SOFTWARE MODEL CHECKING FOR AVIONICS SYSTEMS

Darren Cofer, Michael Whalen, Steven Miller
Rockwell Collins, Cedar Rapids, IA

Abstract

The adoption of model-based development tools is changing the cost-benefit equation for the industrial use of formal methods. The integration of formal methods such as *model checking* into software development environments makes it possible to fight increasing cost and complexity with automation and rigor. This paper describes how formal analysis tools can be inserted into a model-based development process to decrease costs and increase quality of safety-critical avionics software.

Introduction

By any measure, the size and complexity of the safety-critical software deployed in commercial and military aircraft are rising exponentially. Current verification methods will not be able to cope effectively with the software being developed for next-generation aircraft. New verification processes are being developed that augment testing with analysis techniques such as *formal methods*. These processes will help ensure that the advanced functionality needed in modern aircraft can be delivered at a reasonable cost and with the required level of safety.

In the past, formal methods have not been widely used in industry due to a number of barriers:

1. The cost of building separate analysis models
2. The difficulty of keeping these models consistent with the software design
3. The use of unfamiliar notations for modeling and analysis
4. The inadequacy of tools for industrial-sized problems

The wide-spread use of *model-based development* (MBD) tools is eliminating the first three barriers. MBD refers to the use of domain-specific (usually graphical) modeling languages that can be executed in simulation before the actual system is built. The use of such modeling languages allows engineers to create a model of the system, execute it on their desktop, and automatically generate code and test cases. Furthermore, tools are now being developed to translate these design models into analysis models that can be verified by formal methods tools with the results translated back into the original modeling notation. This process leverages the original modeling effort and allows engineers to work in familiar notations for their domain.

The fourth barrier is being removed through dramatic improvements in analysis algorithms, and the steady increase in computing power readily available to engineers due to Moore's Law. The combined forces of faster algorithms and cheap hardware mean that systems that were out of reach a decade ago can now be analyzed in a matter of seconds.

Formal analysis tools permit software design models to be evaluated much more completely than is possible through simulation or test. This permits design defects to be identified and eliminated early in the development process, when they have much lower impact on cost and schedule. By reducing the number of defects that are not discovered until later phases, the cost of software development can be reduced significantly.

Model-Based Development

Model-Based Development refers to the use of domain-specific modeling notations such as Simulink or SCADE that can be analyzed for desired behavior before a digital system is built. The use of such modeling languages allows a system engineer to create a model of the desired system early in the lifecycle that can be executed on the desktop, analyzed for desired behaviors, and then used to automatically generate code and test cases. The emphasis in model-based development is to focus the engineering effort on the early lifecycle activities of modeling, simulation, and

978-1-4244-2208-1/08/$25.00 ©2008 IEEE.

analysis, and to automate the late life-cycle activities of coding and testing. This reduces development costs by finding defects early in the lifecycle, avoiding rework that is necessary when errors are discovered during integration testing, and by automating coding and the creation of test cases. In this way, model-based development significantly reduces costs while also improving quality.

Formal methods may be applied in a MBD process to prevent and eliminate requirements, design, and code errors, and should be viewed as complementary to testing. While testing shows that functional requirements are satisfied for specific input sequences and detects some errors, formal methods can be used to increase confidence that a system will *always* comply with particular requirements when specific conditions hold. Informally we can say that testing shows that the software *does* work for *certain test cases* and formal, analytical methods show that it *should* work for *all* cases. It follows that some verification objectives may be better met by formal, analytical means and others might be better met by testing.

Although formal methods have significant technical advantages over testing for software verification, they are not yet widely used in the aerospace industry. The additional cost and effort of creating and reasoning about formal models in a traditional development process has been a significant barrier. Manually creating models solely for the purpose of formal analysis is labor intensive, requires significant knowledge of formal methods notations, and requires that models and code be kept tightly synchronized to justify the results of the analysis.

The value proposition for formal methods changes dramatically with the introduction of MBD and the use of completely automated analysis tools. Many of the notations in MBD have straightforward formal semantics. This means that it is possible to use models written in these languages as the basis for formal analysis, removing the incremental cost for constructing verification models.

Model Checking

Model checking is a category of formal methods that is particularly well suited to integration in MBD environments. Model checkers are highly automated, requiring little to no user interaction, and provide the verification equivalent of exhaustive testing of the model. A model checker will consider every possible combination of system input and state, and determine whether or not a specified set of properties is true [1]. If a property is not true, the model checker will produce a counterexample showing how the property can be falsified (Figure 1).

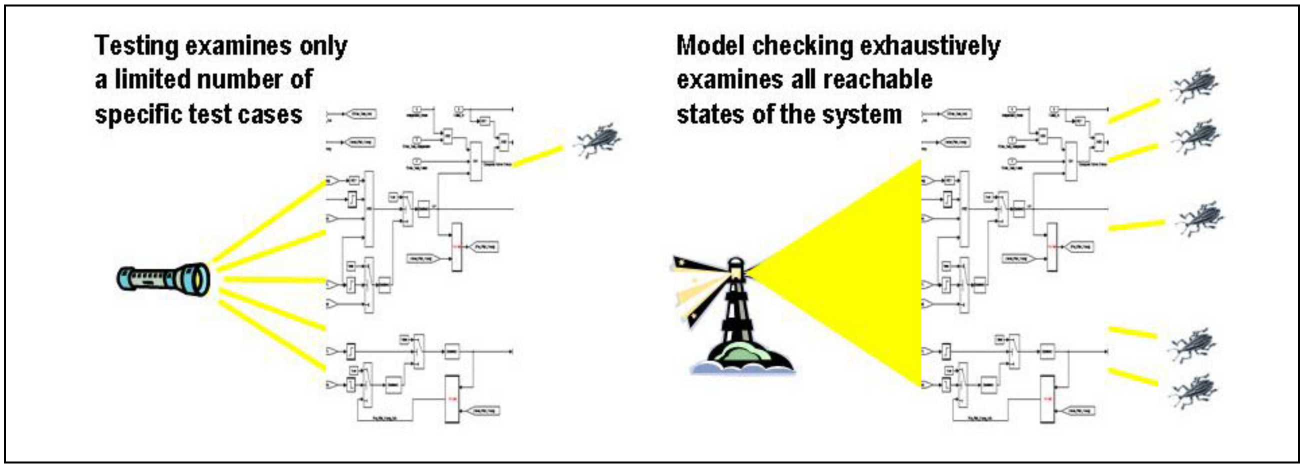
Testing examines only a limited number of specific test cases
Model checking exhaustively examines all reachable states of the system

Figure 1. Testing vs. Model Checking

Model checkers are now sufficiently powerful to allow “push-button” analysis of interesting properties over large models, removing the manual analysis cost. If a property is violated, the model checker generates a counterexample, which is simply a test case that shows a scenario that violates the property. The counterexamples generated by model checkers are often better for localizing and correcting failures than discovering failures from testing and simulation because they tend to be very short (under 10 input steps) and tailored towards the specific requirement in question.

There are many types of model checkers, each with its own strengths and weaknesses. Explicit state model checkers such as SPIN [2] directly execute the formal model and store a representation of each state visited. Implicit state (symbolic) model checkers use Binary Decision Diagrams (BDDs) to store a very compact representation of all the states visited. This allows them to handle a much larger state space than explicit state model checkers. We have used the BDD-based model checker NuSMV [3] to analyze models with over 10^{120} reachable states. Satisfiability modulo theories (SMT) based model checkers such as SAL [4] and Prover [5] use a form of induction to automatically prove that a property holds over the model. While this allows them to handle a larger class of models, including infinite state models that contain real numbers, the properties to be checked must be written to support inductive proofs. In some cases, this makes SMT-based model checkers more difficult to use than an explicit or implicit state model checker.

Development of more efficient and powerful algorithms for model checking is currently an active area of research. Over the past decade, dramatic improvements have be achieved, reducing the time required to analyze models by several orders of magnitude, and similarly increasing the size and complexity of the models that can be analyzed. Furthermore, Moore's Law continues to increase the computing power available on the desktop of software engineers. This means that powerful model checking tools can be made available to engineers, and that they are suitable for analyzing industrial-scale problems. Later in the paper, we describe several successful case studies where we have used model checking to perform verification of real avionics software products.

Automated Translation Framework

In collaboration with the University of Minnesota under NASA's Aviation Safety Program, Rockwell Collins has developed a translation framework that bridges the gap between some of the most popular industrial MBD languages and several model checkers (Figure 2). These automated tools allow us to quickly and easily generate models for verification directly from the design models produced by the MBD process. The counterexamples generated by model checking tools can be translated back to the MBD environment for simulation. This tool infrastructure provides the means for integration of formal methods directly and efficiently into the MBD process. Software engineers can continue to develop design models using the tools that they are already familiar with.

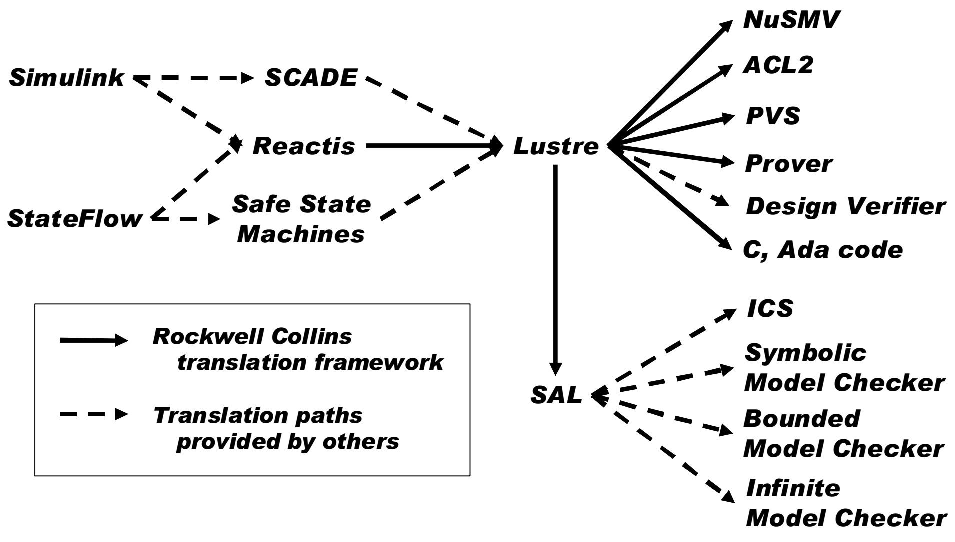

Figure 2. Translation Framework

The translators use the Lustre formal specification language, developed by the synchronous language research group at Verimag, as an intermediate representation for the models. Models developed in Simulink, StateFlow, or SCADE are transformed into Lustre. Once in Lustre, the specification is loaded into an abstract syntax tree (AST) and a number of transformation passes are applied to it. Each transformation pass produces a new Lustre AST that is syntactically closer to the target specification language and preserves the semantics of the original Lustre specification. This allows all Lustre type checking and analysis tools to be used after each transformation pass. When the AST is sufficiently close to the target language, a pretty printer is used to output the target specification. This customized translation approach allows us to select the model checker whose capabilities are best suited to the model being analyzed, and to generate an analysis model that has been optimized to maximize the performance of the selected model checker.

Since Lustre is the underlying language for SCADE models, the initial translation step is immediate. For Simulink and StateFlow models, we use the Reactis test case generator tool [6] to support the initial translation step. We also use the Reactis simulator as the primary means for playback of the counterexample test cases. To ensure that each Simulink or Stateflow construct has a well-defined semantics, the translator restricts the models that it will accept to those that can be translated unambiguously into Lustre.

Our translation framework is currently able to target eight different formal analysis tools. Most of our work has focused on the NuSMV model checker and the Prover model checker. We can also use the same translation framework to generate C or Ada source code.

Analysis Process

In a test-based verification process, test cases must be developed for each requirement. Each test case defines a combination of input values (a test vector) or a sequence of inputs (a test sequence) that specifies the operating condition(s) under which the requirement must hold. The test case must also define the output to be produced by the system under test in response to the input test sequence.

An analysis-based verification process may be thought of in the same way (see Figure 3). We normally consider a group of requirements (often expressed in English as "shall statements"), with related functionality for a particular subsystem. The environmental assumptions or constraints specify the operating conditions under which the requirements must hold. The properties define subsystem behaviors (values of outputs or state variables) that must hold for all system states reachable under the specified environmental assumptions. Properties are simply a precise specification of the requirements of the system that will be analyzed.

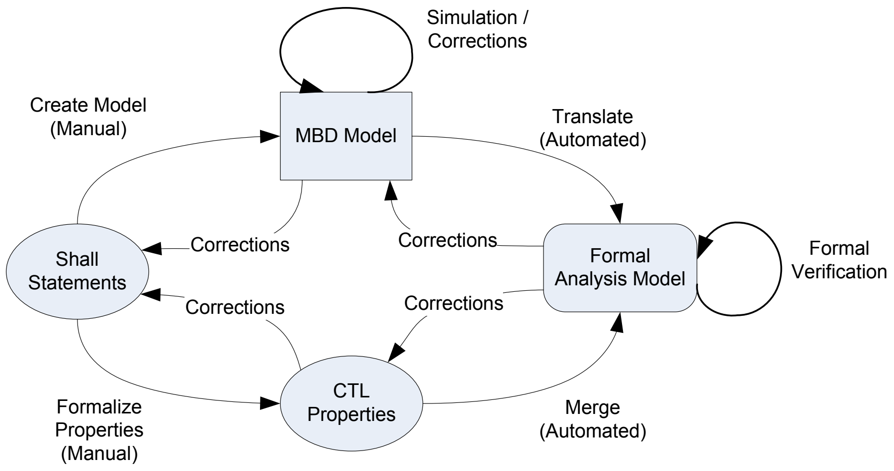

Figure 3. Analysis Process

The essential difference is one of precision: model checking requires the specification of exactly what is meant by specific requirements and determines all possible violations of those requirements at the subsystem level. This precision can be challenging, because an engineer is no longer allowed to rely on an intuitive understanding to create test vectors. Also, in some cases, specialized notation may be used for writing properties (such as CTL and LTL [1]), though there are a variety of notations (including the MBD languages themselves) that can be used to mitigate this difficulty.

After the requirements have been specified as properties to the verified and a model has been built to implement the requirements, the formal analysis can be performed. The model is translated into the native language of the chosen model checking tool and merged with the properties to the verified. The model checker then performs the analysis and either reports that the properties are true, or provides a counterexample that refutes the property.

One of the benefits of using a model checker in the verification process is the generation of counterexamples that illustrate exactly how a property has been violated. This provides useful guidance to the developer to determine what went wrong and what needs to be fixed. For large systems it can be difficult and time consuming to determine the root cause of the violation by examining only the model checker output. Instead, the simulation capabilities of the MBD tools can be utilized to allow playback of a counterexample.

Both Simulink and SCADE have sophisticated simulation capabilities that allow single-step playback of tests and easy "drill down/drill up" through the structure of the model. These capabilities can be used to quickly localize the cause of failure for a counterexample. Third-party tools such as Reactis for Simulink also allow a "step back" function so that it is possible to rewind and step through a sequence of steps within a counterexample, adding to the explanatory power of the tool.

When a counterexample is discovered, it is classified by its underlying cause and appropriate corrective action taken. The cause may be one or more of the following:

- Modeling error
- Property formalization error
- Incorrect/missing invariants for the subsystem
- Requirements error

The fact that a model checker generates a counterexample from the set of all possible violations of a property can lead to "nonsensical" counterexamples in which the model inputs change in ways that would be impossible in the real environment. In order to remove these counterexamples that will not occur in the real system, it is sometimes necessary to describe environmental constraints that describe how the inputs to the model are allowed to evolve. On the bright side, these constraints serve as a precise description of the environmental assumptions required by the component to meet its requirements.

Success stories

FCS 5000 Mode Logic

Our first attempt to apply this approach to an actual product was the mode logic of the FCS 5000 Flight Control System [7]. The FCS 5000 is a family of Flight Control Systems developed by Rockwell Collins 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 implemented in Simulink. This made the FCS 5000 mode logic ideally suited for analysis using our translator framework and a symbolic model checker such as NuSMV. The mode logic we analyzed consisted of five mode transitions diagrams with a total of 36 modes, 172 events, and 488 transitions. Changes in the state of each mode diagram affect at least one, and often more than one, of the other mode diagrams. While each individual diagram is straightforward to understand, grasping all the interactions between them is difficult.

Analysis of an early specification of the mode logic found 26 errors. Seventeen of these were found by the model checker. Of these 17 errors, 13 were classified by the FCS 5000 engineers as being possible to be missed by traditional verification techniques such as testing and inspections. One was classified as being unlikely to have been found by traditional techniques.

One of the main advantages of this analysis was that it could be done early in the development process while the requirements were still under development. Finding and correcting errors at this stage is far more cost effective than waiting until executable code is ready for unit and integration testing.

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 [8]. In modern aircraft, the primary way that aircraft status is provided to pilots is through computer driven flat panel displays similar to those shown in Figure 4. 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.

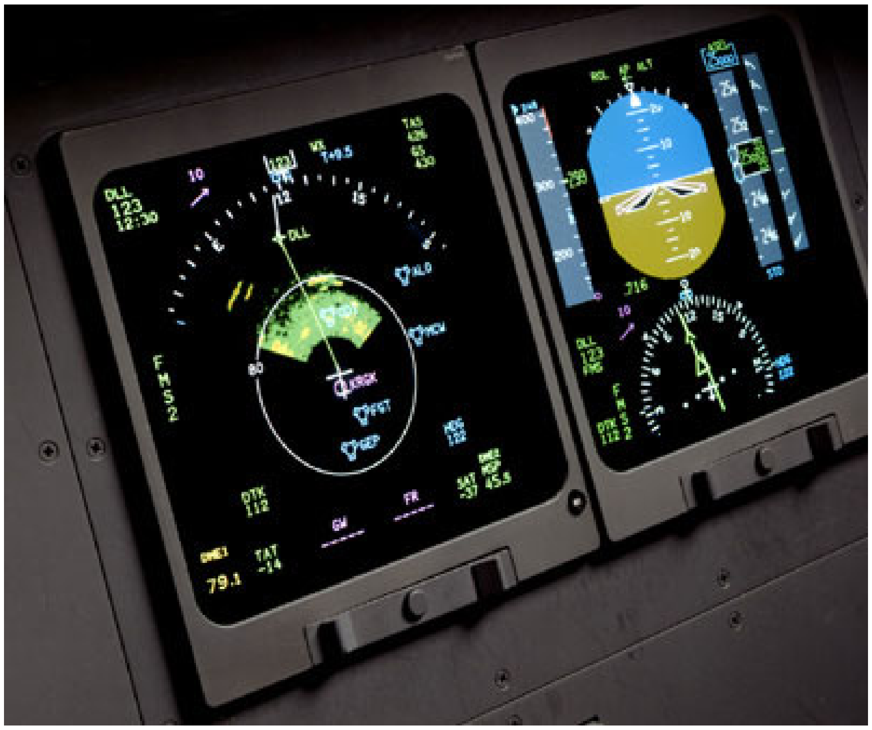

Figure 4. Pilot Display Panels

The pilots can switch each panel between several different displays of information such as primary flight displays, navigational maps, aircraft system status, and flight checklists. However, some information is considered critically important and must always be displayed. For this reason, the ADGS-2100 provides redundant implementations of all its critical functions.

The Window Manager ensures that data from the different displays 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 contains logic errors, critical flight information could be unavailable to the flight crew.

Like the FCS 5000 mode logic, the WM is specified in Simulink using only Booleans and enumerated types, but it is surprisingly complex. 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 instances of Simulink subsystems. The reachable state space of the five components ranges from 9.8×10^9 to 1.5×10^{37} states.

At the start of the project, the immaturity of our tools and the natural skepticism of the WM developers meant that all the early analysis was done by us. However, as the project progressed, we improved the tool chain so that the translation was completely automated and only took a few minutes. We also implemented optimizations to the translator that reduced the time required for NuSMV to check each property to roughly 10 seconds.

At the same time, the developers began to see that model checking could find errors much faster, more easily, and more thoroughly than testing or reviews. This motivated them to start writing and checking CTL properties on their own. At some point, they completely took over the model checking and began relying on us only for consultation and tool improvements.

Ultimately, 593 properties about the WM were developed and checked, and 98 errors were found and corrected in early versions of the model. As with the FCS 5000 mode logic, this verification was done early in the design process while the design was still changing. By the end of the project, the WM developers were checking the properties after every design change.

CerTA FCS Adaptive Flight Control

In the Certification Technologies for Flight Critical Systems (CerTA FCS) project funded by AFRL, we have analyzed several software components of an adaptive flight control system for an unmanned aircraft developed by Lockheed Martin. One system we analyzed was the redundancy manager which implements a triplex voting scheme for fault-tolerant sensor inputs. We performed a head-to-head comparison of verification technologies with two separate teams, one using testing and one using model checking. We recorded in detail how time was spent by each team. In evaluating the same set of system requirements, the model checking team discovered 12 errors while the testing team discovered none. Furthermore, the model checking evaluation required 1/3 less time. Additional details regarding this work can be found in [9].

Next Steps

Current research is focused on further expanding the range of models where model checking can be effectively applied. Our framework can deal with integers and fixed-point data types, but new analysis methods are needed to handle larger data types, floating point numbers, and non-linear functions. Work on the class of analysis algorithms known as SMT model checkers appears promising.

Several commercial MBD environments have begun to incorporate model checkers. This should make formal analysis more widely accessible to software engineers. However, it is not yet clear whether the same power and flexibility can be provided in off-the-shelf development environments as is available in custom approaches.

The impact that the use of formal verification technologies will have on software aspects of aircraft certification is a major issue at this time. The international committee (SC-205/WG-71) responsible for the next generation of certification guidance (DO-178C) is addressing this question.

Conclusions

The growing popularity of model-based development tools and the increasing power of model checkers are making formal verification practical for industrial use. Our translation framework solves one of the main obstacles to the use of model checking by bridging the gap between the commercial modeling tools used by software engineers and existing formal verification tools. This permits model checking to be integrated into the development process so that models can be analyzed automatically with little additional effort.

Our experiences with three case studies show that model checking can be applied effectively for a large class of avionics software models. We were able to find and correct design errors early in the development cycle that could easily be missed by reviews and testing. Eliminating errors early in the development process is much more cost effective and will help to reduce the cost of software development.

Acknowledgements

This work was supported in part by AFRL and Lockheed Martin Aeronautics Company under prime contract FA8650-05-C-3564.

References

[1] Edmund Clarke, Orna Grumberg, and Doron Peled, Model Checking, The MIT Press, Cambridge, Massachusetts, 2001.

[2] Gerard Holzmann, "The SPIN Model Checker: Primer and Reference Manual," Addison-Wesley Professional, 2003.

[3] IRST, "The NuSMV Model Checker," http://nusmv.irst.itc.it.

[4] SRI International, "Symbolic Analysis Laboratory," http://sal.csl.sri.com.

[5] PROVER Technology, "PROVER Product Description," http://www.prover.com.

[6] Reactive Systems, Inc, Reactis Home Page, http://www.reactive-systems.com.

[7] Steven Miller, Elise Anderson, Lucas Wagner, Michael Whalen, and Mats P.E. Heimdahl, "Formal Verification of Flight Critical Software," Proceedings of the AIAA Guidance, Navigation and Control Conference and Exhibit, San Francisco, August 15-18, 2005.

[8] Michael Whalen, John. Innis, Steven Miller, and Lucas Wagner, "ADGS-2100 Adaptive Display & Guidance System Window Manager Analysis," NASA Contractor Report CR-2006-213952, February 2006. Available at http://shemesh.larc.nasa.gov/fm/fm-collins-pubs.html.

[9] Michael Whalen, Darren Cofer, Steven Miller, Bruce Krogh, Walter Storm, "Integration of formal analysis into a model-based software development process," Proceedings of Formal Methods for Industrial Critical Systems, Berlin, Germany, July 2007.

27th Digital Avionics Systems Conference
October 26-30, 2008