# Partial Translation Verification for Untrusted Code-Generators[⋆]

Matt Staats
Mats P.E. Heimdahl

Dept. of Comp. Sci. and Eng.
University of Minnesota
{staats, heimdahl}@cs.umn.edu

**Abstract.** Within the context of model-based development, the correctness of code generators for modeling notations such as Simulink and Stateflow is of obvious importance. If correctness of code generation can be shown, the extensive and often costly verification and validation activities conducted in the modeling domain could be effectively leveraged in the code domain. Unfortunately, most code generators in use today give no guarantees of correctness.
In this paper, we investigate a method of leveraging existing model checking tools to verify the partial correctness of code generated by code generators that offer no guarantees of correctness. We explore the feasibility of this approach through a prototype tool that allows us to verify that Linear Temporal Logic (LTL) safety properties are preserved by C code generators for Simulink models. We find that the approach scales well, allowing us to verify that 55 LTL properties are maintained when generating 12,000+ lines of C code from a large Simulink model.

## 1 Introduction

Tools translating a source language to a target langauge are probably the most used tools in software development. Of particular interest in this article are code generators for modeling notations extensively used in the critical systems domain, such as Simulink and Stateflow [16, 17], and the SCADE tools-suite [8]. In this domain it would be highly desirable if the extensive and often costly verification and validation activities conducted in the modeling domain could be effectively leveraged in the code domain; in other words, if we have have high confidence in the correctness of the model, it would be nice if we could have high confidence in the generated code. Unfortunately, these code generators come with no guarantees of correctness—they are either black-boxes of unknown (though generally good) quality, or have been developed in in-house to address specific needs in terms of generated-code footprint, performance, or applicability to a specific hardware platform.

⋆ This work has been partially supported by NASA Ames Research Center Cooperative Agreement NNA06CB21A, NASA IV&V Facility Contract NNG-05CB16C, and the L-3 Titan Group.

While interest in the correctness of translators such as traditional compilers and code generators has existed nearly as long as translators themselves [14, 18], we still lack the ability to cost-effectively prove the correctness of industrial translators. The automation of correctness proofs is still out of reach, while manual proofs of correctness are both large and difficult (if not entirely infeasible). A different—and potentially more practical—approach to verification is to check individual translations for correctness (as opposed to the translator itself) [22, 24, 25]. These tools generally rely on a proof of correctness being emitted during the translation process (to be checked by a proof checker). Nevertheless, while this approach is promising, is has not seen adoption in industry. Therefore, for the foreseeable future, we will have to rely on code-generators where we cannot fully trust the generated code.

In this paper, we investigate a method of leveraging existing model checking tools to verify the partial correctness of generated code from a translator which offers no guarantees of correctness (we term such a translator an *untrusted translator*). This method operates by checking individual translations (as above) but does not require changes to the translator to do so. We investigate the feasibility of the approach through a prototype tool that verifies that Linear Temporal Logic (LTL) properties [19, 20] are preserved in the translation of a Simulink model to C code.

The intuition behind the prototype is as follows: in collaboration with Rockwell Collins Inc., we have demonstrated how to verify large numbers of model requirements captured as Linear Temporal Logic (LTL) properties [19, 20] over Simulink models. If we could re-verify these properties on the auto-generated C code, we would be able to provide a proof that at least the *essential properties* captured in the modeling domain are preserved in the translation from Simulink to C; the generated C code may not be fully correct with respect to the original Simulink model, but it will be *correct enough* to preserve crucial safety-critical required properties.

Our prototype tool is based on two exisiting model checkers: the NuSMV model checker [23] for the verification of Simulink models, and the ANSI-C Bounded Model Checker (CBMC) [3] for the verification of generated C code. Our tool operates by automatically translating the LTL properties expressed in the Simulink domain to monitors in the C domain, thus allowing us verify the same set of LTL properties over both the model and the generated code.

Our primary interest in this investigation was to determine if this approach would scale to address a production-sized Simulink model. Based on our experiences reported in this paper, CBMC scaled remarkably well, allowing us to easily re-verify a collection of 55 required properties on 12K+ lines of generated C code. Based on these results, we believe this approach to partial verification of auto-generated code holds great promise and we will explore the scalability further on commercial systems in future work.

The remainder of the paper is organized as follows. Section 2 contains an overview of the verification method and our proof of concept and walks through a small example. Section 3 discusses the effectiveness of our approach three

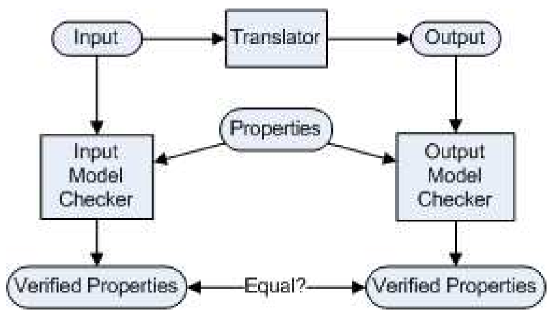

**Fig. 1.** General Approach to Translation Verification

industrial examples. Section 4 compares our work with previous related work, and Section 5 concludes the paper.

## 2 Verification of Individual Translations

As previously mentioned, verifying properties of correctness over individual translations has been shown to be a practical method of gaining guarantees of translator correctness. Current approaches to verifying individual translations have been used successfully to verify that a variety of properties are preserved during translation, ranging from type and memory safety of compiled C programs [22] to complete equivalence of SIGNAL programs translated to C [24].

These approaches are often applied to each transformation pass performed by the translator, thus inductively proving that properties of interest are maintained during translation [25]. Most existing approaches rely on instrumenting the translator to either verify that properties are maintained during translation or provide a proof of such (which can then be verified by a proof checker). These methods are therefore of little use when an untrusted translator must be used.

In the remainder of this section, we will outline how existing model checking tools can be leveraged to provide guarantees of correctness for translations generated from untrusted translators, and describe a prototype we have developed for verifying translations from Simulink to C.

### 2.1 Overview of Verification

As shown in Figure 1, the general method uses three principal tools: two model checkers capable of verifying properties of interest on the input and output of the translator, respectively, and the translator itself. Two inputs are required: the translator input and a set of properties we wish to show are preserved by translation.

The method is simple. The input is first processed by the translator to produce an output. Then, the set of properties are verified on both the input and output by the appropriate model checkers, producing two lists of verified properties. These lists are compared, and any properties which fail to verify on only the input or output are reported, as they indicate that the translation either failed to preserve a property, or—in the case when a property holds on the output but not the input—removed one or more possible behaviors in the translation process. Should no such discrepancies exist, the translation has been proven to preserve the given set of properties.

Note that while we are primarily interested in this method for use in verifying the generated code, the method is applicable for any translation in which the same set of properties can be verified over both the input and output of a translator.

### 2.2 Prototype Implementation

Our prototype concerns the verification of C code generators for Simulink. We perform verification over sets of safety properties defined as Linear Temporal Logic (LTL) [10]. Note that a *safety property* is one that states that "something bad never happens". This is in contrast to liveness properties, which state that "something good eventually happens". We focus our efforts on safety properties for two reasons: first, it is considerably easier to verify safety properties, particularly when using bounded model checking [2]. Second, for our sample systems, all properties are expressed as safety properties, and thus the ability to verify safety properties is sufficient for our goal.

For our prototype, model checking is done by the NuSMV model checker [23] and the ANSI-C Bounded Model Checker (CBCM) [3] for Simulink [16] and C, respectively. NuSMV directly allows us to check LTL properties against finite state systems. In previous work [19, 20], we developed a Simulink to NuSMV translator with Rockwell Collins Inc. This translator allows us to effectively verify the LTL properties over Simulink models, thus making NuSMV an good fit for our task.

CBMC, however, does not directly allow us to check LTL properties over C code. CBMC is designed to statically check a variety of properties such as pointer safety and safe use of arrays, as well as user-specified assertions. We therefore must express LTL properties as assertions to verify them using CBMC. We do this by translating each LTL property into a monitor (expressed as C code) that indicates violations of the LTL property using assertions.

To demonstrate the construction of a monitor, consider the following requirement from a Flight Guidance System (FGS) (this sample system is outlined in Section 3.1):

```
G((!Onside_FD_On & !Is_AP_Engaged) ->
    X(Is_AP_Engaged -> Onside_FD_On))
```

This requirement states the following: "If the onside FD cues are off, the onside FD cues shall be displayed when the AP is engaged." (In this example, the FD refers to the Flight Director – a part of the pilot's primary flight display

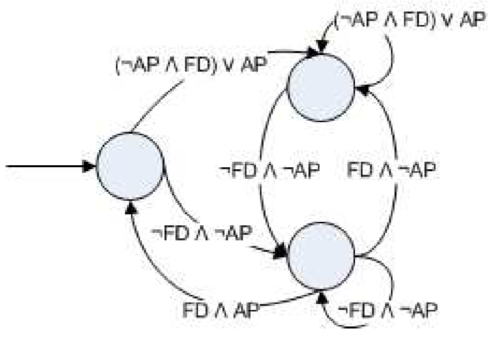

**Fig. 2.** LTL Property Expressed as TGBA. Note that AP represents Is_AP_Engaged and FD represents Onside_FD_On.

– and the AP refers to the Auto Pilot.) Formally, the LTL formula states that it is always the case (**G**) that if the Onside FD is not on and the AP is not engaged, in the next instance in time (**X**) if the AP is engaged the Onside FD will also be on.

To convert this LTL property to a C monitor based on assertions, we first use the Spot [6] library to translate the LTL property to a Transition-based Generalized Büchi Automata (TGBA), seen in Figure 2. A TGBA is simply a Büchi automaton in which acceptance transitions are used rather than acceptance states; an input sequence is accepted if it causes an acceptance transition to be visited infinitely often [12]. Note that for this LTL property, all transitions are acceptance transitions (the reason for this is explained below).

Next, we replace the variable names taken from the LTL property with the variable names present in the generated C code (this step is specific to the naming conventions used in the translator). For this example, we use the naming conventions present in the Real-Time Workshop C code generator, available for Simulink [16]. For the example LTL property, this renaming is reflected in the C monitor described next.

Finally, we insert the C monitor into the generated C code. Note that the C code generated from our Simulink models is intended to run as a recurring task polling the environment input data. The generated C code operates by repeatedly performing three steps: poll the input, update the internal state, and produce output. LTL properties are defined over a sequence of *consistent* states; we are not interested in the system state in the middle of the next state computation, we only care when the computation has been completed. In the context of the generated C code, these consistent states occur after the output is produced and before new input is received. We therefore insert monitors between the portions of C code which produce output and receive new input.

We map the transitions defined by the TGBA to a sequence of *if* statements. Each *if* statement corresponds to a single transition in the TGBA, and the current state of the TGBA is stored as an integer. When a transition is taken,

```
/*LTL Formula #0
  ( G (((! Onside_FD_On) &    (! Is_AP_Engaged))
       -> ( X (Is_AP_Engaged -> Onside_FD_On)))) */
 if ((ltlstate0 == 1) && ((!FGS_Y.Is_AP_Engaged && !FGS_Y.Onside_FD_On))) {
      ltlstate0 = 3;
 } else if ((ltlstate0 == 1) && ((!FGS_Y.Is_AP_Engaged
    && FGS_Y.Onside_FD_On) || (FGS_Y.Is_AP_Engaged))) {
      ltlstate0 = 2;
 } else if ((ltlstate0 == 2) && (!FGS_Y.Is_AP_Engaged && !FGS_Y.Onside_FD_On)) {
      ltlstate0 = 3;
 } else if ((ltlstate0 == 2) && ((!FGS_Y.Is_AP_Engaged
    && FGS_Y.Onside_FD_On) || (FGS_Y.Is_AP_Engaged))) {
      ltlstate0 = 2;
 } else if ((ltlstate0 == 3) && (!FGS_Y.Is_AP_Engaged && !FGS_Y.Onside_FD_On)) {
      ltlstate0 = 3;
 } else if ((ltlstate0 == 3) && (!FGS_Y.Is_AP_Engaged && FGS_Y.Onside_FD_On)) {
      ltlstate0 = 2;
 } else if ((ltlstate0 == 3) && (FGS_Y.Is_AP_Engaged && FGS_Y.Onside_FD_On)) {
      ltlstate0 = 1;
 } else { assert(0); }
```

**Fig. 3.** LTL Property Expressed as C Assertion

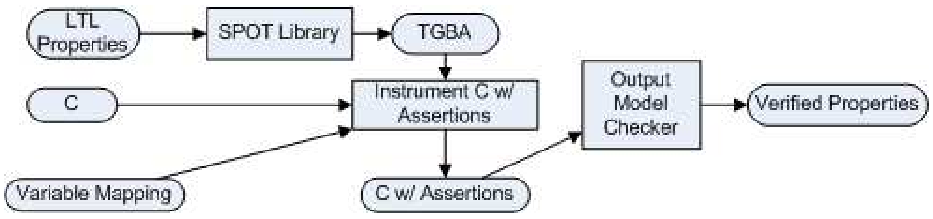

**Fig. 4.** Verifying LTL Properties on C Code

the current state of the TGBA is updated accordingly. During execution of the generated C code, a transition will be taken every "step" based upon the current values of variables in the C code and the current state of the TGBA. If no transition can be taken (indicating an input sequence not accepted by the TGBA), an error is signaled by means of an $assert(0)$ statement. The monitor corresponding to our example property is seen in Figure 3.

Once monitors are created, they are inserted into the C code we wish to verify. CBMC can then statically determine if there exists an execution path which reaches the monitor's assertion, thus checking if the LTL property holds over the C code. An outline of this entire process as performed by our prototype is shown in Figure 4.

Our prototype contains several details worth noting. As mentioned above, to keep our monitor construction simple, we have elected to only verify safety properties. Formally, these are expressed as $\mathbf{G}(p)$, where $p$ is some property which does not use the finally ($\mathbf{F}$) operator. ($\mathbf{F}$ is used to define liveness properties, which we do not support.) For a TGBA corresponding to a safety property, all transitions are accepting transitions, and, thus, any input sequence which causes the TGBA to always correctly transition is accepted. This property is key to our monitor's correctness—if we attempted to capture liveness properties, we might fail to reject an input sequence which always correctly transitions but does not infinitely pass by an accepting transition. Note that it is possible to express liveness properties as safety properties [1], thus allowing us to extend monitor construction to handle liveness properties, though this would require significantly more development effort of our prototype tool. Additionally, expressing liveness properties as safety properties requires the introduction of state recording functionality to the system, likely increasing the size of the searchable state space by a significant amount.

Second, our prototype contains several aspects which are unproven, but must be assumed to work if a user of the prototype wishes to claim a partial proof of correctness. Specifically, errorneous behavior in the variable mapping, monitor creation, or model checkers could lead to either false postives (signaling a property violation in correctly generated code) or false negatives (failing to signal a property violation in incorrectly generated code). However, with the notable exception of the model checkers, the prototype is likely much simpler than the code generator. We therefore believe trusting these prototype components is reasonable.

Finally, note that unlike NuSMV, CBMC is a bounded model checker [3]. Bounded model checkers do not provide a complete proof of correctness for a property unless the search depth of the model checker exceeds the *completeness threshold* [2]. In other words, for a property to be proven to hold on a system with a bounded model checker, a sufficient number of states must have been explored by the model checker. For safety properties that do not use temporal operators besides the initial Globally (i.e., $\mathbf{G}(p)$ where $p$ contains no temporal operators), this threshold is the *reachability diameter* of the model [2]. Using the symbolic model checker in NuSMV and the Simulink model, we can calculate the reachability diameter and use the result to achieve complete verification when using CBMC. Note that it is possible for an incorrect code generator to increase the reachability diameter (though plausible occurrences of this are difficult to formulate); to be conservative, we therefore used a search depth twice the reachability diameter of the Simulink model when verifying properties on the generated C code using CBMC.

For safety properties in which $p$ contains the next operator ($\mathbf{X}$), we use an approach suggested in [2]. Using this approach, we extend the original model with an automaton derived from $p$ (using essentially the same approach used to create C monitors) before calculating the reachability diameter, thus accounting for the state tracking variables introduced by the creation of the C monitors described above.

## 3 Application Results

We applied our prototype to three sample systems (two small toy-examples and one close to production model of the model-logic of a transport class flight guidance system) using two different code generators: Real-Time Workshop, a commercial code generator from Mathworks [16], and a currently in-development code generator courtesy of Rockwell Collins Inc. Our results showed our prototype scaled remarkably well, managing to verify 55 LTL properties over the translation of a Simulink model of significant size.

Initially, we verified single LTL properties over small (but realistic) sample systems. These initial attempts were successful, showing the LTL properties held over the translations in question, and ran very quickly. We then explored the scalability and robustness of our prototype by successfully verifying the translation of a much larger model using a set of 55 LTL properties. These properties used a number of different operators (e.g., the next operator, equivalence operator, implies operator, etc.) and were structured in a variety of ways. Additionally, these properties were defined over a large portion of the input and output variables. We thus feel that these properties give a good indication of the correctness of a translation from Simulink to C, the robustness of our prototype, and the feasibility of the method in general. The models and results are described in more detail below.

## 3.1 Sample Systems

The three sample systems are described below. Measurements relating the size of the systems are given in Table 1. Generated C code lines of code (LOC) counts were performed by SLOCCount [27].

| | **Simulink Nodes** | **System Diameter** | **RTW C LOC** | **In-Dev C LOC** |
|---|---|---|---|---|
| **ASW** | 14 | 3 | 220 | 134 |
| **WBS** | 157 | 2 | 774 | 197 |
| **FGS** | 4510 | 10 | 12,242 | 1,379 |

**Table 1.** Measurements of Sample System Size. The columns labeled RTW and In-Dev refer to the results from Real Time Workshop and the Rockwell Collins code generators respectively.

*Altitude Switch (ASW):* The Altitude Switch (ASW) is a re-useable component that turns power on to a Device Of Interest (DOI) when the aircraft descends below a threshold altitude above ground level (AGL). If the altitude cannot be determined for more than two seconds, the ASW indicates a fault. The detection of a fault turns on an indicator lamp within the cockpit.

*Wheel Brake System (WBS):* The Wheel Brake System (WBS) is a Simulink model derived from the WBS case example found in ARP 4761 [26, 15]. The WBS is installed on the two main landing gears. Braking on the main gear wheels is used to provide safe retardation of the aircraft during the taxiing and landing phases, and in the event of a rejected take-off. Braking on the ground is either commanded manually, via brake pedals, or automatically (autobrake) without the need for pedal application. The Autobrake function allows the pilot to pre-arm the deceleration rate prior to takeoff or landing. When the wheels have traction, the autobrake function will control break pressure to provide a smooth and constant deceleration.

*Flight Guidance System (FGS):* A Flight Guidance System is a component of the overall Flight Control System (FCS) in a commercial aircraft. It compares the measured state of an aircraft (position, speed, and altitude) to the desired state and generates pitch and roll-guidance commands to minimize the difference between the measured and desired state. The FGS consists of the mode logic, which determines which lateral and vertical modes of operation are active and armed at any given time, and the flight control laws that accept information about the aircraft's current and desired state and compute the pitch and roll guidance commands. In this study we have used the model of the mode logic.

### 3.2 Results

For the C code generations performed by both code generators, all LTL properties were verified for each case example. The time to verify properties over the input and output aspects of translations is given in Table 2. All properties were verified on a Intel Centrino Duo machine running at 1.83 GHz with 1 GB RAM.

| | **Real-Time Workshop C** | **In-Development C** | **SMV** |
|---|---|---|---|
| **ASW** | 7 secs. | 8 secs. | 6 secs. |
| **WBS** | 57 secs. | 9 secs. | 2 secs. |
| **FGS** | 1002 secs. | 273 secs. | 66 secs. |

**Table 2.** Time to Verify SMV and Generated C Code for our sample systems.

We feel the FGS is the most interesting case example, being both the largest model and the model with the largest number of associated LTL properties. Translation of the FGS Simulink model yielded over 12,000 lines of source-code with Real Time Workshop and over 1,300 lines of code with the Collins In-Development generator (Table 1). Verification of the LTL properties over the generated C code took approximately 17 and 5 minutes respectively (Table 2). As a reference, verification of the properties over the NuSMV model extracted from the Simulink model took approximately 1 minute. Given the the results for a substantial system such as the FGS (in terms of code size as well as number of properties to verify), we feel that our prototype demonstrates that the method of partial correctness verification of auto-generated C-code using model-checking techniques is both feasible and useful.

## 4 Related Work

The concept of a verifying translator originated in early work by Floyd [9]. This was followed by a number of attempts centered on proving that a translator would produce correct output for any input [7, 13, 21]. While these attempts met with some success, manual proofs are difficult to perform and automatic proofs of translator correctness remain generally out of reach.

More recently, several efforts have focused on proving properties for individual translations correct, rather than proving the translator correct in general. Necula and Lee have developed a compiler for a type-safe subset of C that verifies type and memory safety of the resulting assembler using code annotations [22]. They later extended this idea to the full Java language [4]. Rinard has put forth the logical foundations for "credible compilation" of imperative languages in which a proof is generated showing the equivalence of the compiler's input and output [25]. Pnueli et. al. describe a similar approach to Rinard's and illustrate their approach using a C code generator for the synchronous language SIGNAL [24]. Note that both Pnueli's and Rinard's approaches generate proofs of equivalence between the input and output, while Necula and Lee's approach only produces a proof that the output satisfies certain properties which are known to exist in the input (such as type safety).

These recent efforts share our goal of verifying properties of individual translation, but differ from the approach described in this report in that they are implemented within the translator, and, thus, must be incorporated by the developers of the translator. In contrast, the approach we have presented can be applied to any translator, provided a method of statically verifying a set of properties over both the translator input and output exists (in the case of our prototype, this meant a model checker for both the input and output and knowledge of how to create and insert monitors into the generated C code). Additionally, while these approaches are clearly more feasible than proving a translator correct in general, they do require significant effort on the part of the translator developer, the approach we have presented requires no changes to the translator itself.

Denney and Fischer have developed a method of automatically annotating generated code without modifying the translator [5]. This method achieves similar goals in that individual code generations can be statically verified without the need to instrument the code generator itself. However, method of static verification used differs - rather than directly verifying a set properties are true for generated code, Denney and Fischer's method instead opts to annotate the generated code such that the correctness of the annotations implies the correctness of one or more safety properties. Once the code is annotated, a proof checker is used to verify the annotations.

The construction of C monitors in our prototype is performed similarly in a tool developed by Giannakopoulou et al. [11], though their work focuses on the generation of monitors for run-time verification.

## 5 Conclusions

In this paper, we have explored a method by which useful properties can be verified to hold over individual translations performed by untrusted translators. We have shown that though widespread adoption of translators supporting translation validation has not yet been realized, and provably correct translators still remain difficult to build, the use of already existing model checkers provides a feasible means of establishing guarantees of translation correctness beyond the guarantees generally available.

In particular, we show the applicability of this method in the context of code generators used in model-based development, and have developed a prototype leveraging the NuSMV and CBMC model checkers. These model checkers, in conjunction with the infrastructure to translate LTL properties into C assertions, are used to verify that a set of LTL properties are preserved when generating C code from Simulink models. Notably, we applied our prototype to demonstrate that two code generators generated C code for a large industrial Simulink model correctly with respect to a set of 55 LTL properties. We believe this demonstrates both scalability and effectiveness of this method and intend to explore this method further in the near future.

We feel that this approach holds promise as a relatively easy to implement technique that can be employed by software developers concerned with the correctness of a untrusted translator, specifically code generators. This approach allows developers without access to the internals of a code generator (or without the desire to implement translation validation methods into a code generator) to achieve some guarantee of the correctness of an individual translation.

## 6 Acknowledgements

We would like to thank Dr. Michael Whalen from Rockwell Collins Inc. for allowing us to use his in-development C code generator, as well as for insights and discussions on the use of bounded model checkers.

## References

1. A. Biere, C. Artho, and V. Schuppan. Liveness Checking as Safety Checking. *Electronic Notes in Theoretical Computer Science*, 66(2):160–177, 2002.
2. A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking. *Advances in Computers*, 58:118–149, 2003.
3. E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barselona, Spain, March 29-April 2, 2004: Proceedings*, 2004.
4. C. Colby, P. Lee, G.C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. *ACM SIGPLAN Notices*, 35(5):95–107, 2000.

5. E. Denney and B. Fischer. Annotation inference for the safety certification of automatically generated code. *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE06)*, pages 265–268.
6. A. Duret-Lutz and D. Poitrenaud. SPOT: an extensible model checking library using transition-based generalized Bu/spl uml/chi automata. *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.(MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on*, pages 76–83, 2004.
7. P. Dybjer. *Using Domain Algebras to Prove the Correctness of a Compiler*. Springer.
8. Esterel-Technologies. SCADE Suite product description. http://www.esterel-technologies.com/v2/ scadeSuiteForSafetyCriticalSoftwareDevelopment/index.html, 2004.
9. R.W. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science*, 19(19-32):1, 1967.
10. Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18. Chapman & Hall, Ltd., 1996.
11. D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. *Proceedings, International Conference on Automated Software Engineering (ASE01)*, pages 412–416, 2001.
12. D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulae to Buchi automata. *Proceedings of the 22nd IFIP WG*, 6, 2002.
13. J.D. Guttman, J.D. Ramsdell, and M. Wand. VLISP: A verified implementation of Scheme. *Higher-Order and Symbolic Computation*, 8(1):5–32, 1995.
14. T. Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. *Modular Programming Languages: Joint Modular Languages Conference, JMLC 2003, Klagenfurt, Austria, August 25-27, 2003: Proceedings*, 2003.
15. A. Joshi and M.P.E. Heimdahl. Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In *SAFECOMP*, volume 3688 of *LNCS*, pages 122–135. Springer-Verlag, Sept 2005.
16. Mathworks Inc. Simulink product web site. Via the world-wide-web: http://www.mathworks.com/products/simulink.
17. Mathworks Inc. Stateflow product web site. Via the world-wide-web: http://www.mathworks.com.
18. J. McCarthy. Towards a mathematical science of computation. *Information Processing*, 62:21–28, 1962.
19. S.P. Miller, E.A. Anderson, L.G. Wagner, M.W. Whalen, and M.P.E. Heimdahl. Formal Verification of Flight Critical Software. In *Proceedings of the AIAA Guidance, Navigation and Control Conference and Exhibit*, August 2005.
20. Steven P. Miller, Alan C. Tribble, Michael W. Whalen, and Mats P. E. Heimdahl. Proving the shalls: Early validation of requirements through formal methods. *Int. J. Softw. Tools Technol. Transf.*, 8(4):303–319, 2006.
21. J.S. Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492, 1989.
22. G.C. Necula and P. Lee. The design and implementation of a certifying compiler. *ACM SIGPLAN Notices*, 33(5):333–344, 1998.
23. The NuSMV Toolset, 2005. Available at
http://nusmv.irst.itc.it/.

24. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS*, 98:151–166.
25. M. Rinard. Credible compilation. *Proceedings of the FLoC Workshop Run-Time Result Verification, July*, 1999.
26. *ARP 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment.* SAE International, December 1996.
27. D. Wheeler. SLOCCount: Source Lines of Code Count. *Webpage. Version*, 2, 2004.