Bridging the Gap Between Model-Based Development and Model Checking*

Steven P. Miller

Rockwell Collins, Cedar Rapids IA 52498, USA
spmiller@rockwellcollins.com

Abstract. The growing power of model checking is making it feasible to use formal verification for important classes of software systems. However, for this to be practical it is necessary to bridge the gap between the commercial modeling tools industrial developers prefer to use and the input languages of the formal verification tools. This paper describes a translator framework that makes it possible to use several popular formal verification tools with commercial modeling tools. The practicality of this approach is illustrated by four case studies in which model checking was successfully used in the development of avionics software.

1 Introduction

Great strides have been made in the development of model checking tools over the last few years. However, there have been relatively few instances reported of their successful application to industrial problems outside of the realm of hardware engineering. In fact, software and system engineers are often completely unaware of the opportunities these tools offer.

One of the main reasons for this has been the difficulty of producing software or system design models that can be analyzed by these tools. Typically, users of a model checker must first create a separate model in the input language of the model checker that they believe replicates the behavior of the original design. Besides introducing significant cost and delay, this also undermines the developer's confidence in the analysis since it is not performed on the actual code or design.

This situation is rapidly changing with the growing popularity of Model-Based Development (MBD) for the design of embedded systems. Tools such as MATLAB Simulink® [1] and Esterel Technologies SCADE Suite™ [2] are achieving widespread use in the avionics and automotive industry. The graphical models produced by these tools provide a formal, or nearly formal, specification that is often amenable to formal analysis.

* This work was supported in part by the NASA Langley Research Center under contract NCC-01001 of the Aviation Safety Program (AvSP) and the Air Force Research Lab under contract FA8650-05-C-3564 of the Certification Technologies for Advanced Flight Control Systems program (CerTA FCS) 88ABW-2009-0146.

S. Kowalewski and A. Philippou (Eds.): TACAS 2009, LNCS 5505, pp. 443–453, 2009.

This paper describes a translator framework developed by Rockwell Collins and the Critical Systems Research Group at the University of Minnesota that bridges this gap and allows production Simulink and SCADE models to be automatically translated to a variety of popular model checkers and theorem provers. Four case studies are presented in which model checking was used to find errors in early requirements and design models, sometimes years before the final code could be integrated and tested on a system rig.

2 Background

The value proposition for formal verification is changing due to the convergence of two trends, the growing popularity of Model-Based Development for the development of embedded systems and the growing power of model checkers. This section provides a brief introduction to Model-Based Development and to model checking.

2.1 Model-Based Development

Model-Based Development refers to the use of domain specific, graphical modeling languages that can be executed and analyzed before the actual system is built. MBD allows developers to create a model of a system, execute it on their desktop, analyze it with automated tools for the required behavior, and use it to automatically generate code and test cases. In the automotive and avionics industry, MBD generally refers to the use of synchronous data flow languages such as MATLAB Simulink or Esterel Technologies SCADE Suite. Synchronous languages latch their inputs at the start of a computation step, compute their outputs and the next system state as a single atomic step, and communicate between components using data flow signals. This differs from the more general class of modeling notations that include support for asynchronous execution of components and communication using message passing.

2.2 Model Checking

Model checkers are formal verification tools that evaluate an input 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 will produce a counterexample showing how the property can be falsified. While model checkers cannot be used to verify as large a class of models as theorem provers, they are often easier to use and in many cases can provide results in seconds or minutes. This makes them very attractive for use in industrial settings where software designers may not have the expertise or the time to complete a proof using a mechanical theorem prover.

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 such NuSMV [5] use compact

representations (such as Binary Decision Diagrams) of sets of states to describe regions of the model state space that satisfy the properties being evaluated. This often allows them to handle much larger state spaces than explicit state model checkers. Satisfiability modulo theories (SMT) model checkers such as SAL [6] and Prover® Plug-In [7] use a form of induction to reason about models containing real numbers and unbounded arrays. While SMT-based model checkers can deal with infinite state systems, their properties need to written in such a way that they can be proven by induction over an unfolding of the state transition relationship. For this reason, they tend to be more difficult to use than explicit and implicit state model checkers.

3 The Translator Framework

To bridge the gap between industrial modeling tools and some of the more popular model checkers and theorem provers, Rockwell Collins and the Critical Systems Research Group at the University of Minnesota developed a product family of translators [8] as part of the NASA Aviation Safety Program. An overview of this translator framework is shown in figure 1.

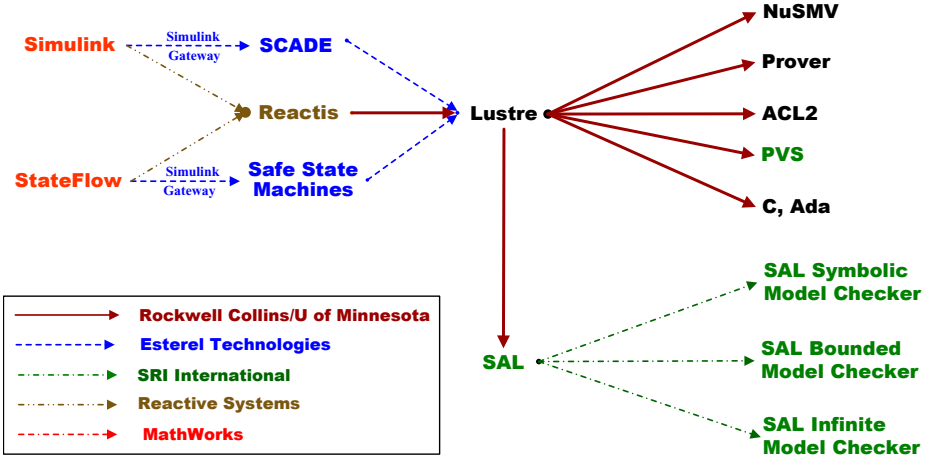

Fig. 1. Translator Framework

The translators work primarily with the Lustre formal specification language [9], but this is hidden from the tool users. The typical user first creates a model in Simulink, StateFlow, or SCADE Suite. Since Lustre is the underlying specification language of SCADE, the initial translation into Lustre is immediate for SCADE models. Simulink and StateFlow users can translate their models into Lustre using either the Simulink Gateway provided by Esterel Technologies or by importing their models into the Reactis® [10] test case generator developed by Reactive Systems and using a translator developed by Rockwell Collins. While Simulink does not have a full, formal semantics, developers of safety-critical

systems routinely restrict themselves to a safe subset of the language and it is usually possible to assign a formal semantics to this subset.

Once in Lustre, the specification is read into an abstract syntax tree (AST) and a number of transformation passes are applied to it. Each transformation pass produces a valid 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.

The translator framework is actually a product family of translators in that many transformation passes are reused in the translators for each target language. Pre-conditions for each transformation specify the properties a Lustre specification must satisfy for the translation to be valid and post-conditions define the properties of the generated specification. Reuse of the transformation passes makes it much easier to support a variety of target languages and allows new translators to be developed in a matter of days. The number of transformation passes depends on how similar the source and target languages are and on how many optimizations need to be made. Currently, the number of transformation passes ranges from a dozen for a simple C code generator to over sixty for an optimized translation to NuSMV.

The translators produce highly optimized models appropriate for the target language. For example, when translating to the NuSMV model checker, the translator produces a specification that is difficult for a human to read, but very efficient for 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 code, the translation is optimized for execution speed on the target processor. Many of these optimizations can have a dramatic effect on the target analysis tools. For example, optimization passes incorporated into the NuSMV translator reduce the time required for NuSMV to check one model from over 29 hours to less than a second, an improvement of over five orders of magnitude.

Tools have also been developed to present the counter examples produced by the model checkers into two formats that are easier to understand. The first is a simple spreadsheet format that shows the inputs and outputs of the model for each step of the counter example. The second is a test script that can be used to step the Reactis tool forward and backward through the counter example. As shown in figure 1, the translator framework currently supports input models written in MATLAB Simulink and Stateflow and Esterel Technologies SCADE Suite and generates models for the NuSMV, SAL, and PROVER model-checkers, the PVS and ACL2 theorem provers, and C and Ada source code.

4 Case Studies

This section describes four case studies in which model checking was used to find errors in early requirements and design models. The use of industrial models has proven invaluable in selecting the features to be added to the translator framework.

4.1 FCS 5000 Mode Logic

The first application of model checking to an actual product at Rockwell Collins was to the mode logic of the FCS 5000 Flight Control System [11]. The FCS 5000 is a family of Flight Control Systems (FCS) developed by Rockwell Collins for use in business and regional jet aircraft. The Flight Guidance System (FGS) is a component of the FCS that compares the measured state of an aircraft (position, speed, and attitude) to the desired state and generates pitch and roll guidance commands to minimize the difference between the measured and desired state. The mode logic is a component of the FGS that determines which lateral and vertical flight modes are armed and active at any time.

While inherently complex and difficult to get right, the mode logic consists almost entirely of Boolean and enumerated types. As described in [11], Rockwell Collins developed an in-house format that produces a very compact specification of the mode logic that can be directly implemented in Simulink. This made the FCS 5000 mode logic ideally suited for analysis using the translator framework and a symbolic model checker such as NuSMV.

The mode logic 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, of the other mode diagrams. While each individual diagram is straightforward to understand, grasping all the interactions between them can be difficult. In fact, the most interesting requirements to be checked defined relationships to be maintained between the mode machines, for example, ensuring that the active vertical flight mode was not "Approach" unless the active lateral flight mode was already "Approach".

Analysis of a very early specification of the FCS 5000 mode logic with NuSMV found 26 errors in the mode logic. Seventeen of these were found by the model checker, six were found in the process of translating the informal requirements into the Simulink model, and three were found during inspections performed to develop the properties to be checked. Of the 17 errors found using the model checker, 13 were classified as being possible to be missed by traditional verification techniques such as testing and inspections, and one was classified as being likely to be missed by traditional techniques.

One of the main advantages of this analysis was that it could be done early in the development process when 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.

4.2 ADGS-2100 Window Manager

One of the largest and most successful applications of model checking at Rockwell Collins was to the ADGS-2100 Adaptive Display and Guidance System Window Manager [12]. In modern aircraft, the primary way that aircraft status is provided to the pilots is through computerized display panels in the cockpit. These panels replace the dozens of mechanical switches and dials found in earlier aircraft and present a unified interface to critical flight information.

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 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 (WM) 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 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 made 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. As shown in table 1, 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.

Table 1. Window Manager Analysis Results

Component	Subsystem Instances	Basic Blocks	Reachable State Space	Number of Properties	Errors Found
GG	2,831	10,699	9.8×10^9	43	56
PS	144	398	4.6×10^{23}	152	10
CM	139	1,009	1.2×10^{17}	169	10
DUF	879	2,941	1.5×10^{37}	115	8
MFD	301	1,100	6.8×10^{31}	84	14
Totals	4,295	16,117		563	98

To begin the analysis, a set of properties that formally state the WM requirements were develped in CTL, one of the property specification languages of NuSMV. Developing the properties to be checked was a gradual process of studying the WM requirements and talking with the WM developers. Some properties were straightforward to write. For example, in a simplified version of the WM with only two Display Units (DU), the requirement

If a DU is available, then it shall display some application.

would be stated as the two CTL properties

```
AG(LEFT_DU_AVAILABLE -> LEFT_DU_APPLICATION != BLANK)

AG(RIGHT_DU_AVAILABLE -> RIGHT_DU_APPLICATION != BLANK)
```

Other properties required discussion with the WM developers to clarify nuances or resolve the ambiguity inherent in English textual requirements.

At the start of the project, the translator chain did not work with the version of Simulink being used by the development team and the models required several hours of hand tweaking before they could be translated from Simulink to NuSMV. As a result, the early analysis was done entirely by the model checking team.

However, as the project progressed, improvements were made to the tool chain so that the translation only took a few minutes and was completely automated. Also, optimizations to the translator reduced the time required for NuSMV to check each property to roughly 20 seconds. Gradually, the developers began to see that model checking could find errors faster, more easily, and more thoroughly than testing or reviews. This motivated them to start writing and checking CTL properties on their own. Eventually, the developers completely took over the model checking and began relying on the model checking team only for consultation and tool improvements.

Ultimately, 563 properties about the WM were developed and checked, and 98 design errors in the model were found and corrected (see table 1). As with the FGS mode logic, this verification was done early in the design process as the design and the requirements were still evolving. In fact, by the end of the project, the WM developers were checking the properties several times each day, usually after each design change.

4.3 CerTA FCS Phase I

The third case study was sponsored by the Air Force Researh Labs (AFRL) Wright Patterson RD Directorate under the Certification Technologies for Advanced Flight Control Systems (CerTA FCS) program [13]. In this study, the translation framework and model checking tools were applied to the Operational Flight Program (OFP) for an Unmanned Aerial Vehicle (UAV) created by Lockheed Martin Aero. The OFP is an adaptive flight control system that modifies its behavior in response to flight conditions.

Phase I of the project focused on investigating the roles of testing and formal verification, and in particular, determining if formal verification could be used to replace some tesing. To this end, two verification teams were set up. One team, based at Lockheed Martin, focused on traditional testing of the OFP. The other team, based at Rockwell Collins, focused on the use of model checking. Neither team communicated directly with the other team and both teams started with identical models and specifications of the requirements.

To ensure the effectiveness of testing was being compared to a mature formal verification technology, the model checking in Phase I was restricted to the Redundancy Management (RM) logic of the OFP. Like the FCS 5000 mode logic and the ADGS-2100 WM, the RM logic is based almost entirely on Boolean and enumerated types. This makes it ideal for analysis with a BDD-based model checker such as NuSMV. However, the RM logic also contained several model constructs that had not been encountered in the FCS 5000 mode logic and the

ADGS-2100 WM, including Stateflow models and truth tables. Extensions to the translator framework to support these features took about two thirds of the total time spent model checking the RM logic. However, the Lockheed Martin team also made comparable investments in enhancing their testing environment. These one time, non-recurring costs were factored out of the final comparison of the effectiveness of testing and model checking.

Like the ADGS-2100 WM, the RM logic is organized into three components that could be analyzed individually (see table 2). While these components are smaller than those in the ADGS-2100 WM, they are replicated once for each of the ten control surfaces on the aircraft and collectively represent a significant portion of the OFP logic.

Table 2. OFP Redundancy Manager Analysis Results

Component	Subsystem Instances	Basic Blocks	Charts/ Transitions/ TT Cells	Reachable State Space	Number of Properties	Errors Found
Triplex Voter	10	96	3/35/198	6.0×10^{13}	43	5
Failure Processing	7	42	0/0/0	2.1×10^{4}	6	3
Reset Manager	6	31	2/26/0	1.3×10^{11}	8	4
Totals	23	169	5/61/198		62	12

To compare the effectiveness of model checking and testing at discovering errors, the formal verification team developed a total of 62 properties from the OFP requirements. While these properties only partially specifed the required behavior of the RM logic, checking them with the model checker uncovered 12 errors in the RM logic. Of these 12 errors, four were classified 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 that 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, including those found through model checking. The conclusion of both teams was that in this case, model checking was more effective than testing in finding design errors.

4.4 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 piece of logic 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 Simpulink blocks organized into 166 Simulink subsystems, many of which are Stateflow models.

Because of its extensive use of floating point numbers and enormous 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 in Phase II, 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 verifed using Prover's bit-level integer decision procedures, the results were unsound due to the loss of precision. Even so, if errors were found in the verified model, it was very likely that they would also be found 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 model checker. After extensive consultation with the OFP designers, the verification team decided to verify whether the six actuator commands would always be within a dynamically computed upper and lower limit. Violation of these properties would indicate a design error in the EB logic.

Even with these adjustments, the EB logic 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 and their composition was manually verified using through simple 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 verification of these properties. In addition, several potential errors that were being masked by defensive design practices were found and corrected.

5 Conclusions and Future Directions

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 portions of logic 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 is 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. When combined with the ease of use discussed above, this makes it very cost effective approch to defect detection. As demonstrated by the CerTA FCS Phase I case study, it can be more cost effective than testing.

However, there are still many areas for further research. As illustrated in the CerTA FCS Phase II study, numerically intensive models still pose a challenge for model checking. In fact, even a handful of integers can render BDD-based model checking ineffective. SMT-based model checkers hold great 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. More work is needed to make them simpler and more intuitive.

Most industrial models used to generate code make extensive use of floating point numbers. As discussed in the CerTA FCS Phase II study, simply using real numbers instead of floating point numbers may not be acceptable, either because of the inherent non-linearity of the system or because of the masking of floating point arithmetic errors. Other models, particularly those that deal with spacial relationships such as navigation, make extensive use of trigonometric and other transcendental functions. A simple way of model checking such systems 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 [14], completeness criteria for properties do not seem to exist. Techniques for developing or measuring the adequacy of a set of properties would be very helpful, particularly when seeking certification credit for the use of formal methods.

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, but introducing even this limited use of theorem proving into an industrial development process poses many challenges unless it can be made quicker and more intuitive.

Finally, most safety critical systems must be designed using redundancy to meet their reliability requirements. These systems are typically implemented as globally asynchronous/locally synchronous systems in which synchronous components, each with their own clock, communicate asynchronously with each other. Verification of such quasi-synchronous systems [15] pose many challenges to model checking. However, these are also precisely the type of systems that would benefit the most from a formal approach to verification.

References

1. The Mathworks, Simulink Product Description, `http://www.mathworks.com`
2. Esterel Technologies, SCADE Suite Product Description, `http://www.estereltechnolgies.com`
3. Clarke, E., Grumberg, O., Peled, D.: Model Checking. The MIT Press, Cambridge (2001)
4. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, Reading (2003)
5. 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. Miller, S., Tribble, A., Whalen, M., Heimdahl, M.: Proving the Shalls. International Journal on Software Tools for Technology Transfer (STTT) 8(4-5), 303–319 (2006)
9. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The Synchronous Dataflow Programming Language Lustre. Proceedings of the IEEE 79(9), 1305–1320 (1991)
10. Model-Based Testing and Validation with Reactis, `http://www.reactive-systems.com`
11. Miller, S., Anderson, E., Wagner, L., Whalen, M., Heimdahl, M.: Formal Verification of Flight Critical Software. In: Proceedings of the AIAA Guidance, Navigation and Control Conference and Exhibit, AIAA-2005-6431. American Institute of Aeronautics and Astronautics (2005)
12. Whalen, M., Innis, J., Miller, S., Wagner, L.: ADGS-2100 Adaptive Display & Guidance System Window Manager Analysis. NASA Contractor Report CR-2006-213952 (2006), `http://shemesh.larc.nasa.gov/fm/fm-collins-pubs.html`
13. Whalen, M., Cofer, D., Miller, S., Krogh, B., Storm, W.: Integration of Formal Analysis into a Model-Based Software Development Process. In: Leue, S., Merino, P. (eds.) FMICS 2007. LNCS, vol. 4916, pp. 68–84. Springer, Heidelberg (2008)
14. Chilenski, J., Miller, S.: Applicability of Modified Condition/Decision Coverage to Software Testing. IEE Software Engineering Journal 9(5), 193–200 (1994)
15. Tripakis, S., Pinello, C., Benveniste, A., Sangiovanni-Vincent, A., Caspi, P., Di Natale, M.: Implementing Synchronous Models on Loosely Time Triggered Architectures. IEEE Transactions on Computers 57(10), 1300–1314 (2008)