The Industrial Use of Formal Methods: Was Darwin Right?

Steven P. Miller
Rockwell Collins
400 Collins Road NE
Cedar Rapids, Iowa 52498
spmiller@collins.rockwell.com

Abstract

Even though the use of formal methods in industry has been documented in numerous case studies, skepticism about their usefulness remains widespread. However, formalisms have evolved rapidly over the last decade and are doing a much better job of meeting the needs of industry. This paper briefly describes several of the experiments in formal methods that have been conducted at Rockwell Collins and attempts to pull these observations together into a profile of what industry needs from the research community.

1. Introduction

Even though the use of formal methods in industry has been documented in numerous case studies [6], [8], skepticism about their usefulness remains widespread. Common complaints are that they do not scale to industrial problems, are too expensive, or require skills and training beyond those available in the workforce.

We have explored the use of formal methods at Rockwell Collins for several years, and while not every experiment has been a success, I find I'm actually more positive about the future of formal methods now than at any time in the past. In part, this is because the needs that drove us to consider formal methods, particularly the need to produce safety-critical systems that perform as expected, are still with us [7], [21]. In fact, with the rapid growth of digital systems, these needs are greater than ever, and formalisms of one sort or another remain the only real solution.

Moreover, things are changing. Even a few years ago, formal verification of hardware was considered a promising research topic, but not yet ready for industrial use. Today, formal methods, particularly model checking, has won widespread use within the hardware industry.

Much of the reason for this new found acceptance is due to a rapid evolution in the types of formal methods available over the last decade, many of which are doing a much better job of meeting the needs of industry. I believe that this evolution will continue, and that there will be a process of natural selection. Moreover, the notations and methods that survive will be the ones that industry accepts, simply because of the vastly greater resources available to industry.

But which methods and notations will succeed? To try to answer that question, this paper briefly describes several of the experiments in formal methods that we've conducted at Rockwell Collins and what we've learned from each one. At the end of the paper, I'll attempt to pull these observations together into a profile of what industry needs from the research community.

2. Specification of the μRTE in RAISE

Much of our interest in formal methods can be traced to our participation in the MCC Formal Methods Transition Study in 1990 and 1991 [12]. In particular, it was there that we learned of the RAISE method and tools developed in Europe as part of the Espirit project [11]. RAISE is certainly one of the most technically impressive formal notations I've seen. It integrates concepts from VDM, OBJ, and CSP in a seamless notation. However, at the time we looked at RAISE it provided very few tools to support formal verification.

As one of our earliest formal methods experiments, we undertook specifying a micro Real Time Executive (μRTE) in the RAISE notation, RSL. To improve performance, the μRTE was to be implemented largely in microcode. While very simple, it supported cyclic and acyclic task scheduling and standard synchronization primitives such as mutex lock and release. We also tried to specify the μRTE at two levels of abstraction and do in-

formal proofs of correctness to show that the more detailed level implemented the more abstract specification.

While we did get the μRTE specified in RAISE, it wasn't all that successful a project. Some of our engineers liked the RAISE language, a few thought it was too complex, but most just didn't see the value in the exercise. While I was impressed by the sophistication of the language, I was also concerned about its complexity. It would have taken a great deal of training to ensure our engineers used it correctly. Finally, the analysis tools that RAISE provided, at least at that time, were limited to syntax and type checking. Formal verification tools were planned, but weren't yet available. As a result, we never had much confidence in our informal proofs of correctness. Eventually, this combination of a complex syntax, the need for extensive training, and a lack of analysis tools led us to put RAISE on the shelf.

2.1 Verification of the AAMP5 Microcode

We followed the μRTE with one of our largest and best known experiments in formal methods, the formal verification of the microcode in the AAMP5 microprocessor [22], [28], [29], [30]. This project was sponsored by Rockwell Collins and by NASA Langley, and was conducted by SRI International and Rockwell Collins. The goal of this project was to assess the feasibility of using formal verification with PVS [27] to check the microcode in the AAMP5 microprocessor.

The AAMP5 is a Rockwell proprietary microprocessor that is widely used in Collins products. Like all AAMP processors, it has a stack-based architecture, a large CICS-like instruction set, and makes extensive use of microcode [1]. It differs from earlier AAMP processors in that it is pipelined and contains complex processsing units such as a LFU (Look Ahead Fetch) unit. It contains approximately 500,000 transistors and provides performance somewhere between an Intel 386 and 486.

The approach taken on both the AAMP5 and AAMP-FV projects is illustrated in Figure 1. We specified the AAMP5 at both the micro-architecture, or register transfer, level and at the macro-architecture, or instruction set, level. We also defined an abstraction function mapping the states of the micro-architecture to the states of the macro-architecture.

Proving the microcode of an instruction F correct consisted of showing that the diagram in Figure 1 commuted, i.e., that starting in micro-state s_0 and applying micro-instructions f_1 through f_N followed by the abstraction function resulted in the same macro-state S_N as by first applying the abstraction function followed by the macro-instruction F. The main lesson we learned from the AAMP5 project was that it was technically possible to prove the correctness of microcode. We specified over half of the AAMP5's instructions in PVS, we specified the AAMP5 design at the register transfer level in PVS, and we proved the correctness of 11 instructions.

We also learned that engineers can read and write formal specifications. While the simplicity of the PVS language played an important role in this, reading and writing PVS was well within their abilities. Anyone that can be taught a modern programming language can be taught a formal specification language. The issue is one of motivation, not ability.

However, our engineers never abandoned their domain specific, graphical notations, and that is still how they specify processors today. What this demonstrates is that these presentations provide value that is lacking in a purely textual language.

Looking back, one minor problem was the lack of state

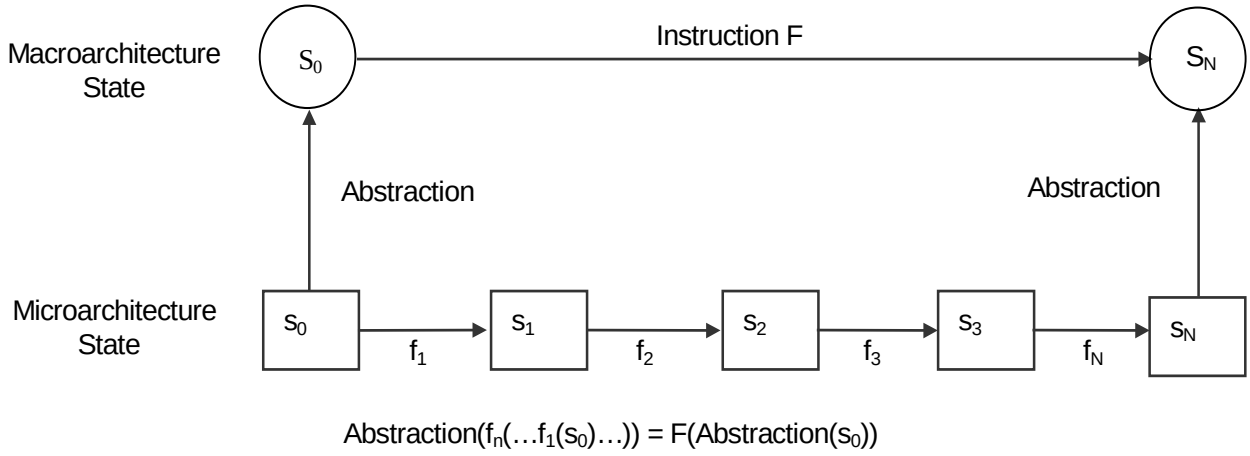

Figure 1 - Approach to Microcode Verification

variables in PVS. Representing state was always something we had to work around, and that made the specifications more difficult to read. While the presence of state variables can indeed make proofs more difficult, their absence also makes the specifications less intuitive, throwing up yet another barrier to acceptance.

We did find two actual errors in the AAMP5 microcode while creating the specification, and this convinced us that there is value in just writing a formal specification.

We also convinced ourselves that formal verification, i.e., mechanical proofs of correctness, do provide a very high level of assurance. We did this by seeding two very subtle errors in the microcode we delivered to SRI, and then waiting to see if they would find them. SRI did indeed discover them, and what impressed us was the fact that this was a very systematic process - there was no way that they would have not found those errors except to have not done the proofs.

The other thing we learned was the importance of tackling large, realistic problems. The reason for this is that one simply won't confront issues of scale and complexity until forced to. On the AAMP5 project, some of our biggest problems were how to organize the specification, how abstract away from irrelevant details, how to structure complex proofs, and how to get more speed out of PVS.

The biggest problem with the AAMP5 project was that the cost was too high, about 308 hours per instruction. We knew this figure was greatly inflated for a variety of reasons. We were on a very steep learning curve, PVS was evolving as we were using it, we had to develop many supporting theories, and rather than trying to verify the largest number of instructions, we tried to verify instructions from a variety of classes. We knew our costs would drop dramatically the next time around, but we couldn't predict by how much.

3. Verification of the AAMP-FV Microcode

To answer this question, we undertook a second experiment, verification of the microcode in the AAMP-FV [25]. Just as with the AAMP5, this project was sponsored by NASA and Rockwell Collins and conducted by Rockwell Collins and SRI. The goal was to demonstrate a dramatic reduction in cost through reuse of the AAMP5 infrastructure and expertise.

The AAMP-FV is a paper and pencil design for a processor specifically intended for use in ultra-critical applications such as auto-land and fly-by-wire. For this reason, it is simpler than the AAMP5. It has about 80 instructions and is not pipelined. However, is not a toy design. We estimate that if fabricated, it would consist of approximately 100,000 transistors and have performance comparable to an Intel 386.

Somewhat to our surprise, the verification of an AAMP-FV instruction was roughly comparable in difficulty to the verification of an AAMP5 instruction. This was because we did not make as much use of abstraction in specifying the micro-architecture of the AAMP-FV as we did on the AAMP5.

Even so, we did manage to achieve a dramatic reduction in cost for most of the AAMP-FV instructions. We completed the verification of 54 of the 80 AAMP-FV instructions at a cost of about 38 hours per instruction, approximately one eighth the cost on the AAMP5 project. Moreover, we estimate that we could get this down to twenty, perhaps even ten, hours per instruction on a similar processor. The key point is that we managed to get the verification of these 54 instructions down to a very well defined, repeatable process. That sort of predictability is very desirable in industry.

Unfortunately, not all the AAMP-FV instructions fit this process. About twenty instructions were substantially more complex, and our proof strategies broke down on these. This forced us back onto a research track for these instructions, with considerably higher costs.

One of the main lessons we learned on the AAMP-FV was that each new domain has a steep learning curve, and that costs will drop dramatically on the second or third attempt. Unfortunately, this also means that we can't draw any meaningful estimates, other than worst case behavior, from our first trials.

Another important lesson was that productivity requires expertise in both the problem domain and the methodology. Our productivity didn't really take off until the domain experts were experienced enough with PVS that they were writing the specifications and doing the proofs. Productivity requires that the same individuals master both sets of knowledge.

For the first time, the robustness of the proofs became a concern for us. We often found ourselves revising the specifications to make the proofs simpler to complete. Unfortunately, this often broke proofs that had been completed months before. The AAMP5 and the AAMP-FV were unusual projects in that they were very stable. In contrast, the norm in industry is constant change, and it is essential that any automated analyses be robust enough to accommodate these changes.

In many ways, the AAMP-FV project was a great success. However, it was conducted by a small number of highly trained engineers, and we were gradually coming to realize that thee widespread use of formal methods would require more intuitive notations, perhaps even notations tailored to a specific domain. Even with such notations, we also knew there would need for more emphasis on method than on notation.

4. Specification of the FGS in CoRE

At the same time, we also decided to focus on the modeling and analysis of requirements. Microcode had been a good vehicle for us to explore formal proofs of correctness, but we knew that the most important errors are made during requirements definition. Also, we expected requirement models to be smaller and more tractable than design models.

We started by looking at the CoRE method from the Software Productivity Consortium, of which Rockwell Collins was a full member [9], [10]. CoRE is based on Parnas' four-variable model, and one of the things that we liked was that it was a true methodology, not just a notation. A CoRE specification consists of a single model of the requirements, but it has two aspects. The behavioral model specifies the functionality of the system. The class model packages the behavioral model, providing a way to structure large specifications and to plan for and manage change.

To validate the CoRE method, we applied it to an example suggested by our product areas, the mode control logic of a Flight Guidance System [23], [23]. The mode control logic is one of the more complex parts of a Flight Control System, and to keep our example tractable, we had to make some simplifications. Specifically, we eliminated some of the more complicated modes of operation, we didn't specify the hardware interfaces, and we didn't deal with failures of internal components. Even so, it is a large example, and we view it as a realistic industrial case study. It took us 560 hour over nine months to complete. It has been approved for external release, and there are several researchers currently working with it.

Since we had no tools to use in creating the CoRE specification, we validated it through a series of inspections with experts in flight control and some of our software engineers. These inspections only took a total of 35 person hours, but they found close to 100 defects in a specification that we had worked long and hard on. It took us over 300 hours to correct these defects.

This points out the effectiveness of inspections. They're very cheap, find many defects, and are an excellent way to educate new team members. Even when we did proofs of correctness of microcode, we found inspections to be one of our most useful techniques. So one obvious lesson is that formal notations and tools should support inspections. However, modeling tools often fail to provide good ways to print the model. In extreme cases, we've actually had to conduct inspections of screen dumps. So while obvious, the importance of inspections in industrial practice is not always understood.

The main thing we learned from this exercise was that requirements can be stated as formal, mathematical models. Since the current state of practice is to specify requirements in English with a few tables and diagrams, this is a big change, and one that I don't expect to be accepted quickly.

CoRE emphasizes architectural issues with its class model. One of the things this made clear to us was that architecture spans requirements, design, code, and test. Much of the system architecture is defined by the problem you're working on, and one of the biggest challenges is figuring out what that architecture is. During the inspections, one engineer complained that while he could read the CoRE specification just fine, it wasn't helping him to see the real problems. Much later, we realized that what he was trying to tell us, but couldn't quite articulate, was that we had the architecture wrong. It has taken considerable effort to correct that mistake, and we could have saved ourselves a great deal of trouble if we had identified the right architecture sooner. Eventually, we came to view the CoRE class model as essential as the behavioral model itself. In fact, we found that we needed more architectural constructs, things such as replication, parameterization, and even inheritance.

We have also found the four-variable model to be an extremely useful paradigm that we kept coming back to. Time and time again, we found ourselves writing it on the whiteboards to discuss some fine point of a specification.

We learned, once again, that engineers prefer graphical notations. Graphical notations are just simpler to read than text and help to make the formalisms more palatable.

While CoRE is based on a formal model, it lacks a formal syntax and semantics. Looking back, it is amazing that we got as far as we did without a formal semantics. This illustrates the importance of the many "informal" aspects of software design, such as the architectural issues discussed earlier. However, we eventually did reach a point where everyone agreed we needed both a formal syntax and semantics.

Finally, our lack of tools was both a weakness and a strength. Doing the syntax and type checking by hand was very tedious and error prone, but not using a tool gave us the freedom to improvise when necessary. Committing to a tool also implies committing to all the constraints imposed by that tool, so its very important to have a clear understanding of what you're trying to accomplish and to be sure the tool supports those goals. Not using any tool is usually better than using the wrong tool.

5. Specification of the FGS in SCR

About this time we received a copy of the SCR tool from the Naval Research Lab [15], [16]. SCR and CoRE both grew out of work on the A-7E aircraft and are very similar. While SCR lacks the class structuring found in CoRE, it does have a formal syntax and semantics that allows the SCR tool support a variety of automated analy-

ses. Also, SCR models can be executed using the simulator provided with the SCR tool.

Because of the similarity of SCR and CoRE, it was straightforward to enter most of the CoRE specification of the FGS into the SCR tool. This only took us a little over 100 hours. Because of the effort we had invested in inspecting and reviewing the CoRE specification, we didn't expect to find many errors in it. To our surprise, we found 27 errors in the original CoRE model, some of them significant. A breakdown of these errors is shown in Table 1.

Trivial errors are spelling and punctuation errors. Minor errors are those that would definitely be caught during implementation. Moderate errors are those that would probably be caught during development, but could be missed. Major errors are the subtle mistakes that probably would not be caught during implementation and would have a significant effect on the final product.

While care has to be taken in drawing too many conclusions from a sample of only 27 errors, there are a few trends suggested by this data. First, simply entering the example into the tool found the most errors, but these tended to be minor or moderate errors. The errors found by the SCR analysis tools were more significant, which is what we expected. A surprise was the seven errors found by simply executing the SCR model in an ad hoc fashion. One of these was a minor error. Two would have been found by the analysis tools, except that the analyses took too long and we turned them off before they were completed. Three were misunderstandings on our part of the semantics of SCR, and one was a significant mistake on our part.

The obvious lesson here is that automated analyses will find errors that that will be missed with manual methods such as inspections. Other researchers have found the same thing [14], [16], and we can probably accept this without controversy. On the other hand, I wouldn't be willing to give up inspections. They're still needed for things tools don't do well, such as checking for issues of style and maintainability.

It was more of a surprise that simulation also found several errors, particularly subtle misunderstandings of semantics. While this reinforces the value of simulation as a validation technique, it is also a bit discouraging. Simulation suffers from the same problem as testing in that it doesn't have a well defined stopping criterion, i.e., there is no easy way to know when we've done enough.

However, the biggest surprise was still to come. We connected the simulation to a mock-up of the flight deck. With this, people could push the buttons and watch the lights turn on, and at the same time watch the requirements model execute. In effect, our requirements model became a rapid prototype. The effect was almost immediate acceptance by the engineers, and this may be the most important lesson of all. Connecting an executable model to a mock-up of the user interface makes the formalism accessible, both to the engineers and the customers.

6. Generating Test Cases with T-VEC

The FGS mode logic specification was also used to evaluate the T-VEC test vector generator. T-VEC is a testing environment that generates test cases and their outcomes from a formal specification of the functional behavior of a system. It also checks for inconsistencies and guards in the specification that cannot be tested [3], [4].

For this experiment, an SCR to T-VEC translator developed by the Software Productivity Consortium was used to generate a T-VEC test specification from an SCR specification of the mode logic. This automatically generated T-VEC specification was combined with hand written T-VEC specifications describing the mapping of low level software input and output variables to the more abstract SCR variables. The entire T-VEC specification was then used to generate test cases and the coverage of these tests assessed on a Java implementation of the mode logic. While many issues remain to be investigated, these early

Table 1 - Errors Found Using SCR

Found By	Severity Of Defect				Total
	Trival	Minor	Moderate	Major	
Creating SCR Model		7	4	1	12
SCR Analysis Tools			2	2	4
Executing SCR Model		1	3	3	7
Other Tools	2	1	1		4
Total	2	9	10	6	27

tests achieved a high level of multiple condition coverage of the Java code. The T-VEC tool also identified several errors in the SCR model and in the Java code.

7. Detecting Mode Confusion

This last year, NASA has sponsored another experiment in formal methods. The goal of this project was to determine if formal methods can be used to detect sources of mode confusion in an automated system. Mode confusion is a situation in which the human operator either becomes confused about the current state of the automation or has an incorrect mental model of how it behaves [19], [20]. Mode confusion has been cited as playing an important role in several aircraft accidents [2], [17].

We know we can specify the behavior of the automation formally. The real question is what sources of mode confusion can be characterized formally so that we can search for them with automated tools. For example, one possible question is "Are there are any states of the system in which we ignore a direct command from the flight crew?"

In this project, we specified the mode logic of the Flight Guidance System in PVS. However, this time we changed the architecture of the mode logic to better support an entire family of Flight Guidance Systems. This resulted in many small, cohesive components, and is very similar to the architecture one would get by applying an object oriented approach. We believe this architecture is far superior to that developed in the original CoRE specification.

However, one result was that there were a large number of system constraints between these components that had to be maintained. Since we were using a formal specification system, it was natural for us to state these as lemmas that could be proved with PVS. The proofs were quite simple, usually requiring only a few PVS commands. Before long, we had developed an entire suite of proofs and fell into a pattern of rerunning all our proofs after each significant change to the model. More often than not, a few proofs failed, indicating something we had gotten wrong.

This turned out to be very helpful in building our models. Our product areas routinely run regression tests after making a change. In a similar fashion, because we had automated some of the verifications we would normally do manually, it was possible for us to easily and cheaply rerun regressions suites of these analyses.

The biggest surprise on this project was that we found ourselves doing proofs not because we were trying to assess their utility or demonstrate a point, but because it was helping us to get the job done. One of the ongoing debates within the formal methods community is whether engineers will do proofs. After this project, it seems clear that proofs will naturally follow the creation of models, providing the proof system is integrated with the modeling environment and is powerful enough that the proofs are not too onerous to construct.

The second thing we learned is that object oriented methods will actually increase the need for formal analysis. The reason is that object oriented specifications consist of many, small, cohesive components, and this increases the number of relationships between components that need to be maintained.

Finally, we did have some success automating the detection of certain sources of mode confusion in our models. The Phase I report should be available from NASA shortly [26], and we hope to extend this work next year.

8. Formal Specification of the JEM Microprocessor

The last experiment to be discussed is an excellent example of how to insert formal methods into an existing industrial process. The JEM microprocessor was created last year by our microprocessor group. It is the world's first direct execution Java Virtual Machine, directly executing Java byte-code in hardware [13].

Two of our engineers have created a formal specification of the JEM micro-architecture in ACL2 [5] that executes JEM micro-code [31]. More importantly, they've managed to optimize the ACL2 specification so that it executes faster than the original C code in the JEM simulator. In fact, they have replaced the C code in the simulator with the ACL2 version and shown that they get the same outcomes running the JEM test suites. The significance of this is that they've found a way to substitute a formal model for a model we already build at no extra cost. Now that they have the formal model, doing proofs follows naturally, and they have already proved several important properties of the JEM micro-architecture.

9. Needs of Industry

This summarizes many of our experiments with formal methods over the last few years. This section tries to pull these together into a profile of what we believe industry needs from the formal methods community.

9.1 Executable Specifications

Our top requirement is for executable specifications. There has been considerable debate within the formal methods community whether specifications should be executable, the concern being that executable specifications are too operational and insufficiently abstract. But our experiences with SCR have convinced us that we can write useful executable specifications, and we have just had too

much success with executable models to not pursue this. By integrating an executable model with a simulation of the user interface, we get a rapid prototype early in the lifecycle that can be used to review the requirements with our customers. One caveat here is that speed can be important. For example, if the ACL2 specification of the JEM ran only half as fast as the C code, it would not have been a viable option.

9.2 Clarity and Readability

Our next most important need is for specifications that are clear and readable. After all these years, this remains the Achilles' heel of formal methods. We need better ways to document the forest before the trees, we need to integrate graphical and textual notations, we need intuitive ways to deal with state, and our notations need to be as simple as possible. In some cases, we may need notations specifically developed for particular domains.

9.3 Support for Scale, Change, and Reuse

We also need to be able to address issues of scale, change, and reuse. We need mechanisms to organize large specifications so that they are readable. The one constant we can count on in industry is change, and we need to be able to plan for and manage change. Since we usually build variations of the same systems over and over, we would like to achieve massive reuse of those specifications by tailoring them for each new product.

9.4 Methodologies

Too often, we're offered notations and tools without the methods we need to use them effectively. We need to understand the fundamental concepts that make an approach work and the steps we need to follow to apply those concepts. It would also help to have some industrial case studies to serve as examples.

9.5 Compatibility With Existing Practices

One of the biggest obstacles to inserting change into industry is incompatibility with existing practices. We have hundreds of practices we follow in producing a product, and we can seldom change more than a few on any given project. For example, it is very difficult to incorporate a large, monolithic tool into our current practices. But if that same tool can be broken down into its components, our engineers can probably figure out ways to use those pieces that haven't even been thought of.

We also need to use the same models for development, execution, and analysis. There just isn't time to create one model for specification, another for simulation, and yet another for verification. Object oriented paradigms seem to be gaining in popularity, and this could provide a path for injecting formal methods into industry. In particular, I'd like to see more work done on assigning a formal semantics to the Unified Modeling Language (UML) appropriate for real-time embedded systems.

9.6 Analysis With Automated Tools

With the possible exception of executable specification, none of the needs discussed so far require a formal semantics. This reflects the fact that so many of our needs are not amenable to formal methods. For this very reason, I expect a large number of tools with executable models, but without well defined semantics, make deep inroads into industry over the next few years. While I expect these to be very successful, industry needs to understand that in some respects, they are a dead end.

The reason is that the next step beyond executable specifications is analysis with automated tools. We need to be able to do simple consistency and completeness checks and to demonstrate application specific properties, particularly safety properties such as freedom from hazardous states. One thing we've learned from our friends as SRI and the NRL is that this requires discipline in the notation. The simpler and more carefully thought out the notation, the further you are able to push analysis.

9.7 Support for Testing

It is also important not to lose sight of testing. Currently, we spend between one to two thirds of our entire project budget on testing. Automatically generating test cases from a formal model could almost justify the cost of building the model. It would be even better if we could figure out how to replace some of that testing by analysis.

9.8 Proving Properties

One of the ongoing debates within the formal methods community is whether proofs will ever be routinely done by real engineers, and at times even I've doubted the likelihood of this. However, our experiences with the PVS models of the FGS mode logic has changed my mind on this. I now believe that doing proofs will be a natural outgrowth of building models if a powerful proof system is integrated into the modeling environment. The key is to get engineers building models, and that will require notations and environments that are natural and intuitive for engineers, not the experts in formal methods.

10. Conclusions

I would like to close with one more analogy. One of the truly delightful discoveries of the last few years is that dinosaurs never really died out. They're all around us,

we've just been calling them birds. In a similar manner, I'd like to suggest that formal methods are not facing extinction. They're in widespread use in industry, we've just been calling them names like Boolean algebra, state machines, and grammars.

The real question isn't whether formal methods will be used by industry. They are and will continue to be. The real question is which of the larger species of formal methods will evolve sufficiently to survive in industry.

REFERENCES

[1] D. Best, C. Kress, N. Mykris, J. Russell, and W. Smith, An Advanced-Architecture CMOS/SOS Microprocessor, *IEEE Micro*, Vol. 2, No. 4, pg. 11-26, August 1982.

[2] C. Billings, *Aviation Automation: The Search for a Human-Centered Approach*, Lawrence Erlbaum Associates: Mahwah, New Jersey, 1997.

[3] M. Blackburn and R. Busser, T-VEC: a Tool for Developing Critical Systems, in *Eleventh Annual Conference on Computer Assurance*, pg. 237-249, Gaithersburg, MD, June 1996.

[4] M. Blackburn, *Specification Transformation and Semantic Expansion to Support Automated Testing*, Ph.D. Dissertation, Information Technology, George Mason University, 1997.

[5] R. Boyer and J. Moore, *A Computational Logic*, Academic Press, New York, NY, 1979.

[6] J. Bowen and M. Hinchey, *Applications of Formal Methods*, Prentice-Hall International Ltd., Hemel Hempstead, UK, 1995.

[7] R. Butler and G. Finelli, The Infeasibility of Experimental Quantification of Life-Critical Software Reliability, *IEEE Transactions on Software Engineering*, Vol. 16, No. 5, pg. 66-76, January 1993.

[8] D. Craigen, S. Gerhart, and T. Ralston, *An International Survey of Industrial Applications of Formal Methods: Volume 2 Case Studies*, NISTGCR 93/62, National Institute of Standards and Technology, Gaithersburg, MD 20899, March 1993.

[9] S. Faulk, J. Brackett, P. Ward, and J.Kirby, Jr., The CoRE Method for Real-Time Requirements, *IEEE Software*, September 1992.

[10] S. Faulk, L. Finneran, J. Kirby, S. Shah, and J. Sutton. Experience Applying the CoRE Method to the Lockheed C-130J Software Requirements, in *Ninth Annual Conference on Computer Assurance*, pg. 3-8, Gaithersburg, MD, June 1994.

[11] C. George, et. al., *The RAISE Specification Language*, Prentice Hall International: Hemel Hempstead, England, 1992.

[12] S. Gerhart, M. Bouler, K. Greene, D. Jamsek, T. Ralston, and D. Russinoff, *Formal Methods Transition Study Final Report*, Technical Report STP-FT-322-91, Microelectronics and Computer Technology Corporation, Austin, Texas, August 1991.

[13] D. Greve and M. Wilding, Stack-Based Java a Back-to-Future Step, *Electronic Engineering Times*, pg. 92, January 12, 1998.

[14] M. Heimdahl and N. Leveson, Completeness and Consistency in Hierarchical State-Based Requirements, *IEEE Transactions on Software Engineering*, 22(6):363-377, June 1996.

[15] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A Toolset for Specifying and Analyzing Requirements, in *Tenth International Conference on Computer Assurance*, pg. 109-122, Gaithersburg, MA, June 1995.

[16] C. Heitmeyer, J. Kirby, and B. Labaw, Automated Consistency Checking of Requirements Specification, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(3):231-261, July 1996.

[17] D. Hughes and M. Dornheim, Automated Cockpits: Who's in Charge?: Parts I & II, *Aviation Week & Space Technology*, January 30-February 6,1995.

[18] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese, Requirements Specifications for Process-Control Systems, *IEEE Transactions on Software Engineering*, 20(9):684-707, September 1994.

[19] N. Leveson, et. al., *Analyzing Software Specifications for Mode Confusion Potential*, in Workshop on Human Error and System Development, Glascow, March ,1997.

[20] N. Leveson, *Safeware: System Safety and Computers*, Addison-Wesley Publishing Company: Reading, Massachusetts, 1995.

[21] B. Littlewood and L.Strigini, Validation of Ultra-High Dependability of Software-Based Systems, *Communications of the ACM*, November 1993.

[22] S. Miller and M. Srivas, Formal Verification of the AAMP5 Microprocessor, *Workshop on Industrial-Strength Formal Specification Techniques (WIFT95)*, April 5-8, Boca Raton, Florida, 1995.

[23] S. Miller and K. Hoech, *Specifying the Mode Logic of a Flight Guidance System in CoRE*, Technical Report WP97-2011, Rockwell Collins, Information Center, 400 Collins Road NE, Cedar Rapids, IA 52498, August 1997.

[24] S. Miller, Specifying the Mode Logic of a Flight Guidance System in CoRE and SCR, *Second Workshop on Formal Methods in Software Practice (FMSP98)*, March 4-5, Clearwater Beach, Florida, 1998.

[25] S. Miller, D. Greve, M. Wilding, and M. Srivas, *Formal Verification of the AAMP-FV Microcode: Final Report*, to be published as a NASA contractor report, 1997.

[26] S. Miller and J. Potts, *Detecting Mode Confusion Through Formal Modeling and Analysis: Phase I Final Report*, to be published as a NASA contractor report, 1998.

[27] S. Owre, J. Rushby, N. Shankar, and F. Henke, Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS, *IEEE Transactions on Software Engineering*, Vol. 21, No. 2, pg.107-125, February 1995.

[28] M. Srivas and S. Miller, Applying Formal Verification to a Commercial Microprocessor, in Proceedings of *IFIP Conference on Hardware Description Languages and Their Applications (CHDL'95)*, Makuhari, Chiba, Japan, August 1995.

[29] M. Srivas and S. Miller, *Formal Verification of an Avionics Microprocessor*, NASA Contractor Report 4682, NASA Langley Research Center, Hampton, Virginia, July 1995.

[30] M. Srivas and S. Miller, Formal Verification of the AAMP5 Microprocessor, in *Applications of Formal Methods*, J. Bowen and M. Hinchey, editors, Prentice-Hall International Ltd., Hemel Hempstead, UK, 1995.

[31] M. Wilding, D. Greve and D. Hardin, *Efficient Simulation of Formal Processor Models*, submitted for publication, Rockwell Collins Advanced Technology Center, 400 Collins Road NE, Cedar Rapids, IA 52498, October, 1998