# Lessons From Twenty Years of Industrial Formal Methods

Steven P. Miller

Rockwell Collins,
400 Collins Road NE,
Cedar Rapids, Iowa, 52245, USA
spmiller@rockwellcollins.com

**Abstract.** Over the last two decades formal methods have achieved widespread use by industry in the development of safety and security critical systems. However, these successes often go unacknowledged as evidence of the successful transition of formal methods simply because the name "formal methods" is often still associated with its early prototypes rather than with the successful methods and tools they have evolved into. To better understand the benefits that formal methods can provide, Rockwell Collins has conducted many experiments over the last twenty years in their use. This paper describes several of these experiments, discusses what worked and what didn't, and identifies the key lessons we have learned about introducing formal methods into an industrial setting.

**Keywords:** Formal methods, theorem proving, model checking, model-based development, safety, security, technology transfer, industry, avionics, microprocessor, flight control

## 1 Introduction

Significant gains have been made over the last twenty years in the use of formal methods by industry. To cite just a few, formal graphical modeling languages such as Esterel Technologies SCADE Suite™ [1] and MATLAB Simulink© [2] are widely used in the design of avionics and automotive systems, and both products offer integrated formal verification tools based on model checking and abstract interpretation. Formal annotation of source code and automated proof of verification conditions are routinely used in developments based on the SPARK Pro Ada development system [3]. Airbus has used the Caveat static analyzer to prove properties about about significant portions of the Flight Control and Guidance Unit of the A380 [4]. The Prover® Plug-In [5] model checker has been

This work was supported in part by the NASA Langley Research Center under contract NCC-01001 of the Aviation Safety Program (AvSP) and by the Air Force Research Lab under contract FA8650-05-C-3564 of the Certification Tech-nologies for Advanced Flight Control Systems program (CerTA FCS) [88ABW-2009-2730].

extensively used for the V&V of railway signaling systems. Formal verification tools are routinely used in hardware development. The upcoming revision of the DO-178B software development standard includes a supplement providing guidance on how to use formal methods in the verification of software for civil avionics. While not widely published, companies developing safety or security critical systems routinely make use of proprietary formal specification and verification systems.

These very real gains often go unacknowledged as evidence of the successful transition of formal methods into industry. In some cases, this is because the technologies have been given new names such as model-based development. In other cases, it is simply because the name "formal methods" is often still associated with its early prototypes rather than with the successful methods and tools they have evolved into. Other examples are not well known simply because industry does not have the same need to publicize its use of formal methods as the research community does.

To understand the benefits that formal methods can provide, Rockwell Collins has conducted many trials in their use over the last twenty years. This paper describes several of these experiments, discusses what worked and what didn't, and tries to identify the key lessons we have learned about the industrial use of formal methods.

## 2 Case Studies

This section describes, in roughly chronological order, several experiments conducted in the use of formal methods at Rockwell Collins. These include formal verification of the microcode in the AAMP5 and AAMP-FV microprocessors using the PVS theorem prover, formal verification of the intrinsic partitioning mechanism in the AAMP7G microprocessor using the ACL2 theorem prover, formal verification of the FCS 5000 Flight Control System mode logic and the ADGS-2100 Adaptive Display System Window Manager using the NuSMV model checker, formal verification of the Redundancy Management logic in an adaptive flight control system using NuSMV, and formal verification of the Effector Blendor in an adaptive flight control system using the Prover Plug-In model checker.

### 2.1 AAMP5 Microprocessor

One of our first experiments in the use of formal methods was the verification of the microcode in the AAMP5 microprocessor in 1993 and 1994 [6], [7]. The Advanced Architecture Microprocessor (AAMP) consists of a family of Rockwell Collins proprietary microprocessors used in civil and military avionics systems. The AAMP family is based on a stack architecture that allows it to provide exceptional code density and low power consumption while providing performance comparable to that found in commercial microprocessors of the same era. Once fabricated, the AAMP5 consisted of approximately 500,000 transistors as

compared to 3.1 million in an Intel Pentium. The AAMP5 was designed as an object-code compatible replacement for earlier members of the AAMP family while providing a threefold performance improvement through the use of techniques such as pipelining.

The NASA Langely Research Center funded the Computer Science Research Laboratory of SRI International to work with Rockwell Collins to use the PVS theorem proving system to formally verify the microcode in the AAMP5. To do this, a model of the AAMP5 macroarchitecture and microarchitecture were created by SRI in PVS. The macroarchitecture described the behavior of 24 AAMP instructions at the level of detail seen by an assembly language programmer, specifying how each instruction altered memory, the processor stack, and registers visible to the programmer. These instructions were chosen to include members of each class of the AAMP5's 209 instructions, with 13 instructions in a core set to be verified by SRI and 11 additional instructions to be verified by Rockwell Collins. The microarchitecture described the AAMP5 at the register transfer level, specifying the functionality and interconnection of AAMP5 components such as the arithmetic logic unit, the bus interface unit, and look ahead fetch unit. The microarchitecture specification also included a translation of the microcode for the 24 instructions into PVS. To prove the correctness of the microcode, an abstraction function *abs* was defined mapping the microarchitecture state to the macroarchitecture state and the PVS theorem prover was used to show that the change in the microarchitecture state $m$ caused by the sequence of microcode instructions $i_1, ..., i_n$ implementing each macroinstruction $I$ preserved the abstraction function, i.e., that $abs(i_n(...i_1(m))) = I(abs(m))$.

The final PVS specification of the macroarchitecture for 108 of the AAMP5's 209 instructions resulted in 2550 lines of PVS and took a total of 1,155 hours to to develop and review. The PVS specification of the microarchitecture resulted in 2,679 lines of PVS and took 1,119 hours to develop and review. The proofs of correctness were completed for eleven instructions and took about 800 hours to develop. An additional 316 hours were spent on project managment and education.

Two errors were found in the AAMP5 microcode while developing the macroarchitecture specification simply by exposing ambiguities in the informal specification to the AAMP5 developers. Two test the effectiveness of the proof process, two subtle errors were "seeded" in the microcode delivered to SRI. Both of these errors were detected during completion of the proofs. No other errors were found in the microcode, providing high confidence that the microcode was correct.

The primary lesson we learned from the AAMP5 experiement was that formal verification tools such as the PVS theorem prover had matured to the point where they could be used on industrial problems and that they would systematically find errors. However, there were still many concerns with the cost of formal verification and the level of expertise required to do it. Simply dividing the total project hours by the number of instructions verified put the cost at 308 hours per instruction. Another lesson was that it could be very costly to manually

develop a formal specification that was separate from the design specification the engineers were using for development.

### 2.2 AAMP-FV Microprocessor

At the end of the AAMP5 experiment, it was still unclear what proportion of the AAMP5 verification costs were one time expenses incurred mastering the technology and developing infrastructure (such as supporting PVS libraries) and what costs would be incurred on subsequent projects. Our intuition was that the costs would be much lower, but it was impossible to be sure. To answer this question, the NASA Langley Research Center, Rockwell Collins, and SRI decided to repeat the experiment on the AAMP-FV [8], [9]. Although not actually fabricated, the AAMP-FV was designed for use in ultra-critical applications such as autoland or fly-by-wire. For this reason, it had only 80 instructions as compared to the AAMP5's 209 instructions, used fewer data types and addressing modes, had a flat address space, and was not pipelined. If fabricated, the AAMP-FV would have required about 100,000 transistors compared to about 500,000 for the AAMP5.

The AAMP-FV experiment confirmed our expectations. In contrast to the AAMP5, specification of the AAMP-FV macroarchitecture in PVS took only 130 hours and specification of the microarchitecture took only 138 hours. Proofs were completed for 57 of the AAMP-FV's instructions, at a cost of about 38 hours per instruction, almost an order of magnitude reduction from the AMMP5.

There were several reasons for the reduced costs. For one thing, the AAMP-FV was designed to be verifiable and this did make the proofs easier to complete. However, reuse of PVS libraries, specification patterns, and proof strategies from the AAMP5 project also made significant contributions. Another important factor was that the designers of the AAMP-FV now knew enough to write the PVS specifications themselves. This eliminated the cost of having a PVS expert learn the design of the AAMP-FV. Finally, more instructions in each instruction class were verified. Since verification of each instruction in a class was very similar to that of others in the class, the verification costs per instruction dropped dramatically.

The primary lesson from the AAMP-FV project was that it is very hard to estimate the cost of formal verification from initial experiments. As with anything else, costs drop dramatically as techniques, tools, and skills are developed. The AAMP-FV experiment also pointed out that it is easier for the developers to master the verification tools than it is for experts in formal verification to master a complex product domain.

### 2.3 FCS 5000 Flight Control System

As part of NASA's Aviation Safety Program (AvSP), Rockwell Collins and our partner, the University of Minnesota, decided to investigate the feasibility of using model checking rather than theorem proving. One of our first applications of model checking was to the mode logic of the FCS 5000 Flight Control System

[10],[11]. The FCS 5000 is a family of Flight Control Systems for use in business and regional jet aircraft. The mode logic determines which lateral and vertical flight modes are armed and active at any time. While inherently complex, the mode logic consists almost entirely of Boolean and enumerated types and is written in Simulink. The mode logic analyzed consisted of five interrelated mode transition diagrams with a total of 36 modes, 172 events, and 488 transitions.

Desired properties of the mode logic were formally verified using the NuSMV model checker [12]. Rather than manually creating a formal model to be verfied, we developed a translator framework in conjunction with the University of Minnesota that would automatically convert the Simulink models into the NuSMV specification language. This framework has been extended over the years and now translates Simulink, Stateflow, SCADE, and Lustre[13] models into the NuSMV, Prover, SAL [14], and KIND [15] model checkers, the PVS [16] and ACL2 [17] theorem provers, and into Ada and C source code [18], [19].

This approach was very successful, allowing us to verify properties of the mode logic in seconds. Formal verification of an early version of the mode logic found 26 errors, seventeen of which were found by the model checker. Of these 17 errors, 13 were classified by the FCS 5000 engineers as being possible to miss by traditional verification techniques such as testing and inspections. One was classified as being unlikely to be found by traditional verification techniques.

An important lesson we learned from this experiment was to automatically translate the models the engineers used for code generation rather than to create a verification model by hand. This eliminated the tedious process of trying to keep the engineering model and verification model in sync and enabled us to rerun the verification quickly each time the engineering model was changed. Being able reverify quickly after each modification and to find errors rapidly built confidence with the engineers that the tools worked and could find real errors.

However, the most important lesson we learned was that industrial systems have significant portions of their logic that can be verified with today's model checkers. Even if the entire system can't be formally verified, there are almost always large portions that are amenable to model checking or that can be made amenable with some small changes. This has been been the case in almost every system we have looked at, leading us to conclude that model checking could be used in far more places than is the case today.

### 2.4 ADGS-2100 Window Manager

One of the largest and most successful applications of our tools was to the ADGS-2100 Adaptive Display and Guidance System Window Manager [18],[11],[20]. In modern aircraft, the main way that aircraft status is provided to pilots is through computerized display panels on the flight deck.

The ADGS-2100 is a Rockwell Collins product that provides the heads-down and heads-up displays and display management software for next-generation commercial aircraft. The Window Manager (WM) ensures that data from different applications is routed to the correct display panel. In normal operation, the

WM determines which applications are being displayed in response to the pilot selections. However, in the case of a component failure, the WM also decides which information is most critical and routes this information from one of the redundant sources to the most appropriate display panel. The WM is essential to the safe flight of the aircraft. If the WM contained logic errors, critical flight information could be unavailable to the flight crew.

While very complex, the WM is specified in Simulink using only Booleans and enumerated types, making it ideal for verification using an implicit state model checker such as NuSMV. The WM is composed of five main components that can be analyzed independently. These five components contain a total of 16,117 primitive Simulink blocks that are grouped into 4,295 Simulink subsystems. The reachable state space of the five components ranges from $9.8 \times 10^9$ to $1.5 \times 10^{37}$ states. Ultimately, 563 properties about the WM were developed and checked, and 98 errors were found and corrected in early versions of the WM model.

The ADGS-2100 reinforced the lessons we had learned on the FCS 5000. We were able to find a large portion of the system (the WM) that was well suited for verification with an implicit state model checker. Automatically translating the model the engineers used for code generation into NuSMV was key to providing rapid feedback to the developers and gaining their trust, even when the model was being changed every day.

Another lesson was that it is possible for formal verification to provide real value even if the modeling language itself does not have a published, formal semantics. Languages that support simulation and code generation probably do have an underlying, if unstated, formal semantics. Translating them into a language such as Lustre, implicitly embedding the semantics in the translator, is a viable approach for making formal verification possible. If the translation also replicates the semantics of the generated code, formal verification can be used very effectively to find errors even while the model is being developed.

However, the most important lesson we learned during the ADGS-2100 experiment was that practicing engineers were willing and able to use model checking during their development. During this project, we continuously improved our translation framework from one that required significant manual intervention each time the translation was done to one that was completely automated and took only minutes. Once our tools reached that level of maturity, the developers were more than happy to use the tools, write their own properties, and debug the counterexamples. By the end of the project, the WM developers were checking their properties after every design change [20].

### 2.5 AAMP7G Microprocessor

The AAMP7G is another member of the AAMP family of Rockwell Collins microprocessors. Its design includes an intrinsic hardware partitioning mechanism that provides strict time and space partitioning and allows applications at different criticality levels to execute concurrently on the microprocessor. Originally designed to provide partitioning for Integrated Modular Avionics (IMA) systems

in civil aircraft, it has also received an NSA certificate that allows it to concurrently process Unclassified through Top Secret codeword information, enabling it to be used for security partitioning in many Rockwell Collins products. This certificate is based in large part on a formal proof of correctness of its separation kernel microcode [21].

The verification of the AAMP7G intrinsic partitioning mechanism was completed using the ACL2 theorem prover [17]. The switch to ACL2 from PVS was made partially because of a preference expressed by the customer and partially to for us to learn more about ACL2. The first step in the proof was to formally define what "data separation" means. Informally, data separation means that the data in one partition depends only on the data in partitions that are allowed to interact with that partition, i.e., what occurs in other partitions can have no affect on the partition of interest. This definition, now referred to as the GWV theorem after its authors, was specified as a formal property in the language of the ACL2 theorem prover. It states that the effect of a single step on the system state on an arbitrary segment of memory is a function only of the segments associated with that segment's partition.

To prove that the GWV theorem was satisfied by the AAMP7G, a model of the AAMP7G was constructed at two different levels, the functional level and the abstract level. The functional level corresponds closely to the actual AAMP7G microarchitecture implementation, while the abstract model represents the AAMP7G in a manner more convenient for describing properties. The proof of the GWV theorem was then organized into three pieces:

1. Proofs validating the correctness of the theorem.
2. Proof that the abstract model meets the security specification.
3. Proof that the functional model implements the abstract model.

Just as with the AAMP5 and AAMP-FV projects that required the development of several supporting PVS libraries, several ACL2 libraries were developed to support the AAMP7G verification. The AAMP7G partitioning information is stored in data structures with pointers to other data structures, but this is of course implemented over a linear address space. Most of the libraries developed in ACL2, termed GACC for Generalized Accessor, provided support for reasoning about such structures.

Since the ACL2 models of the AAMP7G were constructed by hand, they were validated through an extensive code-to-spec review with a National Security Agency evaluation team. This review first validated that the separation theorem itself was the correct theorem. Then each of the functions in the ACL2 formal specification were reviewed. The most time consuming part of this was the review that the AAMP7G functional model correctly described the actual AAMP7G implementation. This exhaustive review accounted for each line of trusted microcode and each model of a line of trusted microcode, ensuring that there was nothing left unmodeled, that there was nothing in the model that was not in the actual device, and that each line of the model represented the actual behavior of the microcode.

This review was possible because the functional model was designed to correspond almost exactly to the actual device. Of course, this complicated the proof of the GWV theorem, but this was mitigated by first proving that the astract model satisifed the theorem and then that the functional model implemented the abstract model. The final step in the certification process was for the certification authorities to take the reviewed models and rerun the proof at their location using a trusted copy of the ACL2 theorem prover.

In May of 2005, the AAMP7G was certified as meeting the EAL-7 requirements of the Common Criteria as "... capable of simultaneously processing Unclassified through Top Secret Codeword information". This certificate has allowed the AAMP7G to be used as a foundational component in a number of Rockwell Collins security products.

The time spent manually verifying the AAMP7G model reinforces the need to automatically translate the model the developers use in design into a model that can be used formal verification. However, the most costly part of the AAMP7G verification was the development of supporting libraries, particularly those for reasoning over over complex data structures. These libraries have been reused in subsequent verification efforts, greatly reducing the cost of those efforts.

The AAMP7G verification points out that while model checking may be simpler and more easily used by product developers, many applications still require the full generality and power of theorem proving. These efforts may still be cost effective if sufficient infrastructure has been developed or if the problem is sufficiently important.

### 2.6 CerTA FCS Phase I

The Air Force Research Laboratory (AFRL) Air Vehicles Directorate (RB) has sponsored several studies to investigate the feasiblity of applying model checking to avionics systems. The first of these was conducted under the Certification Technologies for Advanced Flight Critical Systems (CerTA FCS) program in order to compare the effectiveness of model checking and testing [18],[11],[22]. In this study, we applied our tools to the Operational Flight Program (OFP) of an unmanned aerial vehicle developed by Lockheed Martin Aerospace.

The OFP is an adaptive flight control system that modifies its behavior in response to flight conditions. For Phase I of the project, we first looked for a portion of the OFP that was specified using Boolean and enumerated types that could be verified with an with implicit state model checker such as NuSMV. Just as for the FCP 5000 and the ADGS-2100, a large portion of the OFP did fit this criteria. In the case of the OFP, it was the Redundancy Management (RM) logic.

The RM logic was broken down into three components that could be analyzed individually. While relatively small (they contained a total of 169 primitive Simulink blocks organized into 23 subsystems, with reachable state spaces ranging from $2.1 \times 10^4$ to $6.0 \times 10^{13}$ states), the RM logic was replicated in the OFP once for each of the ten control surfaces on the aircraft, making it a significant portion of the OFP logic.

Even for the Redundancy Management logic, our translation framework had to be extended to support all the modeling constructs used by the Lockheed Martin team. In particular, the models made extensive use of StateFlow, which had not been used in the ADGS-2100. Fortunately, we had been working on a translator for StateFlow, so many of the needed enhancements were already done. In addition to handling StateFlow, changes also needed to handle data stores with multiple reads/writes within a step, triggered and enabled subsystems with merge blocks, boundary-crossing and directed acyclic transitions through junctions, variables used as both integers and as bit flags, bit-level operations, and StateFlow truth tables and functions.

To compare the effectiveness of model checking and testing at discovering errors, the project had two independent verification teams, one that used testing and one that used model checking. The two teams communicated only through a single engineer at Lockheed Martin and every effort was made to make sure the verification being done by one team did not influence the other. The formal verification team developed a total of 62 properties from the OFP requirements and checked these properties with the NuSMV model checker, uncovering 12 errors in the RM logic. Of these 12 errors, four were classified by Lockheed Martin as severity 3 (only severity 1 and 2 can affect the safety of flight), two were classified as severity 4, two resulted in requirements changes, one was redundant, and three resulted from requirements that had not yet been implemented in the release of the software.

In similar fashion, the testing team developed a series of tests from the same OFP requirements. Even though the testing team invested almost half again as much time in testing as the formal verification team spent in model checking, testing failed to find any errors. The main reason for this was that the demonstration was not a comprehensive test program. While some of these errors could be found through testing, the cost would be much higher, both to find and fix the errors. In addition, the errors found through model checking tended to be intermittent, near simultaneous, or combinatory sequences of failures that would be very difficult to detect through testing.

Clearly, one of the lessons learned from this experiment was that model checking can be used to find errors more cheaply than testing. More importantly, these errors can be found early in the lifecycle, avoiding the expensive rework that occurs when errors are found during system integration and test. Finally, many of the errors found are those that would be missed by both reviews and testing.

A less obvious lesson is that it is indeed possible to assign a formal semantics to a practical subset of Simulink and StateFlow that can be analyzed with formal verification tools. Since the CerTA FCS project, we and others have used our tools to verify systems modeled by several different companies, demonstrating the portability of our tools. In each case, the key enabling factor was the use of Simulink and StateFlow.

In actuality, the widespread use of languages such as Simulink and SCADE Suite are a huge advantage for the formal methods community. Most researchers

greatly underestimate the difficulty industry faces in switching from one set of modeling tools and processes to another. Once hundreds of engineers are trained in one set of tools, standards and and workflows are developed, and the entire process has been reviewed and accepted by the certification authorities, changes are made only for very compelling reasons. It is far easier for the formal methods community to develop verification tools that work with the formalisms industry is already using than for industry to switch to new formalisms.

### 2.7 CerTA FCS Phase II

Phase II of the CerTA FCS project was to investigate whether model checking could be used to verify large, numerically intensive models. In this study, our translation framework and model checking tools were used to verify properties of the Effector Blender (EB) logic of an OFP for a UAV similar to that verified in Phase I.

The EB is a central component of the OFP that generates the actuator commands for the aircraft's six control surfaces. It is a large, complex model that repeatedly manipulates a $3 \times 6$ matrix of floating-point numbers. It inputs 32 floating-point inputs and a $3 \times 6$ matrix of floating-point numbers and outputs a $1 \times 6$ matrix of floating-point numbers. It contains over 2,000 basic Simulink blocks organized into 166 Simulink subsystems, many of which are StateFlow models.

One of the key challenges in verifying the EB was dealing with the floating-point arithmetic used widely in the EB. Translating the floating- point numbers into rational numbers was rejected since much of the arithmetic in EB is inherently nonlinear, so the decision procedure for linear arithmetic found in many SMT solvers such as Prover and SAL would not help. Even if this hadn't been the case, the use of rational numbers would mask possible floating-point errors such as overflow and underflow.

Instead, the translator framework was extended to convert floating-point numbers to fixed point numbers using a scaling factor provided by the OFP designers. The fixed point numbers were then converted to integers using bit-shifting to preserve their magnitude. This allowed the EB to be verified using Prover's bit-level integer decision procedures. Of course, the results were unsound due to the loss of precision. However, errors found during verification could be checked through simulation to determine if they were also present in the original model. This allowed the verification to be used effectively for debugging, but it did not guarantee correctness.

Even with these adjustments, the EB model was large enough that it had to be decomposed into a hierarchy of components several levels deep. The leaf nodes of this hierarchy were verified using Prover and their composition was verified using manual proofs. This approach also ensured that unsoundness could not be introduced through circular reasoning since Simulink enforces the absence of cyclic dependencies between atomic subsystems.

Ultimately, five errors in the EB design logic were discovered and corrected through model checking of the properties developed for the EB. In addition,

several potential errors that were being masked by defensive design practices were found and corrected.

Verification of the EB helped us to understand the limits of model checking. Since most industrial system use floating-point numbers, industry needs more effective means of dealing with floating-point arithmetic. Even if rational numbers are an acceptable approximation, many systems are nonlinear. In particular, nonlinear functions such as trigonometric functions occur repeatedly in navigation systems and we still lack simple ways of dealing with such functions other than to use linear approximations.

Model checking numerically intensive systems such as the EB also becomes computationally expensive for model checkers, including SMT solvers. A theorem prover could have been used very effectively to automate the compositional reasoning performed in the EB verification, but there wasn't sufficient budget to explore this option. The ideal arrangement would have been a unified model checking and theorem proving environment where model checking could have been used to automatically verify the leaf nodes and a theorem prover used to compose those results and prove properties of the entire EB.

## 3 Lessons

The experiences described in this paper and the examples cited in the introduction show that formal methods can and are being used successfully in industry. In this section, we summarize the main lessons we have learned in using formal methods in an industrial setting.

*All systems have large parts that can be formally verified.* In every system we have looked at there are portions that use only Booleans, enumerated types, and a few small integers. More often than not, these are also the parts of the system causing the developers the most trouble. These parts can be verified very effectively using implicit state model checkers such as NuSMV. Sometimes small changes need to be made to the design to make them amenable to model checking, for example, moving comparisons of numeric variables to thresholds out of the portion of the model to be verified and inputing the results of those comparisons instead of the numeric values themselves. In other cases, the developers could have organized the design differently to make model checking possible if they'd understood how much it could help them.

*One of the main benefits of formal verification is the early detection of errors.* Industry understands all too well that finding errors during system integration (or later) is much more costly than finding those errors early in the lifecycle. Senior management repeatedly emphasizes the importance of finding errors early. But how should developers do this? Increasingly, engineers are building executable, graphical models precisely because this makes it easier to find errors sooner. If these models are also used for code generation, the cost of their development is paid for by eliminating coding and the models are automatically kept in sync

with the implementation. Our experiences have shown that if formal verification, particularly model checking, can be introduced as soon as the first models are developed, it will find errors. The CerTA FCS project demonstated that formal verification can find errors more cheaply than testing. We believe that any additional costs incurred doing formal verification will be recouped many times over by avoiding the rework that would occur if the errors were not found until much later.

*Formal verification will find errors that traditional methods will miss.* Since formal verification considers every possible combination of input and state while testing considers only a tiny fraction of all possible inputs, it is not surprising that formal verification finds errors that testing misses. Formal verification also provides a much more systematic, repeatable, and thorough way to check a model than reviews or inspections. This does not mean that testing or reviews should be discarded. Testing is still needed to demonstrate that the actual product works on its target platform in its final environment and reviews are an effective way to ensure that standards are followed, to identify design alternatives, and to educate new team members. Rather, formal methods should be used to find and correct errors as soon as possible and to find errors that traditional approaches would miss.

*Make it easy to reverify properties when the model changes.* Just as regression testing is a powerful technique for managing change, "regresssion proof" can be a powerful aid during product development. Generally, models will change several times during a project. It is not unusual for developers to change a model several times a day. However, this is precisely when formal verification can provide the greatest benefit. This requires that properties be developed that are not sensitive to the internal structure of the model. It may also require that the model be designed to facilitate verification with the available tools.

*Take advantage of formalisms already in use in industry.* The widespread use of model-based development and commercial modeling tools such as SCADE Suite, Simulink, and StateFlow is a huge advantage for the formal methods community. Our experiences show that it is possible to assign a formal semantics to these notations and that formal verification can provide real value back to the developers. In addition, many companies have developed and are using proprietary formalisms for which formal verification tools can be built. A very effective technique to take advantage of these formalisms is to develop translators from that formalism to one or more formal verification tools. It is much easier for the formal methods community to develop verification tools that work with the formalisms industry is already using than for industry to switch to different formalisms.

*Costs drop rapidly with experience.* As demonstrated directly by the AAMP-FV verification, there are significant costs in mastering the formal verification tools and in developing the infrastructure, such as domain specific libraries, that will not be incurred on subsequent verification efforts. It is not unrealistic for there

to be an order of magnitude reduction in cost once the verification process is well understood and the supporting infrastructure is fully developed.

*Use the right verification technology for the problem.* Systems that can be specified using Boolean, enumerated, and small integer types can often be verified using imnplicit state model checkers, as demonstrated on the FCS 5000 mode logic, the ADGS-2100 Window Manager, and the OFP RM logic. We have found implicit state model checkers to be something our engineers can pick up and use effectively with only a couple days of training and ongoing mentoring. More complex models using linear floating-point arithmetic can often be verified using SMT-solvers, but they are often harder to use than implicit state model checkers. In the case of the Effector Blendor, the presence of non-linear arithmetic forced us to use approximate techniques that product developers probably would not have the time to learn or use. Finally, some domains, such as the AAMP5, AAMP-FV, and AAMP7G microprocessor verification, still seem to require the full power and generality of theorem proving.

*Pick important problems.* While model checking is starting to approach the point where it could be effectively used on almost any sort of development, it is best suited for applications demanding high levels of assurance. Each of the experiments described in this paper were on components either used in or representative of the safety and security critical systems we develop. While we consider each of these experiments to be a successful demonstration of the effectiveness of formal verification. However, even though the verification of the AAMP7G was probably the most expensive, it was also probably one of the most important since it was key to the development of several subsequent products.

## 4 Future Directions

There also remain many areas for further research. The difficulty of formal verification of numerically intensive systems, even for linear arithmetic using rational numbers, makes it prohibitively expensive for most industrial developers. Cost effective techniques for dealing with floating-point numbers remains an outstanding problem; treating the floating-point variables as rational numbers introduces the potential to miss rounding errors such as underflow and overflow. Better techniques are also needed to deal with transcendental functions, particularly trigonometric functions, that are used routinely in navigation systems.

Compositional verification is also an area where better tools and techniques are needed. The verification of the Effector Blendor in the CerTA FCS project suggests that a unified environment for model checking and theorem proving might be a viable solution. At this point, the state of the art appears to be theorem proving environments that make use of decision procedures similar to those used in model checkers. A better solution would be an integrated enviroment that allowed parts of a system to be verified with different tools and those results composed to achieve verification of the entire system.

As demonstrated by the AAMP7G verification, theorem proving remains an important choice in the formal methods toolbox. Development and sharing of domain specific libraries of properties can greatly reduce the cost and time associated with theorem proving. Additional work could also be done to more fully automate the discovery of complex proofs.

Trusting the verification tools themselves will become more of an issue in the years to come. The Formal Methods Supplement to be released with the new standard for the development of civil avionics software, DO-178C, will require that formal verification tools be "qualified" if they are to be used to meet DO-178C objectives. Techniques to develop trusted verification tools, or to develop separate checkers that are highly trusted, will become increasing important as industry makes greater use of formal verification.

Finally, most industrial systems today consist of many components that each execute synchronously, but that communicate asynchronously over a bounded-delay network. At this time, formal verfication seems to be limited to the verification of the synchronous components or to some aspect of the overall system. Much better tools are needed that allow individual components to be formally verified and those results composed (correctly modeling the asynchronous execution and communication of the components) to verify safety, security, and performance properties of the entire system.

**Acknowledgments.** Many individuals have contributed to the work described in this paper. The author thanks Ricky Butler, Celeste Bellcastro, Kelly Hayhurst and Paul Miner of the NASA Langley Research Center, Mats Heimdahl, Yunja Choi, Anjali Joshi, Ajitha Rajan, Sanjai Rayadurgam and Jeff Thompson of the University of Minnesota, Brad Martin and Mark Vanfleet of the National Security Agency, Eric Danielson, Scott Ervin, Cleveland Gilbert, John Innis, Ray Kamin and David Lempia of Rockwell Collins, Bruce Krogh of CMU, Vince Crum, Wendy Chou, Ray Bortner, and David Homan of the Air Force Research Lab, and Greg Tallant and Walter Storm of Lockheed Martin for their contributions and support. Several of the studies described in this paper were conducted by Mike Whalen of the University of Minnesota and Darren Cofer, Michael Dierkes, Dave Greve, Dave Hardin, Lucas Wagner and Alan Tribble of Rockwell Collins.

## References

1. Esterel Technologies, SCADE Suite Product Description, `http://www.estereltechnolgies.com`
2. The Mathworks, Simulink Product Description, `http://www.mathworks.com`
3. AdaCore, SPARK Pro Product Description, `http://www.adacore.com/home/products/sparkpro/`
4. Souyris, J. and Favre-Felix, D.: Proof of Properties in Avionics, In: Jacquart, R. (ed.) Building the Information Society, 18th IFIP World Computer Congress. pp. 528–535, Kluwer Academic Publishers, Norwell, MA (2004).

5. Prover, Prover Home Page, http://www.prover.com
6. Srivas, M., Miller, S.: Formal Verification of the AAMP5 Microprocessor, In: Hinchey, M., Bowen, J. (eds.) Applications of Formal Methods. Prentis-Hall International Ltd, Hemel Hempstead, UK (1995)
7. Srivas, M., Miller, S.: Formal Verification of an Avionics Microprocessor. NASA Contractor Report 4682, NASA Langley Research Center, Hampton, VA (1995).
8. Miller, S., Greve, D., Srivas, M.: Formal Verification of the AAMP5 and the AAMP-FV Microcode. In Third AMAST Workshop on Real-Time Systems, Salt Lake City, Utah, March 6-8 (1996)
9. Miller, S., Greve, D., Wilding, M.: Formal Verification of the AAMP-FV Microcode. NASA Contractor Report CR-1999-208992, NASA Langley Reseearch Center, Hampton, VA (1999).
10. Miller, S., Anderson, E., Wagner, L., Whalen, M., Heimdahl, M.: Formal Verification of Flight Critical Software. In AIAA Guidance, Navigation and Control Conference and Exhibit, AIAA-2005-6431, American Institute of Aeronautics and Astronautics (2005)
11. Miller, S.: Will This Be Formal? In: Mohamed O.A., Munoz, C., Tahar, S. (eds.) Theorem Proving in Higher Order Logics, $21^{st}$ International Conference, TPHOLs 2008, Montreal, Canada, Aug 18-21, LNCS, vol. 5170, pp. 6–11. Springer (2008)
12. IRST, The NuSMV Model Checker, http://nusmv.irst.itc.it
13. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The Synchronous Dataflow Programming Language Lustre. Proceedings of the IEEE, vol. 79, no. 9, pp. 1305–1320 (1991)
14. SRI International, Symbolic Analysis Laboratory, http://sal.csl.sri.com
15. Kind - A Safety Property Verifier for Lustre Programs, http://combination.cs.uiowa.edu/Kind/
16. SRI International, PVS Specification and Verification System, http://pvs.csl.sri.com
17. The University of Texas at Austin, ACL2 Version 4.1, http://www.cs.utexas.edu/users/moore/acl2/
18. Miller, S., Whalen, M., Cofer, D.: Software Model Checking Takes Off. Communications of the ACM. vol. 53, no. 2, pp. 58–64 (2010)
19. Miller, S., Tribble, A., Whalen, M., Heimdahl, M.,: Proving the Shalls. International Journal on Software Tools for Technology Transfer (STTT), February (2006)
20. Whalen, M., Innis, J., Miller, S., Wagner, L.,: ADGS-2100 Adaptive Display & Guidance System Window Manager Analysis, NASA Contractor Report CR-2006-213952, http://shemesh.larc.nasa.gov/fm/fm-collins-pubs.html, February (2006)
21. Wilding, M., Greve, D., Richards, R., Hardin, D.: Formal Verification of Partition Management for the AAMP7G Microprocessor, In: Hardin, D., (ed.) Design and Verification of Microprocessor Systems for High-Assurance Applications. Springer, New York (2010)
22. Whalen, M., Cofer, D., Miller, S., Krogh, B., Storm, W.: Integration of Formal Analysis into a Model-Based Software Development Process, In Proceedings of the 12th International Workshop on Formal Methods for Industrial Critical Systems (FMICS2007), Berlin, Germany, (2007)