late error detection we need to escape from if we are to improve the state of the industry.

To avoid the negative outcome, users of code generation tools should challenge their vendors to explain exactly what diagram-to-code mappings are used and exactly what analysis can be performed at the model level.

6.3 Lightweight Formal Methods

The idea of lightweight formal methods was presented by Daniel Jackson at Formal Methods Europe 2001 although the proceedings do not include a full paper on the subject. An earlier version of his ideas is available on the web [Jackson 2001].

The concept of lightweight formal methods is that we are prepared to trade some of the universality and precision of formal methods to make them easier to apply to particular common problems and in common situations. The result is that we may not be able to obtain the range and precision of results that a fully formal approach offers but that we may instead get simpler and more rapid results for particular classes of problems. Crucially Jackson is not advocating abandoning the mathematical rigour of formal methods but its selective deployment with appropriate approximations. Perhaps this is analagous to the simplification that can be made to the analysis of fluid flows at low speeds where we can ignore the effects of compressibility. Our analysis is no longer universal or exact but it is much simpler and still immensely useful for a common class of problem.

Lightweight formal methods overlap with formality by stealth. For example, the Polyspace Verifier tool uses a mathematical technique called abstract interpretation to produce an approximate model of part of the behaviour of a computer program. The approximate model can produce useful results in the identification of potential run time errors. The underlying mathematics places this tool in a different class from the purely heuristic approach of, say, lint.

I have to admit to a slight scepticism with Jackson's ideas. The mathematics of computer software is so much simpler than that of aerodynamic flows that the simplifications don't always seem necessary (although simplifications may greatly speed up some analyses and make them feasible in new situations). Furthermore, for critical systems the failure to detect an error because it falls outside the approximate model being used may be unacceptable. There is for example a clear difference between *proof of the absence of run-time errors* using SPARK (a fully-formal approach) and the static detection of *some run time errors* using Polyspace.

Perhaps lightweight formal methods have most to offer in non-critical systems where an easily adopted approach that found many (but not all) common kinds of error would have a very significant effect on the overall quality of software. For critical systems, their main benefit might be in providing an initial "fast filter" that catches common problems leaving more rigorous processes to eliminate the subtle problems that this first pass misses.

7 Conclusions

Rumours of the death of formal methods are much exaggerated. The methods, in their traditional form, are still being used by the more enlightened practitioners in our industry. In particular, model checking has become an almost unremarked standard process in the design of computing hardware and in the area of communication protocols. More significantly, formality by stealth, the encapsulation of rigour in a less threatening wrapper, has started to make itself felt. Users of SPARK, for example, are employing an unambiguous language with formally-defined semantics supported by tools performing rigorous and exact analyses based on sound mathematical principles. But if you asked some of the SPARK users who are routinely using proof techniques to show their code is free from all predefined exceptions, whether they used formal methods they would probably say "no".

Business thrives on precision. We expect our contracts to be accurate and subject to a single interpretation and we employ expensive specialists to ensure that this is so. We even use standardized forms of formal address in our correspondence: "Dear Sir, yours faithfully" defines a form of interface and it is perhaps in component interfaces that formality has most to offer most quickly as we move towards the construction of systems from off-the-shelf components.

It is clear that the software industry should require the same precision and rigour in the definition and construction of its primary product as it does in any of its other activities. Don't worry about whether something carries the label "Formal Method" but *do* worry, a lot, about whether it is formal in the sense of being: *precise, rigorous, exact, amenable to reasoning;* or *susceptible to analysis*. If your suppliers cannot satisfy you that their methods, however fashionable, meet these criteria then they are falling short of currently achievable engineering standards; they are being unprofessional. If they claim their informally produced software is suitable for use in a highly-critical system, then they are being dishonest as well as unprofessional because such a claim cannot be sustained by dynamic test evidence alone [Butler 1993][Littlewood 1993]. If you, the reader, are a software supplier how well do you pass these tests?

The real benefit of a more formal approach is that it changes the software development mindset from one of *construct and debug* to the more sensible *correctness by construction*. Too often techniques such as static analysis are just seen as new and different ways of finding bugs; only formal methods—and tools based on formal methods—offer a route to avoiding the bugs in the first place.

References

[Amey 2002] Peter Amey. *Correctness by Construction: Better Can Also Be Cheaper*. CrossTalk Magazine, March 2002.

[Barnes 2003] John Barnes. *High Integrity Software - the SPARK Approach to Safety and Security*. Addison Wesley Longman, ISBN 0-321-13616-0.

[Butler 1993] Butler, Ricky W.; and Finelli, George B. *The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software*. IEEE Transactions on Software Engineering, vol. 19, no. 1, Jan. 1993, pp 3-12.

[Calero 1997] J Calero, C Roman and G D Palma. *A practical design case using formal verification*. In Proceedings of Design-SuperCon97.

[DePalma 1996] G. DePalma and A. Glaser 1996. *Formal verification augments simulation*. Electrical Engineering Times 56.

[Hall 1990] Anthony Hall. *Seven Myths of Formal Methods*. IEEE Software, September 1990, pp 11-19.

[Hall 1996] Anthony Hall. *Using formal methods to develop an ATC information system*. IEEE Software 13(2): pp 66-76, 1996.

[Hall 2002a] Anthony Hall. *Correctness by Construction: Integrating Formality into a Commercial Development Process*. FME 2002: Formal Methods - Getting IT Right, LNCS 2391, Springer Verlag, pp 224-233.

[Hall 2002b] Anthony Hall and Roderick Chapman. *Correctness by Construction: Developing a Commercial Secure System*. IEEE Software January/February 2002, pp 18-25.

[Heimdahl 1996] Heimdahl M. and Leveson N. *Completeness and consistency in hierachical state-based requirements*. IEEE Transactions on Software Engineering SE-22, June 1996, pp 363-377.

[Hodges 1992] Quoted in: Andrew Hodges. *Alan Turing: The Enigma*. Vintage, Random House, London, ISBN 0-09-911641-3 or Walker & Co., New York, ISBN 0-802-77580-2.

[ITSEC 1991] Information Technology Security Evaluation Criteria (ITSEC). Provision Harmonised Criteria, Version 1.2, June 1991.

[Jackson 2001] See `sdg.lcs.mit.edu/~dnj/pubs/ieee96-roundtable.html`

[Keller 1993] Keller. T.W. *Achieving error-free man-rated software*. 2nd international conference on Software Testing, Analysis and Review, Monterey, California, 1993.

[King 2000] Steve King, Jonathan Hammond, Rod Chapman and Andy Pryor. *Is Proof More Cost Effective Than Testing?*. IEEE Transactions on Software Engineering Vol 26, No 8, August 2000, pp 675-686

[Littlewood 1993] Littlewood, Bev; and Strigini, Lorenzo. *Validation of Ultrahigh Dependability for Software-Based Systems*. CACM 36(11): 69-80 (1993)

[MULTOS] See `www.multos.com`

[RTCA 1992] RTCA-EUROCAE. *Software Considerations in Airborne Systems and Equipment Certification*. DO-178B/ED-12B. 1992.

[Schubert 2003] Tom Schubert. '*High Level Formal Verification of Next-Generation Microprocessors*. Proceedings of 40th Design Automation Conference. ACM Press June 2003.

[SPC 1993] Software Productivity Consortium. *Consortium requirements engineering guidebook*. Technical Report SPC-92060-CMC version 01.00.09.

[Thomas 2003] Martyn Thomas. *The Modest Software Engineer*. Presention at Software Reliability and Metrics Club, London on 20th May 2003.

[Warmer 2003] Jos Warmer. *The future of UML*.
`http://www.klasse.nl/english/uml/uml2.pdf`