Realising the Benefits of Formal Methods

Anthony Hall
Independent Consultant, UK
email: anthony@anthonyhall.org

I. Introduction

What are the real benefits of formal methods and Why should we care about them? When and Where should we expect to use them, and Who should be involved? I suggest some answers to those questions and describe one approach, *Correctness by Construction* [1], that has achieved practical success on several real industrial developments. Based on this I propose some challenges for formal methods research.

II. Some Questions About Formal Methods

What have formal methods ever done for us?

Formal methods consist of writing formal descriptions, analyzing those descriptions and in some cases producing new descriptions —for example refinements— from them. In what way is this a useful activity?

First, experience shows that the very act of writing the formal description is of benefit: it forces the writer to ask all sorts of questions that would otherwise be postponed until coding. Of course, that's no help if the problem is so simple that one can write the code straight away, but in the vast majority of systems the code is far too big and detailed to be a useful description of the system for any human purpose. A formal specification, on the other hand, is a description that is abstract, precise and in some senses complete. The abstraction allows a human reader to understand the big picture; the precision forces ambiguities to be questioned and removed; and the completeness means that all aspects of behaviour —for example error cases— are described and understood.

Second, the formality of the description allows us to carry out rigorous analysis. By looking at a single description one can determine useful properties such as consistency or deadlock-freedom. By writing different descriptions from different points of view one can determine important properties such as satisfaction of high level requirements or correctness of a proposed design.

There are, however, stronger claims sometimes made for formal methods that are not, in my opinion, justified. The whole notion of proof as qualitatively superior to other analysis methods seems to me wrong: proof is no more a guarantee of correctness than testing, and in many cases far less of one. Furthermore, formal methods are descriptive and analytic: they are not creative. There is no such thing as a formal design process, only formal ways of describing and analyzing designs. So we must combine formal methods with other approaches if we actually want to build a real system.

Why bother?

There sometimes seems to be a belief that formal methods are somehow morally better than other approaches to software development, and that they can lead to the holy grail of zero defect software. This is nonsense, and the fact that it's so obviously untrue is part of the reason for the strong backlash against formal methods. What is true, however, is that formal methods contribute to demonstrably cost-effective development of software with very low defect rates. It is economically perverse to try to develop such software without using them. Figure 1 shows the defect rates achieved by organisations at different capability levels together with the much better defect rates achieved by using *Correctness by Construction* [1], a formal-methods based process described later in this paper.

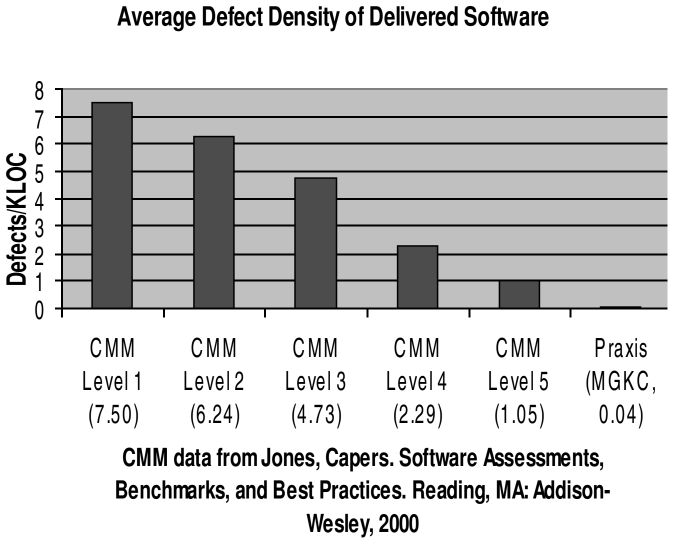

Fig. 1. Evidence for the achievement of low defect rates.

The reason that, contrary to popular belief, formal methods actually save money is illustrated in Figure 2. This shows the cost of fixing a requirements error —the most common kind— depending on when it is discovered. Since formal methods help us discover errors early in the lifecycle, they actually reduce the overall cost of the project.

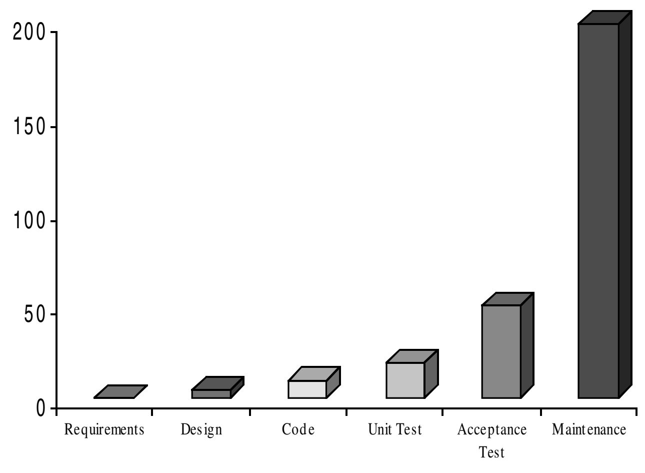

Fig. 2. Cost of correcting a requirements defect according to the stage at which it is discovered.

Furthermore, formal methods provide, for free, the kind of evidence that is needed in heavily regulated industries such as aviation. They demonstrate responsible engineering and give solid reasons for trust in the product. As more and more industries demand such trust, formal methods become increasingly attractive.

In trying to realise the benefits, therefore, we should be looking at cost-effective methods that address the major risks and that provide tangible evidence of trustworthiness. That is not the same as looking for perfection or proving every single piece of code. It does mean using formality where it adds value and exploiting the synergy between formal methods and other activities. For example, if you have a formal specification you can systematically derive effective test cases directly from the specification [2].

When do formal methods bring benefit?

It is well known that the early activities in the lifecycle are the most important. According to the 1995 Standish Chaos report [3], half of all project failures were because of requirements problems. It follows that the most effective use of formal methods is at these early stages: requirements analysis, specification, high-level design. For example it is effective to write a specification formally rather than to write an informal specification then translate it. It is effective to analyse the formal specification as early as possible to detect inconsistency and incompleteness. Similarly, defining an architecture formally, for example as a set of CSP processes [4], means that you can check early on that it satisfies key requirements such as security.

As well as concentrating on the early lifecycle, formal methods need to be used from the start of each activity, not as a check at the end. We should concentrate, I believe, on correct construction rather than post-hoc analysis. Lots of experience with analysis tools tells us that it is far easier and more effective to demonstrate the correctness of a well constructed program than to analyse a poorly constructed one to find the numerous flaws that it contains. However, there is a real human problem in persuading people to think carefully rather than adopting the classic hack and test approach to programming.

Where are they best used?

Formal methods traditionally live in a ghetto where they are applied to critical parts of critical systems. While I don't believe that they will ever be widely applied to fast-moving software such as web pages where the occasional failure is tolerated or even expected, there is an increasing amount of software where failure is becoming unacceptable and costly, and we need to extend the reach of formal methods to a wide range of systems such as banks, cars, telecommunications and domestic appliances.

Within a project, formal methods can be used to a greater or lesser extent. One approach is to use them in a highly focussed way on critical parts of the project. For example one can give a formal specification of just the critical requirements, or specify and model-check critical algorithms. Another approach is to use formal methods to support the V&V process, for example by using a formal specification as a test oracle, without requiring the developers to use them. The most ambitious approach, and the one that yields the most benefit, is to integrate them into the mainstream development process. This does not mean that every single deliverable is completely formal. It does mean that descriptions are formal wherever that brings a benefit, and that the level of formality is chosen to suit the particular area. It means that analysis is done wherever the benefit, in terms of avoided risk, outweighs the cost. It means that the formal specification is the basis for both developing and testing. This approach does require that the formal notations are strongly integrated with the other notations used in the project. For example a formal state model might be closely linked with a more approachable UML Class Diagram.

Who uses formal methods?

There are two ways of answering this question, with different costs and benefits. One way is to have a small specialist team carrying out the formal work. This is relatively easy to introduce and concentrates the use of what is at the moment a scarce resouce, that is skilled formal methods practitioners. However, the benefits are correspondingly limited since the team can only concentrate on small areas, and there is a serious danger of divergence between the formal deliverables and the mainstream project.

The more ambitious approach, and the one that yields bigger benefits, is to integrate formal methods into the whole project. More people need to be trained to use the methods: for example testers need to be able to read formal specifications. The benefits come because now formal and informal methods can be integrated, the whole project benefits from the formality and it is far easier to ensure that formal specification, design, code and tests are all kept in step.

III. One Answer: Correctness by Construction

Correctness by Construction [1] is one point in the spectrum of possible answers to the preceding questions. It exploits the benefits of abstraction to achieve clarity and completeness in the specification. It uses formality to improve both the development process and the assurance activities. It uses formality from the earliest possible stage and throughout the lifecycle, and it involves the whole team in an integrated way.

Correctness by Construction has been developed over many years and is continuing to drive down defect rates as the process is refined. Figure 3 shows the defect rates of projects over a 10 year period.

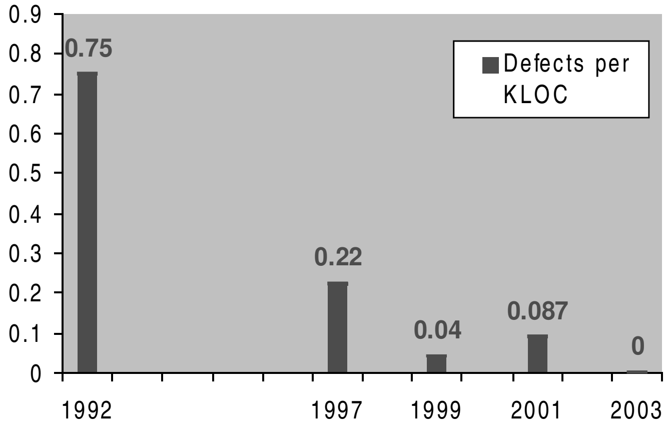

Fig. 3. Progress in Reducing Defect Rates.

Correctness by Construction embodies many of the principles of Lean Engineering. Rather than having an inflexible set of procedures, the process for any given project is designed to address the major risks particular to that project. Nothing is ever done unless it adds value to the project. There is a tight feedback loop making corrections to the product —and indeed the process itself— at all stages. The approach can be summed up in two principles:

- Avoid introducing errors as far as possible.
- Remove those errors that are introduced as soon as possible.

Correctness by Construction achieves these by using the most rigorous notation possible at each stage and by carrying out the most rigorous analysis that is feasible on each artefact.

For example, the system specification is written in a formal notation such as Z. The meaning is unambiguous, and all side effects are clearly visible. One particular benefit is that it forces you to consider all the cases, including error behaviour: this is an aspect often omitted from informal descriptions.

A key principle in C-by-C is to say things only once. For example the Z specification is an abstract, black-box description of the system's behaviour. There is also a user interface specification: this adds syntactic and lexical detail, but it does not repeat the description in the formal specification. Similarly the system architcture defines the structure of modules and processes but does not repeat or expand on the behavioural description in the specificaiton — the descriptions are complementary. Before coding, you can design the detailed module structure and data flow relations in a formally-defined language like SPARK [5].

A final, indirect, but very important benefit of formality is that it strongly encourages simplicity. Because a formal specification needs to be complete, it does not allow you to hide complexity in an informal summary. If it is complicated to specify it will be complicated to build, so effort spent in simplifying the specification is well worthwhile.

Each of the artefacts can be checked, helping to eliminate errors before they get any further. The formal specification can at least be typechecked; where justified, it can be proved consistent and (to some extent) complete, and you can show, perhaps by proof or by model checking, that it satisfies key requirements. It may also be possible to animate it to help validate, with the stakeholders, that it captures the intended behaviour. A user interface specification can be prototyped and critiqued by the users. Process designs can be analysed, for example to show that they are free from deadlocks. SPARK annotations can be analysed to show the absence of dataflow errors — and thus, maybe, the satisfaction of critical safety or security properties. Even better, you can prove the complete absence of run-time errors in your code.

These methods, taken together, are both effective and economical. It has been shown in [6], for example, that carrying out proofs of correctness can actually find more errors, more economically, than traditional unit testing. In addition, C-by-C involves continuous process improvement. We measure the defect introduction and removal rate, and in particular how soon we can remove the defects that are introduced. Figure 4 is a diagram of this information for a recent project [7]. It shows, for example, that 57 errors introduced in the specification were detected and removed at the architecture stage. The aim of process improvement is to reduce the numbers of errors overall and to move them as far to the left of this diagram as possible. Where errors are anomalous —for example the one specification error that survived into operation— we carry out root cause analysis to determine what went wrong and improve the process to try and eliminate such errors in future.

This discussion of C-by-C might make it seem that it is a traditional waterfall lifecycle. On the contrary, it is highly risk-driven rather than document-driven, and it supports concurrent and iterative development. For example, because the architecture and the formal specification are talking about different aspects of the system, they can be written to a large extent in parallel. Once the architecture and the main structure of the specification is in place, there can be many incremental and overlapping build cycles, each adding increments of functionality to the system. This shortens the overall timescale, offers early delivery of partial functionality and reduces project risk by getting early visibility of a working system.

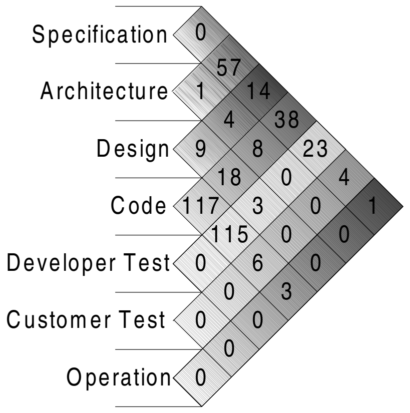

Fig. 4. Defects by Point of Introduction vs. Point of Removal.

IV. How can we realise the benefits?

C-by-C has several successful projects under its belt, and has been proved to deliver outstandingly low defect rates. Praxis routinely offers a warranty with software developed in this way. At the same time, the cost is no more than that of a conventional, lower quality development.

Nevertheless, there is a long way to go before it or anything like it is a mainstream process. There are many challenges for formal methods researchers and tool developers if they are going to support a practical process on a large scale in industry. In this section I suggest some problems that need to be addressed.

Problems of Specification

The key problem for formal specifications is that they are not yet accessible to the stakeholders who need to read them. We need to retain the mathematical rigour but make the vocabulary and even the syntax much more domain dependent so that specifications can be read not just by trained software engineers but by domain experts in automobile engineering, air traffic control or whatever the application is. From a technical point of view, we need languages that are expressive (like, say, Z) but at the same time we want to be able to animate specifications and carry out proofs easily or even automatically. There is an intrinsic conflict between expressiveness and executability, and we need far more powerful tools to allow us to do this.

All real projects need large specifications and all large formal specifications are unacceptably cluttered. We need far better modularity mechanisms, but there are, again, difficulties of principle. For example, a key modularity technique in programs is information hiding, but this is completely inappropriate for specifications since the specification of a module is precisely what its user *must* know about it. We need some more powerful modularity concept of somehow hiding information until it is needed. We also need to solve the framing problem —specifying what does not change, without cluttering the specification with uninteresting predicates.

Although the abstraction necessary for a formal specification removes a lot of implementation information, nevertheless the precision of a formal specification necessarily seems to add detail. This too makes the specification large and unwieldy. Methods of presenting specifications by incremental addition of detail would be very valuable.

Formal notations cannot stand alone. They must be integrated with diagrams, English text and indeed with each other. This raises problems both of principle and of pragmatics. For example we need to be able to compose finite state machines with more general logics, sequential operation specifications with process algebras and abstract specifications with more concrete user or system interface specifications. We need to minimise the number of different documents, eliminate duplication of information between documents and be able to present common information in different views. Current formal methods tools don't allow this sort of integration.

Problems of Design

There is within the formal methods world a very rigorous notion of refinement. Unfortunately this notion bears very little resemblance to the real process of design. First, it supposes that the concrete implementation is a perfect reflection of the specification. In fact, however, all realistic specifications are to a greater or lesser extent idealisations: so the relation we need is retrenchment [8], not refinement [9].

Second, it supposes that the structure of a system is essentially unchanged —it has, in particular, the same set of operations— by the design process. In a real project, however, there is a huge difference between the structure of the specification and the structure of the implementation. A single 'operation' at the specification level may be implemented by a distributed collection of heterogeneous machines running, in parallel, COTS products with poorly defined interfaces, whose operations bear no relation to the user's concepts. There are no practical refinement methods that can deal with this sort of situation yet.

Problems of Verification

While it is true that the mere act of writing a specification is a powerful tool for exposing errors, formal methods seem to offer much more: the possibility of rigorous analysis. In practice, complete analysis is simply infeasible on large specifications. Manual or tool-supported proof is inaccessible to all but a small priesthood. The other great white hope, model checking, is crippled by the state explosion problem. Huge progress is needed if these techniques are to be routinely applied.

On a more practical level, it is possible to derive good test cases from formal specifications, as noted earlier. The theory is well understood but there is a disappointing dearth of tools that will put this theory into practice, so test case generation remains a tedious manual process.

It is particularly disappointing that much effort going into validation is being devoted to the chimerical notion of 'proving programs correct'. Not only is this an impossible goal, it is addressing the least important part of the whole process — however successful this effort is, it will make negligible difference to the quality of delivered software, which is determined far earlier in the process.

V. Conclusion

I am an enthusiast for formal methods, and I can show that they offer clear benefits. However, these benefits are not automatic — they depend on intelligent application of methods where they can add value. There is no single best way of using formal methods, and no single best method. Furthermore, formal methods are only part of the solution to the software development problem and success depends crucially on integrating them into a larger process.

We do have excellent results from their intelligent use in a good process. We have systems that have few defects, that satisfy their users and meet the requirements of regulatory bodies.

There are, however, big challenges for researchers in making methods practical. Research needs to be informed by an understanding of what the real benefits are and in particular by where formal methods are best applied. Pragmatic issues of accessibility and integration are just as important as theoretical issues.

For practising engineers, though, there is a positive message: it is demonstrably possible to succeed with formal methods now.

Acknowledgment: The material in this paper has been adapted from an earlier version that appeared in *Formal Methods and Software Engineering*, Lecture Notes in Computer Science Vol. 3785, Lau and Banach (eds.), Springer, Berlin. The publisher's permission to reuse the material is gratefully acknowledged.

References

[1] Information about *Correctness by Construction* can be obtained from Praxis High Integrity Systems. `http://www.praxis-his.com`.

[2] J. Dick and A. Faivre, "Automating the Generation and Sequencing of Test Cases from Model-Based Specifications," *LNCS*, vol. 670, pp. 268–284, 1993, Springer.

[3] Standish Group, 1995, The Standish Group Chaos Report. `http://www.projectsmart.co.uk/docs/chaos_report.pdf`.

[4] C. Hoare, *Communicating Sequential Processes*. Prentice-Hall, 1985.

[5] J. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[6] S. King, J. Hammond, R. Chapman, and A. Pryor, "Is Proof more Cost-Effective than Testing?" *IEEE Transactions on Software Engineering*, vol. 26, pp. 675–686, 2000.

[7] A. Hall and C. R., "Correctness by Construction: Developing a Commercial Secure System," *IEEE Software*, vol. 19, pp. 18–25, 2002.

[8] See the *Retrenchment Homepage*. `http://www.cs.man.ac.uk/retrenchment`.

[9] W.-P. de Roever and K. Engelhardt, *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.