Formal Methods: More Than Just Proofs.

Danny House

Abstract

In this report we argue that formal methods are well on their way to becoming a routine part of every project. Industry appears to accept that verification via proof can be useful, but not enough to embraced widely. The theme of this report is that there is much more to gain from formal methods: specifically, support for reuse, implementation, and testing. By considering all of these uses taken together we perform a qualitative cost/benefit analysis of formal methods that uses compilers as the standard of comparison. We then propose a systematic integration of these uses to obtain a software development support system that would not be possible without formal methods.

1. Introduction

Formal methods are currently being used in many ways. Although none of them appear to be ready for prime time just now, taken together they will one day be famous. In this report we will argue that formal methods will inevitably become a routine part of every software project because they will provide far more than just proofs that requirements are being met. This argument, and the remainder of this report, is presented in four parts. The first part argues that the software industry is immature, and thus will not be swayed by the standard reasons for using formal methods. The second part presents an alternate qualitative cost/benefit analysis based on an analogy between formal methods and compilers. It contains the following four questions to which we propose immediate answers.

1. Do people need more training for formal methods than for compilers?
   We say they need the same amount and kind of training.

2. Can formal methods get to the point that they are as cheap to use as compilers?
   We say yes.

3. Can formal methods get to the point that they improve productivity as much as compilers?
   We say yes, in a relative sense that will be made clear later.

4. Can formal methods get to the point that they improve reliability as much as compilers?
   We say yes, in a relative sense that will be made clear later.

While justifying these claims, we describe some of the ways formal methods are currently being used to provide more that just proofs.

The third part proposes the construction of a software development support system that takes one or more formal specifications as input and provides test suites and implementations as output. Such a system will be based on the integration of formal methods techniques already in practice. Bringing the applications of formal methods together in one place can have as much impact as any one of the methods alone and perhaps more than any one of them.

Finally, the conclusion is drawn that some such system will inevitably be built. If we are right, then eventually everyone will be using them just like we currently use compilers.

2. The Software Industry Is Immature

There are still new products coming out all the time. There are still lots of money to be made just by being first. It is more important to add new features than to fix problems. Customers frequently don't really know what they want. With a market like this, mature organizations tend to be at a disadvantage.

None of the following would be true in a mature industry, where margins are low, products are treated as commodities, and change is incremental.

- The demand for practitioners is so high that many beginners have little or no training specifically in software - domain knowledge is often sufficient.

- There is little more compatibility between applications than there was between early railroad tracks, and the costs are just as great (though the impact is somewhat less tangible).

- We are plagued by software that fails badly, and even successful products frequently do not perform as intended.

- Far too many projects fail without producing any finished software at all (so we don't know if the software would have also failed).

- It is more common to hear that "there is a bug in the computer" than "your cheque is in the mail", and it is less likely to cause suspicion.

- Budgets and schedules are notoriously unreliable.

Perhaps immature is too gentle a term: this is an industry in its infancy.

The usual justification for using formal methods is that it improves quality by leading to the early detection of ambiguities and inconsistencies. With such a market, quality does not seem to be an issue. In the safety critical areas formal methods are much more accepted, but they cannot be said to be representative of the industry. There is also some evidence that formal methods improve productivity, but only a little when compared to typical budget and schedule overruns. In a climate like this, incremental improvements tend to be ignored – everyone is looking for a silver bullet (even when they claim to know better[1]). Worse, the improvement in overall productivity comes at a high cost in the early phases of a project, and must be amortized at least over the time to release of the software, and possibly into the maintenance phase. There is a real chance that the break-

---

1 The author is somewhat nervous that the reader will see this very report as asserting that formal methods provide a silver bullet.

even point lies after the project's cancellation date - especially if there is trouble convincing senior management that sufficient progress is being made.

3. An Alternative Qualitative Cost/Benefit Analysis

Here we draw an analogy between formal methods and compilers. More precisely, we might say that the notations of formal methods and the tools that are being developed to support them and build on them stand on one side of the analogy, while high level computer languages and their compilers and interpreters stand on the other. For brevity, we shall just refer to the two sides as 'formal methods' and 'compilers'.

For this cost/benefit analysis, we have identified four issues. On the cost side, we look at the availability of people in terms of how much training they need and how many can be trained. On the benefit side, we look at how new tools are currently having direct impact on real projects in terms of productivity and reliability.

3.1 Do People Need More Training for Formal Methods than for Compilers?

We claim that the same amount and kind of training is needed.

We will first list the skills needed to use a formal method, and then the skills needed for using a compiler. They appear in point form to make the comparison more visual.

What do we need to know to use a formal method?

1. We need to know the intuitive meaning of the notation. For example:
   When we say sorted, we mean (All i,j)(Bi<=Bj).
   When we say next, we mean AX (in the CTL formalism[2]).

2. We need to know the formal meaning of the notation. For example:
   What are the symbols and grammar of the notation?
   What are the valid modes of reasoning in the notation?
   What are the fundamental theorems of the notation?

3. We need to know the empirical data about the environment. For example:
   What objects will we model?
   What are their characteristics?
   What do we want to do with them?

What do we need to know to use a compiler?

---

[2] CTL (computation tree logic, [0]) allows us to say things like "no matter what happens next, the elevator door will be closed" formally as AX(doorClosed).

1. We must know what a program is. Thus we must know:
   How to map the abstract behavior of the computer into the concrete behavior of the world.
   How to express a task in the source language of the compiler.

2. We must know how to work with programs. For this we need to know:
   The language of the compiler.
   How to subdivide, generalize and fold tasks together (i.e., change a multi-pass task into a single-pass task) within the language of the compiler.
   The idioms of the language.

3. We must know what a program is good for. This requires we know:
   What the computer's I/O devices can do.
   How the computer carries out a task.
   The mapping the compiler uses to express the task in the language of the computer.
   How to estimate the cost for the computer to carry out a task.

These two lists look like they are describing the same skills: the second is just expressed more concretely than the other. The author's personal, unscientific experience has been that the people that could never learn formal methods can't write working programs in the languages they are paid to program in.

Many computer languages have been used successfully. There have been many successful projects that used very non-intuitive computer languages. The minds that can staff companies based on APL, Lisp or Prolog can certainly handle the concepts of formal methods.

It appears that the skills are the same, so the training should be the same.

3.2 Can Formal Methods Ever be as Cheap to Use as Compilers?

We claim that the costs can and will be quite similar.

There are two costs in using a compiler. One: buy a compiler. Two: pay someone to write software. For formal methods the costs are buying the support tools and hiring staff that can use them.

The capital cost of support tools for formal methods will only go down. It will take a while, but eventually this cost will be irrelevant compared to the cost of personnel.

From above, the same skills are needed for both, and so the same number of people with sufficient talent is available. Bauer and Parnas [1] provide support for this claim in the form of an example of formal methods being used by a fourth year undergraduate computer engineering student with no prior exposure to them. Hall [2] claims that *"people with only high-school math training can write excellent formal specifications"*, and that as little as three weeks training is enough to use the basic math and a specific formal notation. It would be worth the effort to get a better idea of just how large the talent pool is, but it does appear comparable to the pool of people capable of writing good programs.

As time goes by we can expect more and more of the newly trained to have some familiarity with formal methods. Once the skills start being demanded, they will spread very rapidly, and soon will not command an especially high price. It appears that the costs will be driven by the supply of trained people, and the supply of people trained in formal methods will eventually be the same as the supply of people trained in compilers.

3.3 Can Formal Methods Ever Improve Productivity as Much as Compilers?

We claim that they can and will.

By 'as much as compilers' we mean the percentage reduction of the time between initiation of a project and delivery of a finished product. Certainly, formal methods should not be expected to save the same absolute number of hours as compilers do.

There are at least four ways that formal methods can improve productivity. The most commonly cited is that getting the specifications right saves the cost of expensive re-work during the test and the integration phases. As noted above, this approach does not appear to save enough to interest industry. However, since it already pays for itself, any additional benefits come at little extra cost.

Formal methods can bring savings in both the implementation and testing phases, and can make software reuse easier and more effective. Each of these will be discussed below. In what follows we will see many ways for formal methods to improve productivity in each of these three areas. In each area, the gains are or will be quite large. If only half of these approaches work out, the total impact across the entire life-cycle appears to be at least as great as that of compilers.

3.3.1 Formal Methods in Implementation

If intermediate notations for low level designs can be directly compiled into code for a compiler, then a lot of intermediate work disappears. It appears that this is not just possible; it is being done now. C code can be generated directly from low level designs in B, SDL, ObjecTime, and other notations. Bicarregui et. al. describe a specific application of this in [5], where they use B. (B represents specifications as abstract state machines [3] [4].)

Any plan can be viewed as a type of code for a suitable machine, so a system that generates plans should be adaptable to the code generation task. In [6] Maghrabi and

Golshani demonstrate how a program can be generated in parallel to the steps in a proof in the sequent calculus. They only restrict the specification to being a well-formed formula of first-order logic. Their target language is a subset of Prolog. Coupled with a Prolog compiler we are nearly done. PROW [7] and QA3 [8] are resolution-based systems that also generate code. Lots of special purpose languages with corresponding code generators are written every year, and when they are well conceived, they reduce the software engineering task just a little more. When a special purpose language has a formal semantics, then we have automatic programming from a formal specification.

Currently, code from automatic generators tends to be slow and bulky. This was also true of early compilers. Today compilers yeild results that are usually superior to what can be done by hand. There is reason to expect that these techniques will improve significantly. Even a poor implementation that is automatically generated can give a big boost to productivity. Once a working model exists, it can be tuned, and sections of it can be rewritten to meet performance goals. Many parts of a system only need to run rarely, and must only meet loose time and memory restrictions. There are lots of ways in which formal methods can already improve productivity during implementation. More are appearing as we write this.

3.3.2 Formal Methods in Testing

Tools to support testing based on formal methods already exist. If they only do as well as humans, then some of the most unpleasant and costly work associated with any project can be eliminated. Not only will this yield direct time savings, it will improve employee morale and reduce staff turnover, improving productivity in an indirect manner. An additional benefit comes during maintenance, as these tools reduce the cost of regression testing [9]. In fact, as described in the section on formal methods and reliability, existing tools provide better coverage and reliability than humans in some cases.

Tools for generating test cases already exist. In [10] test kernels were computed using TVeda-V3, which uses both symbolic computation and reachability analysis. In [5] the test cases were generated from VDM using VAT [11]. (Being a prototype research tool, a 1400 line specification was enough to make VAT 'struggle'. It is not clear how well it will scale up.) In [9], the test class generation was supported by DST Z-Tools. Some other tools of this sort are TAOS and SDL.

Even when direct generation of tests is still too difficult, the use of a test oracle or test checker will improve productivity. It can require quite a bit of work to determine the correct behavior in response to a given test, and in a non-deterministic test there is no unique correct behavior. A reference implementation obtained from animating a specification or from an inefficient code generator can be used to predict correct test results. Even without such an oracle, a test can be run, and then the test and the result together can be given as input to a checker that indicates if they satisfy the relationship defined by the specification. Tools for generating test checkers already exist; for example, a predicate compiler for Z is described in [12]. Certainly, these tools can still be improved a lot, but the gains in productivity are already real, and can only increase.

3.3.3 Formal Methods in Software Reuse

If reuse can be made easier then designs can be created quicker. To reuse existing software engineering products, the developer must perform several tasks: analyze the problem, find objects worth reusing, understand the objects found, select the best objects, adapt the objects, justify the choices made, evaluate the result, and record the choices and adaptations [13]. Formal methods can contribute significantly to all of these except recording the choices and adaptations (which should be straight forward).

Large libraries of reusable components already exist. Nortell has a library of about 16 million lines of code [14] and IBM has a library of more than 1,200 components [15]. In either choice of units, that is a lot for a designer to learn. In [15], it is suggested that these components should be classified by hand using formal methods, so that we look for objects worth reusing by searching for specifications that unify with one or more goals in the problem. There is a further suggestion in [15] as to how the search might be performed efficiently.

As an aside, there are problems with how the components are to be classified. In [15], the provider of each component is required to anticipate the way in which the component will be used, then specify the properties that correspond to the intended use. This seems to add opportunities for error while expecting the provider to have oracular power. If formal methods are used throughout, then the components available for reuse must each be formally specified. In this case the classification should come for free. This begs the question: which formal notation should be used? As we shall see, using just one notation would help in many ways.

In [13], problem analysis means choosing descriptors as a basis for search. This includes the map from the notation of the problem to the notation of the repository. Ideally, there should be a straight forward map between these notations. If not, the reuse repository may be rendered useless due to the potential for mistranslation.

After a list of objects that meet the search criteria has been found, the ‘best’ must be selected. Evaluating them will require understanding how they behave in areas not specified during the search, and what trade-offs must be accepted to use them. Understanding of this sort would be enormously facilitated by a common notation.

There are several ways to adapt an existing object to a new use. The most common way is to place a wrapper around it. Wrappers introduce great inefficiency when it takes a lot of work to translate between representations. Another common method appears in object oriented reuse, where a generic object is adapted by having it act on concrete objects which must provide a specific interface required by the generic object. This approach has problems with efficiency when the generic object is something like an entry in a linked list, which points to the concrete object that holds the data - there is no direct path from the concrete object to the next entry in the list, so an extra level of indirection appears. Yet another way is to rewrite and optimize the object for its new chore. Rewriting and optimizing gives efficient code, but is expensive in human time and suffers from a greater risk of human error.

[16] takes an approach to adaptation that is half way between these last two. The generic object is specified using a high level description that allows the compiler to rewrite it. The concrete object is first specified via an abstract object called a view which defines an interface the concrete object must implement. Then a compiler rewrites the generic object with the concrete object folded into it.

This is an application of formal methods that has been explored much less, so its final impact on productivity is not clear. It looks very interesting, and can be viewed as a 'free' extra, once formal methods become commonplace.

3.4 Can Formal Methods Ever Improve Reliability as Much as Compilers?

We claim that they can and will.

By 'as much as compilers' we mean the percentage reduction of the total cost due to bugs in products delivered to customers. It would be unreasonable to make the comparison based on absolute numbers since humans make more slips than mistakes, and compilers get credit for fixing many more slips (i.e., typos, mismatched types, etc). Note: the same types of slips occur in formal specifications, and tools like type-checkers and automatic code generators will improve the quality of the formal specifications in the same way. For a given project, the size of the formal specification is much smaller than the size of the source code in a traditional language, which in turn is much smaller than the size of the source in an assembly language, and the number of slips in each will be proportionate to the size. Thus formal methods cannot compete in an absolute sense.

Improved reliability can come in several forms. It is generally agreed that formal specifications catch mistakes both in understanding what the customer wants, and in designs (especially if proofs are used). Proofs of conformance between specifications and annotated implementations have also been explored [17].

Gains can also come from better test suites. Bicarregui et. al. [5] report better coverage through automated test domain generation. A common error in writing tests is to choose input that will not trigger the problem being sought (a trivial example would be to write a test for the well-known error in division in early pentium chips, but by making a small slip, only specifying inputs on which the chips work). Fernandez et. al. [10] compared their generated tests to a suite of pre-existing hand written tests addressing the same features. Both the hand-written and machine-generated tests were expressed in the same test scripting language. There was a direct correspondence between the two suites: each hand-written test had a matching machine-generated test that was expected to exercise the same behaviors of the system. When the corresponding hand written and machine generated tests were compared, Fernandez et. al. found that they had very similar structures. The most interesting finding in the comparison was that a few of the hand written tests were invalid: they simply did not test what they were supposed to test.

An important part of concocting a test by hand is to state the behavior that should be observed. This can be quite difficult and error prone. By generating a test oracle or reference implementation, or by animating a specification, errors in the prediction of behavior can be eliminated. When a test result checker can be generated, the opportunity for error disappears along with the need for a prediction.

When buying to solve a problem, generated tests can help to avoid the purchase of products that are unreliable. They can also help in choosing the right product (which will also improve productivity). Such tests can help to identify where a product already meets the needs of a project, and where work must be done to make it fit. As formal methods take root, vendors will find they must give formal specifications of their products. This will make it much easier to know what is being bought and will improve perceived reliability. It will also cause unreliable products to leave the market much quicker.

Automated code generation from low level formal specifications will help improve reliability even without support for verification that the low level design meets the higher level designs or the requirements. We still eliminate the problems that are introduced during the final implementation. When a problem is not discovered until testing (or later), it is generally most expensive to fix when it was introduced into the system during requirements specification, somewhat less expensive if it was introduced during the high level design, still less expensive if it was introduced during the low level design, and even less costly if it was introduced during the final implementation. Unfortunately, this repair cost has no relation to the effect that the problem may have on a user. An airplane builder may be able to fix a typo made when writing the low level code at relatively little cost, but the family of a dead airplane passenger won't feel any less loss.

Since formal methods can make so many contributions to reliability, we conclude that the comparison to compilers is viable.

4. Proposal: a Formal Methods Based Software Development Support System.

The goal is to build a development system that will ensure that the product meets the requirements, rather than a verification system that will check if a completed product does. It is assumed that one formal notation has been chosen to tie it all together. It should be possible to construct a suite of tools based on the chosen notation that will:

1. generate test cases,
2. come with a rich set of reusable objects, each of having a formal specification in the chosen notation,

3. support the search for reusable objects via the use of unification and the algorithms described in [15], and a natural user interface to restrict the search to objects that unify with a user specified sub-problem,
4. adapt reusable objects as described in [16],
5. accept the introduction of new objects, so long as they have specifications in the chosen notation,
6. prevent the assembly of components that are not compatible (design-time type checking),
7. track requirements of the system being developed and how the design refines them,
8. generate the final code, and
9. run the test cases against the final implementation.

The tools for the various parts of such a system have already appeared in the literature, lacking at most a common notation to bring them together. Given a common notation, this should be possible, and the system could be said to support a mode of design by refinement. Which formal notation should we use? We like the second order logic, but others can work, too. The important thing will be to settle on just one or a very small list in order to unify the support system.

5. Conclusion

Formal methods are in use, but not widely. We have seen that the software industry is not yet ready to accept formal methods based on incremental gains. Formal methods are already providing significant benefits in several areas of system building. By considering these applications together and comparing formal methods to compilers, we have shown that the gains can be far more than just incremental. As the notations of the formal methods tools begin to converge, the tools will work better together and the cost/benefit ratio will improve. We have suggested a specific software development support system that can only be realized by integrating the tools. Such a system or variations on this theme will provide a strong motivation to get a convergence of notations.

We feel that it is safe to predict that eventually everyone will use formal methods and will wonder what all the fuss was about. (Note: it is worthwhile to remember that when compilers first hit the scene, implementers objected that they could write tighter code by hand.)

6. References

[0] E. M. Clarke, E. A. Emerson and A. P. Sistla, “Automatic Verification of Finite-State Concurrent systems Using Temporal Logic Specifications”, ACM Transactions on Programming Languages and Systems, vol. 8, no. 2, pp. 244-263, April 1986.

[1] Brian James Bauer and David Lorge Parnas, "Applying Mathematical Software Documentation: an Experience Report", COMPASS 95, Proceedings of the Tenth Annual Conference on Computer Assurance, IEEE, New York, NY, USA, 1995, ix+285 pp. p273-285.

[2] Anthony Hall, "Seven Myths of Formal Methods", IEEE Software, September 1990, Vol. 7, No 5, pages 11-19.

[3] J.-R. Abrial, "The B-Book: Assigning Programs to Meaning”, Cambridge University Press, Cambridge, 1996.

[4] B-Core (UK) Ltd, B-Toolkit User's Manual, Version 3.0., 1996. For details, contact Ib Sorensen, B Core (UK) Ltd, Magdalen Centre, Robert Robinson Avenue, The Oxford Science park, Oxford 0X4 4GA. Tel.: +44 865 784520. E-mail: Ib.Sorensen@comlab.ox.ac.uk. WWW: http://www.b-core.com/.

[5] Juan Bicarregui, Jeremy Dick, Brian Matthews, Eoin Woods, "Making the most of formal specification through animation, testing and proof", Science of Computer Programming, vol. 29, no. 1-2, pp. 53-78, July 1997.

[6] Talal Maghrabi and Forouzan Golshani, "Automatic Program Generation Using Sequent Calculus", Proceedings of the 1992 ACM Computer Science Conference, xviii + 574pp, p73-81, 1992.

[7] R. Waldinger and R. Lee, "PROW: A Step Toward Automatic Program Writing", Proceedings of IJCAI, pp. 241-252, 1969.

[8] C. Green, "Application of Theorem Proving to Problem Solving", Proceedings of IJCAI, pp. 219-239, 1969.

[9] Hans-Martin Horcher, "Improving Software Tests using Z Specifications", in J.P. Bowen and M.G. Hinchey, eds,, Proc. ZUM'95, Lecture Notes in Computer Science 967, (Springer, Berlin 1995).

[10] Jean-Claude Fernandez, Claude Jard, Thierry Jeron, and Cesar Viho, "An experiment in automatic generation of test suites for protocols with verification technology", Science of Computer Programming, vol. 29, no. 1-2, pp.123-146, July 1997.

[11] A.J.J. Dick and A. Faivre, "Automating the generation and sequencing of test cases from model-based specifications", in: J.C.P. Woodcock and P.G. Larsen, eds., "Proceedings of FME'93: Industrial-Strength Formal Methods", lecture Notes in Computer Science, Vol. 670 (Springer, Berlin, 1993) 268-284.

[12] Erich Mikk, "Compilation of Z Specifications into C for Automatic Test Result Evaluation", in J.P. Bowen and M.G. Hinchey, eds,, Proc. ZUM'95, Lecture Notes in Computer Science 967, (Springer, Berlin 1995).

[13] Arun Sen, "The Role of Opportunism in the Software Design Reuse Process", IEEE Transactions on Software Engineering, vol. 23, no. 7, pp. 418-436, July 1997.

[14] P. Cashin, "BNR Remains at the Forefront of Computing Technology", Telesis, vol. 92, pp. 73-75, 1991.

[15] Rym Mili, Ali Mili, and Roland T. Mittermeir, "Storing and Retrieving Software Components: A Refinement Based System", IEEE Transactions on Software Engineering vol. 23, no. 7, pp. 445-460, July 1997.

[16] Gordon S. Novak Jr., "Software Reuse by Specialization of Generic Procedures through Views", IEEE Transactions on Software Engineering, vol. 23, no. 7, pp. 401-417, July 1997.

[17] Marsha Chechik and John Gannon, "Automatic Analysis of Consistency between Implementations and Requirements: A Case Study", COMPASS 95, Proceedings of the Tenth Annual Conference on Computer Assurance, IEEE, New York, NY, USA, 1995, ix+285 pp. 123-31.