SOFTWARE MODEL CHECKING TAKES OFF

Steven P. Miller, Michael W. Whalen, Darren D. Cofer
Advanced Technology Center, Rockwell Collins, Cedar Rapids, IA 52498

Abstract

The increasing popularity of model-based development and the growing power of model checkers are making it practical to use formal verification for important classes of software designs. A barrier to doing this in an industrial setting has been the need to translate the commercial modeling languages developers use into the input languages of the verification tools. This paper describes a translator framework that enables the use of model checking and theorem proving on complex avionics systems and describes its use in three industrial case studies.

Introduction

Although formal methods have been used in the development of safety and security critical systems for years, they have not achieved widespread industrial use in software or systems engineering. However, two important trends are making the industrial use of formal methods practical.

The first is the growing acceptance of model-based development for the design of embedded systems. Tools such as MATLAB Simulink® [1] and Esterel Technologies SCADE Suite™ [2] are achieving widespread use in the design of avionics and automotive systems. The graphical models produced by these tools provide a formal, or nearly formal, specification that is often amenable to formal analysis.

The second is the growing power of formal verification tools, particularly model checkers. For many classes of models they provide a "push-button" means of determining if a model meets its requirements. Since these tools examine all possible combinations of inputs and state, they are much more likely to find design errors than testing.

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 Technologies for Advanced Flight Control Systems program (CerTA FCS) [88ABW-2009-2730].

In this paper, we describe a translator framework developed by Rockwell Collins and the University of Minnesota that allows us to automatically translate from some of the most popular commercial modeling languages to a variety of model checkers and theorem provers. We describe three case studies in which these tools were used on industrial systems that demonstrate that formal verification can be used effectively on real systems when properly supported by automated tools.

Background

Model-Based Development

Model-based development (MBD) refers to the use of domain-specific, graphical modeling languages that can be executed and analyzed before the actual system is built. The use of such modeling languages allow the developers to create a model of the system, execute it on their desktop, analyze it with automated tools, and use it to automatically generate code and test cases.

In this paper, we use MBD to refer specifically to software developed using synchronous dataflow languages such as those found in MATLAB Simulink and Esterel Technologies SCADE Suite. Synchronous modeling languages latch their inputs at the start of a computation step, compute the next system state and its outputs as a single atomic step, and communicate between components using dataflow signals. This differs from the more general class of modeling languages that include support for asynchronous execution of components and communication using message passing. MBD has become very popular in the avionics and automotive industries and we have found synchronous dataflow models to be especially well suited for automated verification using model checking.

Model checking

Model checkers are formal verification tools that evaluate a model to determine if it satisfies a given set of properties [3]. A model checker will

consider every possible combination of inputs and state, making the verification equivalent to exhaustive testing of the model. If a property is not true, the model checker produces a counterexample showing how the property can be falsified.

There are many types of model checkers, each with their own strengths and weaknesses. Explicit state model checkers such as SPIN [4] construct a searchable representation of the design model and store a representation of each state visited. Implicit state (symbolic) model checkers use logical representations of sets of states (such as Binary Decision Diagrams) to describe regions of the model state space that satisfy the properties being evaluated. Such compact representations generally allow symbolic model checkers to handle a much larger state space than explicit state model checkers. We have used the BDD-based model checker NuSMV [5] to analyze models with over 10^{120} reachable states.

More recent model-checkers, such as SAL [6] and Prover® Plug-In [7] use *satisfiability modulo theories* (SMT) solvers for reasoning about infinite state models containing real numbers and unbounded arrays. These checkers use a form of induction over the state transition relation to automatically prove that a property holds over all executable paths in a model. While these tools can handle a larger class of models, the properties to be checked must be written to support inductive proof.

The Translator Framework

As part of NASA's Aviation Safety Program (AvSP), Rockwell Collins and the University of Minnesota developed a product family of translators that bridge the gaps between some of the most popular commercial modeling languages and several model checkers and theorem provers [8]. An overview of this framework is shown in figure 1.

These translators work primarily with the Lustre formal specification language [9], but this is hidden from the users. The starting point for translation is a design model in MATLAB Simulink/Stateflow or Esterel Technologies SCADE Suite/Safe State Machines. SCADE Suite produces Lustre models directly. Simulink or Stateflow models can be imported using SCADE Suite or the Reactis® [10] tool and a translator developed by Rockwell Collins. 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.

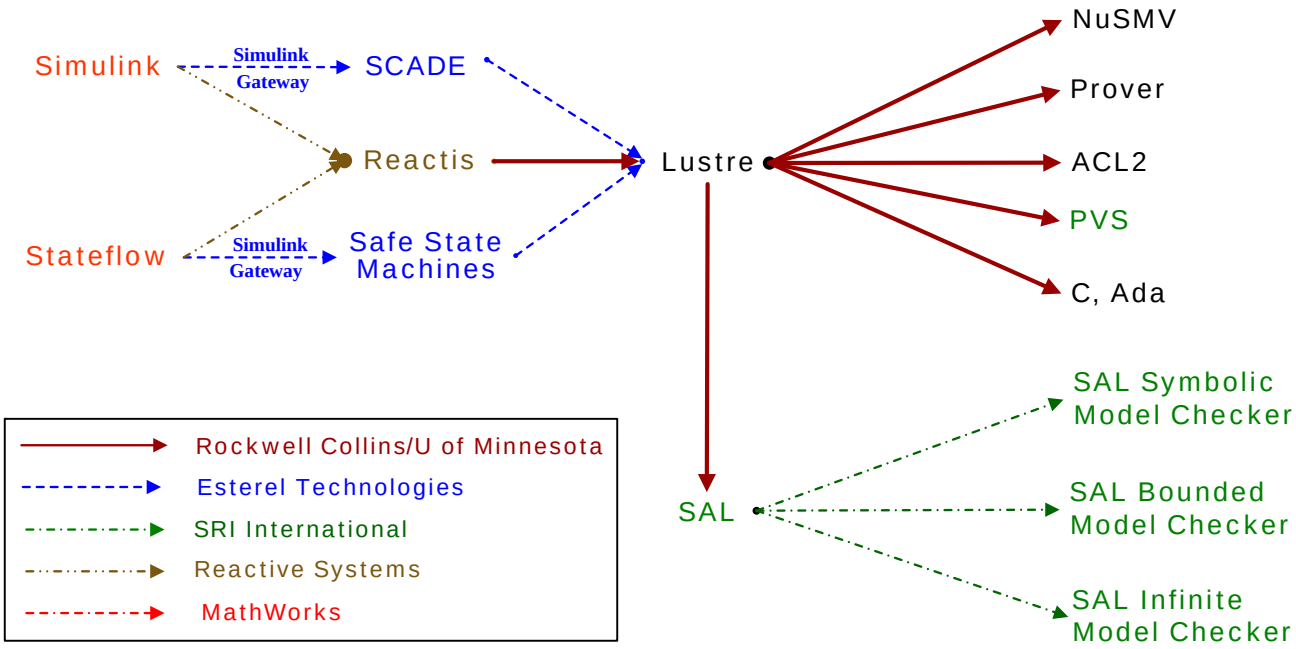

FIGURE 1 – THE TRANSLATOR FRAMEWORK

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 as debugging aids during the development of the translator. When the AST is sufficiently close to the target language, a pretty printer is used to output the target specification.

We refer to our translator framework as a product family since most transformation passes are reused in the translators for each target language. Reuse of the transformation passes makes it much easier to support new target languages; we have developed new translators in a matter of days. The number of transformation passes depends on the similarity of the source and target languages and on the number of optimizations to be made. Our translators range in size from a dozen to over sixty passes.

The translators produce highly optimized specifications appropriate for the target language. For example, when translating to NuSMV the translator eliminates as much redundant internal state as possible, making it very efficient for BDD-based model checking. When translating to the PVS theorem prover, the specification is optimized for readability and to support the development of proofs in PVS. When generating executable C or Ada code, the code is optimized for execution speed on the target processor. These optimizations can have a dramatic effect on the target analysis tools. For example, optimization passes incorporated into the NuSMV translator reduced the time required for NuSMV to check one model from over 29 hours to less than a second.

However, some optimizations are better incorporated into the verification tools rather than the translator. For example, predicate abstraction [3] is a well known technique for reducing the size of the reachable state space, but automating this during translation would require a tight interaction between our translator and the analysis tool to iteratively refine the predicates based on the counter examples. Since many model checkers already implement this technique, we have not tried to incorporate it into our translator framework.

We have developed tools to translate the counterexamples produced by the model checkers into two formats. The first is a simple spreadsheet that shows the inputs and outputs of the model for each step (similar to Table 1 in the next section). The second is a test script that can be read by the Reactis tool to step forward and backward through the counterexample in the Reactis simulator.

Our translator framework currently supports input models written in Simulink, Stateflow and SCADE. It generates specifications for the NuSMV, SAL, and Prover model checkers, the PVS and ACL2 theorem provers, and C and Ada source code.

A Small Example

To make these ideas concrete, we present here a very small example, the mode logic for a simple microwave oven shown in figure 2. The microwave initially starts in *Setup* mode. It transitions to *Running* mode when the *Start* button is pressed and the *Steps Remaining* to cook (initially provided by the keypad entry subsystem) is greater than zero. On transition to *Running* mode, the controller enters either the *Cooking* or *Suspended* submode, depending on whether *Door Closed* is true. In *Cooking* mode, the controller decrements *Steps Remaining* on each step. If the door is opened in *Cooking* mode or the operator presses the *Clear* button, the controller enters the *Suspended* submode. From the *Suspended* submode, the operator can return to *Cooking* submode by pressing the *Start* button while the door is closed, or return to *Setup* mode by pressing the *Clear* button. When *Steps Remaining* decrements to zero, the controller exits *Running* mode and returns to *Setup* mode.

Since this model consists only of Boolean values (*Start, Clear, Door Closed*), enumerated types (*mode*), and two small integers (*Steps Remaining* and *Steps to Cook* range from 0 to 639, the largest

value that can be entered on the keypad) it is well suited for analysis with a symbolic model checker

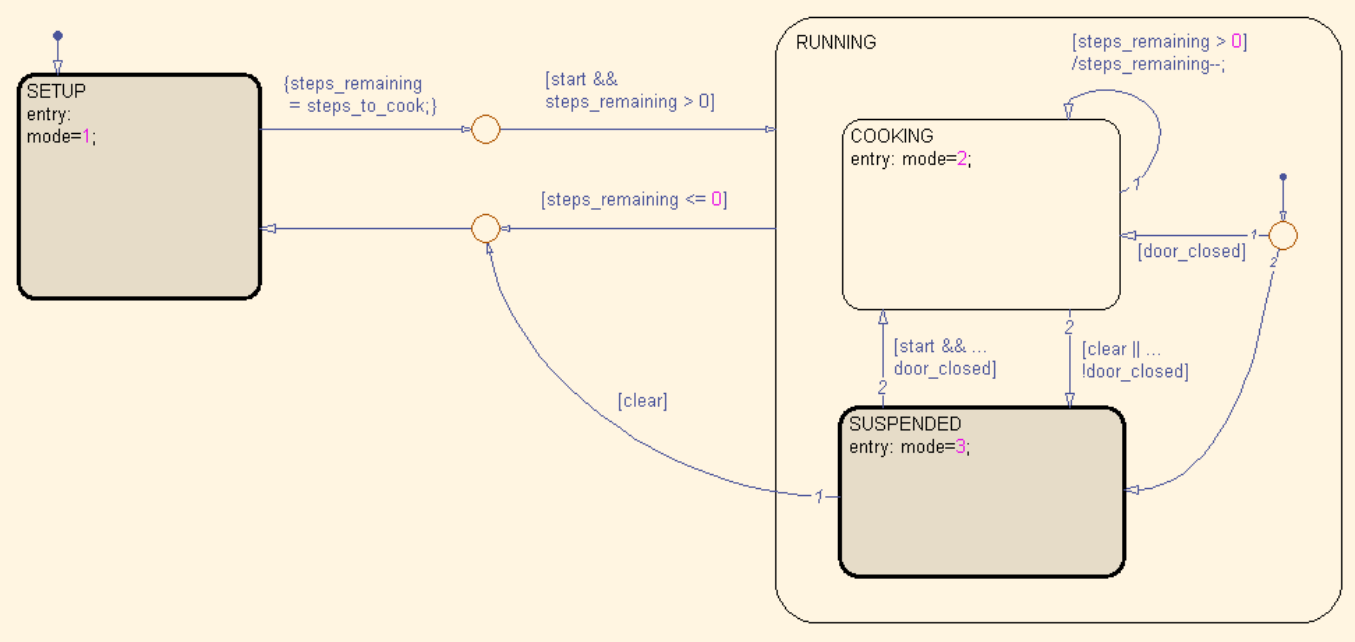

FIGURE 2 – MICROWAVE MODE LOGIC

such as NuSMV. A valuable property to check is that the door is always closed when the microwave is cooking. In CTL [3] (one of the property specification languages of NuSMV), this is written as

```
AG(Cooking -> Door_Closed)
```

Translation of the model into NuSMV and checking this property takes only a few seconds and yields the counter example shown in Table 1.

TABLE 1 – COUNTER EXAMPLE

Step	1	2	3
Inputs			
Start	0	1	0
Clear	0	0	0
Door Closed	0	1	0
Steps to Cook	0	1	1
Outputs			
Mode	Setup	Cooking	Cooking
Steps Remaining	0	1	0

In step 2 of the counter example, we see the value of *Start* change from 0 to 1, indicating the start button was pressed. Also in step 2, the door is closed and *Steps Remaining* takes on the value 1. As a result, the microwave enters *Cooking* mode in step 2. In step 3, the door is opened, but the microwave remains in *Cooking* mode, violating our safety property.

To better understand how this happened, we use Reactis to step through the generated counter-example. This reveals that instead of taking the transition from *Cooking* to *Suspended* when the door is opened, the microwave took the transition from *Cooking* to *Cooking* that decrements *Steps Remaining* because this transition has a higher priority (priority 1) than the transition from *Cooking* to *Suspended* (priority 2). Worse, the microwave would continue cooking with the door open until *Steps Remaining* becomes zero. Changing the priority of these two transitions and rerunning the model checker shows that in all possible states, the door is always closed if the microwave is cooking.

While this example is tiny, the two integers (*Steps Remaining* and *Steps to Cook*) still push its reachable state space to 9.8×10^6 states. Also note that the model checker does not necessarily find the "best" counterexample. It actually would have been clearer if the *Steps Remaining* had been set to a value larger than 1 in step 2. However, this counterexample is very typical. In the production models that we have examined, very few counterexamples are longer than a few steps.

Case Studies

Of course, to be of any real value, model checking must be able to handle much larger problems. Three case studies on the application of our tools to industrial examples are described here. A fourth case study is discussed in [11].

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 [12]. In modern aircraft, the main way that aircraft status is provided to pilots is through computerized display panels similar to those shown in.Figure 3. 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 contains logic errors, critical flight information could be unavailable to the flight crew.

Figure 3 – Pilot Display Panels

While very complex, the WM is specified in Simulink using only Booleans and enumerated types, making it ideal for verification using a BDD based 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 instances of Simulink subsystems. The reachable state space of the five components ranges from 9.8×10^{9} to 1.5×10^{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. 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 Phase I

Our second case study was sponsored by the Air Force Research Laboratory (AFRL) under the Certification Technologies for Advanced Flight Critical Systems (CerTA FCS) program in order to compare the effectiveness of model checking and testing [13]. 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. Phase I of the project concentrated on applying our tools to the Redundancy Management (RM) logic, which is based almost entirely on Boolean and enumerated types

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 2.1×10^{4} to 6.0×10^{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.

To compare the effectiveness of model checking and testing at discovering errors, this project had two independent verification teams, one that used testing and one that used model checking. 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. The conclusion of both teams was that model checking was shown to be more cost effective than testing in finding design errors.

CerTA FCS Phase II

The purpose of 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, the translation framework and model checking tools were used to verify important 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 3×6 matrix of floating point numbers. It inputs 32 floating point inputs and a 3×6 matrix of floating point numbers and outputs a 1×6 matrix of floating point numbers. It contains over 2,000 basic Simulink blocks organized into 166 Simulink subsystems, many of which are Stateflow models.

Because of its extensive use of floating point numbers and large state space, the EB cannot be verified using a BDD-based model checker such as NuSMV. Instead, the EB was analyzed using the Prover SMT-solver from Prover Technologies. Even with the additional capabilities of Prover, several new issues had to be addressed, the hardest being dealing with floating point numbers.

While Prover has powerful decision procedures for linear arithmetic with real numbers and bit-level decision procedures for integers, it does not have decision procedures for floating point numbers. Translating the floating point numbers into real numbers was rejected since much of the arithmetic in the EB is inherently non-linear. Also, the use of real numbers would mask floating point arithmetic 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. While this allowed the EB to be verified using Prover's bit-level integer decision procedures, the results were unsound due to the loss of precision. However, if errors were found in the verified model, their presence could easily be confirmed in the original model. This allowed the verification to be used as a highly effective debugging step, even though it did not guarantee correctness.

Determining what properties to verify was also a difficult problem. The requirements for the EB are actually specified for the combination of the EB and the aircraft model, but checking both the EB and the aircraft model exceeded the capabilities of the Prover Plug-In model checker. After consultation with the OFP designers, the verification team decided to verify whether the six actuator commands would always be within dynamically computed upper and lower limits. Violation of these properties would indicate a design error in the EB logic.

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 then verified using Prover Plug-In and their composition was manually verified using manual proofs. This approach also ensured that unsoundness could not be introducedthrough 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 these properties. In addition, several potential errors that were being masked by defensive design practices were found and corrected.

Lessons from the Case Studies

The case studies described in this paper demonstrate that model checking can be effectively used to find errors early in the development process for many classes of models. In particular, even very complex models can be verified with BDD-based model checkers if they consist primarily of Boolean and enumerated types. Every industrial system we have studied contains large sections that either meet this constraint or that can be made to meet it with some alteration.

For this class of models, the tools are simple enough for developers to use them routinely and without extensive training. In our experience, a single day of training and a low level of ongoing mentoring are usually sufficient. This also makes it practical to perform model checking early in the development process while a model is still changing.

Running a set of properties after each model revision is a quick and easy way to see if anything has been broken. We encourage our developers to “check your models early and check them often.” The time spent model checking is recovered several times over by avoiding rework during unit and integration testing.

Since model checking examines every possible combination of input and state, it is also far more effective at finding design errors than testing, which can only check a small fraction of the possible inputs and states. As demonstrated by the CerTA FCS Phase I case study, it can also be more cost effective than testing.

Future Directions

There are many directions for further research. As illustrated in the CerTA FCS Phase II study, numerically intensive models still pose a challenge for model checking. SMT-based model checkers

hold promise for verification of these systems, but the need to write properties that can be verified through induction over the state transition relation make them more difficult for developers to use.

Most industrial models used to generate code make extensive use of floating point numbers. Other models, particularly those that deal with spatial relationships such as navigation, make extensive use of trigonometric and other transcendental functions. A sound and efficient way of checking systems using floating point arithmetic and transcendental functions would be very helpful.

It can also be difficult to determine how many properties need to be checked. Our experience has been that checking even a few properties will find errors, but that checking more properties will find more errors. Unlike testing for which many objective coverage criteria have been developed, completeness criteria for properties do not seem to exist. Techniques for developing or measuring the adequacy of a set of properties are needed.

As discussed in the CerTA FCS Phase II case study, the verification of very large models may be achieved by using model checking on subsystems and more traditional reasoning to compose the subsystems. Combining model checking and theorem proving in this way could be a very effective approach to the compositional verification of large systems.

Acknowledgements

Many individuals have contributed to the work described in this paper. The authors wish to thank Ricky Butler, Celeste Bellcastro, and Kelly Hayhurst of the NASA Langley Research Center, Mats Heimdahl, Yunja Choi, Anjali Joshi, Ajitha Rajan, Sanjai Rayadurgam, and Jeff Thompson of the University of Minnesota, Eric Danielson, John Innis, Ray Kamin, David Lempia, Alan Tribble, Lucas Wagner, and Matt Wilding 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.

References

[1] The Mathworks, Simulink Product Description, http://www.mathworks.com

[2] Esterel Technologies. SCADE Suite Product Description, http://www.estereltechnolgies.com.

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

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

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

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

[7] Prover Technology, Prover Plug-In Product Description, http://www.prover.com.

[8] Steven Miller, Alan Tribble, Michael Whalen, Mats P. E. Heimdahl, *Proving the Shalls*, International Journal on Software Tools for Technology Transfer (STTT), February 2006.

[9] N. Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud, The Synchronous Dataflow Programming Language Lustre, Proceedings of the IEEE, 79(9):1305- 1320, 1991.

[10] Reactive Systems, inc, http://www.reactive-systems.com.

[11] 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.

[12] 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.

[13] Michael Whalen, Darren Cofer, Steve Miller, Bruce Krogh, and Walter Storm, Integration of Formal Analysis into a Model-Based Software Development Process, Proceedings of the 12th International Workshop on Formal Methods for Industrial Critical Systems (FMICS2007), Berlin, Germany, July 1-2, 2007.