

# Automated Validation of Software Models \*

Steve Sims    Rance Cleaveland  
Reactive Systems, Inc.  
www.reactive-systems.com  
sims,cleaveland@reactive-systems.com

Ken Butts  
Ford Motor Company  
kbutts1@ford.com

Scott Ranville  
New Eagle Software  
www.neweagle.net  
sranville@neweagle.net

## Abstract

*This paper describes the application of an automated verification tool to a software model developed at Ford. Ford already has in place an advanced model-based software development framework that employs the Matlab®, Simulink®, and Stateflow® modeling tools. During this project we applied the invariant checker Salsa to a Simulink®/ Stateflow® model of automotive software to check for nondeterminism, missing cases, dead code, and redundant code. During the analysis, a number of anomalies were detected that had not been found during manual review. We argue that the detection and correction of these problems demonstrates a cost-effective application of formal verification that elevates our level of confidence in the model.*

## 1 Introduction

This paper describes a project by engineers at Ford and Reactive Systems, Inc. (RSI) to investigate how automated verification tools can be most effectively packaged for quick integration into an industrial software development process such as that used by Ford. We anticipate that a number of companies, including those in the automotive, aviation and medical-device industries, will have similar needs, owing to the fact that many concerns in these fields employ model-based software engineering technology similar to Ford.

Ford already has in place an advanced model-based software development framework that employs the Matlab®, Simulink®, and Stateflow®<sup>1</sup> components of the engineering modeling environment built and marketed by the The MathWorks, Inc. [Mat]. For this work we selected a Ford Simulink®/Stateflow® model that specifies the behavior of a piece of the software in a powertrain controller. The model

\*Research supported in part by National Science Foundation Small Business Innovation Research Grant DMI-9961012

<sup>1</sup>Matlab®, Simulink®, and Stateflow® are registered trademarks of The MathWorks, Inc., Natick, MA.

contains nine Simulink® and six Stateflow® diagrams and has an implementation consisting of approximately 1000 lines of C code.

We analyzed this model using Salsa [BS00], a tool that automatically checks whether a given logical formula is an invariant of a given model. Salsa's invariant checker enables an engineer to formulate and check application specific properties such as *Will a car's anti-lock brakes ever engage when the brake pedal is not pressed?* Additionally, Salsa can serve as the engine for *consistency checking* [TR99, HJL96, HL96], i.e. to compute entire classes of checks that any well-formed model should pass. For example all models should be free of nondeterminism, missing cases, dead code, and redundant code. Salsa includes facilities to automatically search a model for these four types of anomalies. The results of applying Salsa to the production model exceeded our expectations. The verification effort revealed one piece of dead code, one redundant update, and several violations of Ford's modeling guidelines in a model that had previously undergone extensive manual review and simulation. Although we cannot state that our analysis proves that the model or code is *correct*, the detection and correction of several anomalies and the assurance that the resulting model is free from other instances of these types of errors demonstrates a cost-effective application of formal verification that elevates our level of confidence in the model. Ford uses automated testing and code generation to guarantee that the C code conforms to the model thereby ensuring that a better model leads to better software.

## 2 Background

### 2.1 Ford Software Development Process

We now briefly describe Ford's existing software development process [BCD<sup>+</sup>01, SPS<sup>+</sup>00] and explain how the analyses performed in this project could be incorporated to enhance it.

Ford has implemented an advanced model-based software development process that benefits from tool support at

each stage from requirements capture through system testing. The process is centered on executable models developed during the early stages of the process. Preliminary experience is showing that the extra effort required for modeling is recouped in several ways. First, the models serve as an effective communication medium for the dozens of engineers responsible for the development of a piece of automotive software and the components it interacts with. Second, the models facilitate reuse among different but related pieces of software (e.g. powertrain software for different vehicles) and from one generation to the next (e.g. model year 2000 to 2001). Finally, the models enable rigorous validation to be applied earlier in development thereby allowing problems to be detected earlier when they are much less costly to fix.

For modeling, the Powertrain Division of Ford has selected the Matlab®, Simulink®, and Stateflow® components of the engineering modeling environment built and marketed by the The MathWorks, Inc. [Mat]. Simulink® is a graphical block diagramming language that enables the specification of mathematical constraints over input and output values of variables. Simulink® also provides a means of specifying the hierarchical decomposition of a system into simpler subsystems. Stateflow®, a version of the well known Statecharts notation [HP98], supports the specification of the behavior of a system component in terms of the states the component may enter and the transitions that indicate how the component evolves from one state to another. The tools offer extensive support for interactive and automatic simulation of models.

A key focus of Ford’s methodology is that as development moves from one stage to the next, the artifact of a stage (e.g. requirements, model, software) is validated against previous stages. For example, an automated test harness executes a model in parallel with its software implementation to ensure that the software conforms to its model. Ford has also developed a set of modeling style guidelines and consistency checks [TR99], many of which are performed automatically by tools developed in house at Ford and with partners. The primary goal of this work was to extend the set of checks that could be performed on Simulink®/Stateflow® models beyond the capabilities of the tools currently integrated into Ford’s software development process. We were able to meet this goal as described in the remainder of the paper.

## 2.2 Salsa

The tool we used to perform our analysis was *Salsa* [BS00], an invariant checker for models specified in SAL (the SCR Abstract Language). To check whether a formula is an invariant of a SAL model, Salsa carries out an induction proof (without any user guidance) that utilizes

tightly integrated decision procedures, currently a combination of BDD algorithms and a constraint solver for integer linear arithmetic. The user interface of Salsa is designed to mimic the interfaces of widely used model checkers such as Spin [Hol97], SMV [McM93], and the CWB-NC [CS96]; i.e., given a formula and a model, Salsa either establishes the formula as an invariant of the model or provides a *counterexample* to explain why the given formula is not an invariant of the model. In either case, the algorithm will terminate.

Salsa has attributes of both a model checker and a theorem prover. It is automatic and provides counterexamples like a model checker, but like a theorem prover, it uses decision procedures, can handle infinite state systems, and can use auxiliary lemmas to complete an analysis. Currently, Salsa implements decision procedures for propositional logic, the theory of unordered enumerations, and integer linear arithmetic.

The use of induction enables Salsa to combat the state explosion problem that plagues model checkers – Salsa can handle specifications whose state spaces are too large for model checkers to directly analyze. The model examined in this project is a good example because it contains inputs from large integer domains making its analysis with finite-state model checkers infeasible, but a good fit for Salsa. Note that since operations over the integer values are such an integral part of the model, widely used abstraction techniques could not be employed to make finite-state analysis possible. Salsa does have several drawbacks viz-a-viz traditional model checkers. The primary disadvantage of Salsa (and proof by induction in general) is its incompleteness – a failed check does not necessarily imply that a formula is not an invariant because the returned state pair may not be reachable. No such false negatives were encountered during this project however. A second drawback is the format of *counterexamples* provided by Salsa when a check fails. Unlike model checkers, which return an execution sequence from the start state, Salsa returns a counterexample that is a state or a state pair and not a complete execution sequence. Finally Salsa currently lacks the capability to check more general properties (such as liveness), but as we describe below a very wide variety of checks may be built on top of Salsa’s invariant checking engine.

## 3 Analyses Performed

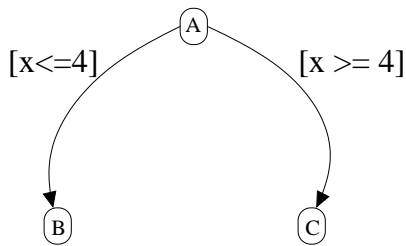
We now describe in more detail the types of checks we performed using Salsa.

### 3.1 Nondeterminism

Nondeterminism is a useful modeling construct often necessary to specify the uncertainty inherent in the envi-

ronments in which embedded software operates. Therefore, many modeling languages, including SAL, have constructs for representing nondeterminism. It is almost always the case, however, that the software in embedded systems should behave deterministically, i.e. for a given input there should be only one possible response (the right one!). We therefore determined that checking the model for nondeterminism would be useful.

Nondeterminism can arise in a number of ways in a specification. For example, consider the following fragment of a Stateflow® diagram that contains three states and two transitions.



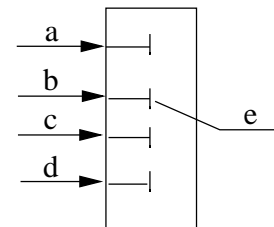
Each condition in brackets labeling a transition is a *guard* that indicates when the transition it labels may fire. If the diagram is in state *A*, then it may evolve to state *B* if  $x \leq 4$  and it may evolve to *C* if  $x \geq 4$ . Note that in the case when  $x = 4$  the successor of *A* could be either *B* or *C*. We can search for such instances of nondeterminism using an invariant checker. In this case the formula we would propose as an invariant is  $\neg(x \leq 4) \vee \neg(x \geq 4)$ . Obviously, since both disjuncts are false when  $x = 4$ , the formula is not an invariant of the diagram and therefore nondeterminism is possible. Salsa has the capability to automatically generate the invariants necessary to check an entire model for nondeterminism.

It should be noted that the semantics of Stateflow® actually has a rule for resolving the nondeterminism described above. Specifically, the edges emanating from a state are evaluated in order, as determined by Stateflow®’s *clockwise rule*; starting at the 12 o’clock position of a state and proceeding clockwise around the state, conditions are evaluated, and the first transition to have a true condition is fired. However, although the above diagram is deterministic because of the clockwise rule, Ford has determined

that relying on this rule in practice leads to misunderstandings among designers. Consequently, Ford’s style guidelines [TR99] explicitly forbid a reliance on the clockwise rule, and therefore flagging situations such as the one above is of interest to Ford. A similar Stateflow® rule determines the order in which states in a diagram are evaluated based on their position on the screen. Although relying on this rule is not a violation of Ford’s style guidelines, it is useful to identify and document these cases for full-disclosure and maintenance purposes.

### 3.2 Missing Cases

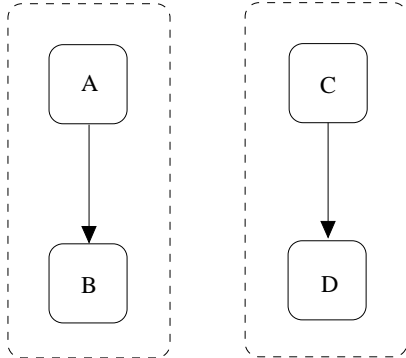
Missing cases are another common type of error in both programming and modeling. Multiport switches are a type of Simulink® block where the possibility arises for a modeler not to consider all cases. Multiport switches are represented graphically as follows.



This block uses the value of *a* to determine its output *e*. If the value of *a* is 1, 2, or 3, then the output is set to the value of *b*, *c*, or *d* respectively. If *a* is any value other than 1, 2, or 3 when the switch is evaluated, then a missing case exists. We can check for such errors by checking whether  $(a = 1) \vee (a = 2) \vee (a = 3)$  is an invariant when the switch is encountered. Salsa includes a routine to automatically generate the invariants necessary to check for missing cases.

### 3.3 Mutual Exclusion Violations

Another interesting check is to determine which states in a Stateflow diagram could be simultaneously active. This is a generalization of the well know critical-section problem, i.e. no two processes should be in their critical sections at the same time. In the case of Stateflow®, we want to check whether two states are simultaneously active. For example, consider the diagram in Figure 1. We can determine whether *B* and *D* are mutually exclusive by checking whether  $\neg(\mathbf{in}(B) \wedge \mathbf{in}(D))$  is an invariant. Here  $\mathbf{in}(B)$  is a property that holds when control resides in state *B*. The



**Figure 1. Checking mutual exclusion of two states in a Stateflow® diagram may be performed as an invariant check.**

invariants to check for mutual exclusion violations were formulated by hand.

### 3.4 Dead and Redundant Code

Due to the limited amounts of ROM available in their on-board processors, Ford engineers must deal with very stringent constraints on program size. The need to minimize program size makes the detection of dead or redundant code essential to the development of automotive software.

*Dead code* is simply defined as code that will never be reached during model execution. For example, if a Stateflow® diagram includes a transition  $A \xrightarrow{[x \leq 4]} B$  and the value of  $x$  is always larger than 4 when the diagram is in state  $A$ , then this transition will never fire and therefore the transition is dead code. Moreover, if this transition is the only one leading to  $B$  then  $B$  is dead code as well. We can determine if this transition is dead by checking whether  $\neg(\text{in}(A) \wedge x \leq 4)$  is an invariant. If it is, then it must follow that whenever  $\text{in}(A)$  is true, then  $x \leq 4$  must be false, implying that the transition can never fire. Due to the importance of finding dead code, we extended Salsa during the project to automatically generate the invariants necessary to search a model for dead code and applied the new check to the production model that we examined.

*Redundant code* is defined as code that can be removed from a model without changing its behavior. A common

type of redundancy occurs when a variable is updated twice before being read. In such cases the first update can be eliminated without changing the system’s behavior. We are able to find such problems using Salsa’s nondeterminism check as follows. In SAL all updates to a variable are collected into a single definition of the variable’s behavior. Therefore multiple updates to a variable within the same execution step are flagged as nondeterminism.

## 4 Analysis of Production Ford Model

In this section, we describe the results of applying Salsa to the production model of software from a Ford powertrain controller.

### 4.1 The Model

The model we selected for analysis specifies the behavior of an on-board diagnostic feature named *Self Test* that allows service technicians to monitor and test a powertrain control system. Two primary tests are performed by Self Test depending on the state of the engine: *Key On Engine Off (KOEO)* and *Key On Engine Running (KOER)*. Results of the tests are reported back using output codes controlled by the *Output Test Mode (OTM)* component of the software. The model contains nine Simulink® and six Stateflow® diagrams and has an implementation consisting of approximately 1000 lines of C code. For a more detailed description of the development of the Simulink®/Stateflow® Self Test model see [SPS<sup>+</sup>00]. Unfortunately, since the models are proprietary, we are unable to make the complete models publically available.

### 4.2 Performing the Model Validation

The first step in our analysis was to manually translate the Simulink®/Stateflow® version of the Self Test model into SAL, Salsa’s input language. This translation was by far the most time- and labor-intensive step in the analysis. The translation was primarily mechanical and amenable to automation. Work is under way at RSI to build a tool REACTIS™ VALIDATOR that offers Salsa-style invariant checking and is also capable of directly importing Simulink®/Stateflow® models. Table 1 lists some characteristics of the three main components of the model. As each subsystem communicates with other parts of the system and the environment via its set of input and output ports, the number of such ports gives some indication of how much interaction the component engages in.

Salsa uses BDDs to manipulate a model’s transition relation efficiently. BDDs may be seen as a way of encoding, for a given set of boolean-valued variables, a collection of different assignments of values to these variables [Bry86].

To give an idea of the complexity of these BDDs, we list the number of boolean BDD variables required for each subsystem. Some of these BDD variables may have linear constraints over integers associated with them, and we also give the total number of such constraints. These figures taken together convey some idea about the relative complexity of the submodels.

Throughout the process of creating the SAL model, we used simulation as our first line of debugging. Simulation can reveal errors as simple as undeclared variables and misspelled variable names as well as more subtle behavioral inconsistencies. When simulation stopped revealing problems, we ran checks for nondeterminism and missing cases which revealed another wave of translation errors that we had made. After a number of iterations, we had a SAL model whose behavior mirrored that of Ford's Simulink®/Stateflow® model and were ready to proceed with our comprehensive verification efforts.

With the SAL model in hand, we applied each of the checks described in Section 3. The verification effort, the results of which are detailed in Table 2, revealed one piece of dead code, one redundant update, and several violations of Ford's modeling style guidelines. We also had a hard-copy printout of the model containing some annotations made by a Ford engineer. One remark stated that a particular statement was redundant; however, we were able to prove using Salsa that the removal of the statement would have changed the behavior of the system. Neither the piece of dead code nor the redundant update that we detected with Salsa had been found during a rigorous but manual review process by Ford engineers to identify such code. These results support our contention that automated analysis offers a much more efficient and thorough means of detecting errors than manual techniques and gives evidence that this type of analysis extends the capability of Ford's current tool set.

### 4.3 Results

All checks described in this report were run on a Linux box powered by a 500 MHz Intel Pentium III processor and containing 512 MB of RAM. Note that Salsa contains built-in routines for searching for nondeterminism and missing cases, i.e. the tool automatically examines the model and determines the invariants that must be checked; however, the invariants to check for simultaneously active states had to be formulated by hand.

The dead code detected by Salsa occurred in component *OTM*, as indicated by the six failures during the dead code check. (Each failure is a different manifestation of the same piece of dead code.) The one failure in the nondeterminism check of subsystem *KOEO* showed the previously mentioned redundant update. Each of the failures in the nondeterminism check for component *KOER* points to a case

where the execution of the system depends on the graphical placement of a state on the screen, which as mentioned in Section 3 is not a violation of Ford's style guidelines, but is useful to know for full-disclosure and maintenance purposes.

Note that Table 1 shows that component *KOER* is the largest of the three; this explains the longer run times for the checks. The nondeterminism check of component *KOER* was invoked with Salsa's `-a` (approximation) flag given an argument of 2 which means there is a small probability that one of the formulas reported to be an invariant is actually not an invariant, i.e. a false positive occurs. All other checks were run with no approximation, so any formula reported as an invariant is guaranteed to be an invariant. The full model, i.e. the composition of the three components, is currently too big for Salsa to handle.

## 5 Conclusions

The number of problems detected by Salsa during the analysis of the production model is a clear and concrete measure of the potential of this technology. The project demonstrates that formal verification tools can be successfully applied in an industrial setting. Furthermore the fact that the majority of the verification effort resided in the manual translation from MathWorks to SAL gives us hope that the application of this type of checking can be performed in a very cost-effective manner once an automatic translator is completed.

Several key insights emerged from the project:

- *The automation of mundane tasks such as model translation and verification condition generation is as important as a sophisticated verification engine to the success of integrating formal methods into an industrial software development process*
- *Even when proving that software is correct is not cost effective, formal verification tools can be useful for detecting anomalies.*

While very successful, the project did uncover some limitations of current verification tools. First, the work revealed that traditional model checkers such as those found in Spin, SMV or the CWB-NC will not work with models such as those in place at Ford, because the state spaces of the models are simply too large to handle. Second, although Salsa performed quite well for the model we examined, it will also need to be extended to handle some MathWorks models. In particular Salsa's current set of decision procedures (for boolean, enumerated, and linear integer constraints) must be augmented to include a decision procedure for constraints over rationals.

**Table 1. Characteristics of the main subsystems of the production Ford model.**

<i>Subsystem</i>	<i>Number of</i>			
	<i>Inputs</i>	<i>Outputs</i>	<i>BDD Variables</i>	<i>Integer Constraints</i>
<i>OTM</i>	4	4	137	112
<i>KOEO</i>	5	9	160	131
<i>KOER</i>	13	25	183	108

**Table 2. Results of applying Salsa to the production Ford model of a component of the software in a powertrain controller.**

<i>Check</i>	<i>Number of Invariants</i>	<i>Number of</i>		<i>Time to Perform Check (in seconds)</i>
		<i>Passes</i>	<i>Fails</i>	
<b>OTM</b>				
Nondeterminism	84	84	0	6.7
Missing Cases	8	8	0	0.5
Dead Code	100	94	6	4.9
<b>KOEO</b>				
Nondeterminism	214	213	1	7.1
Missing Cases	44	44	0	1.2
Dead Code	182	182	0	2.3
<b>KOER</b>				
Nondeterminism	238	227	11	175.6
Missing Cases	13	13	0	24.8
Dead Code	144	144	0	54.7

## References

- [BCD<sup>+</sup>01] K. Butts, J. Cook, C. Davey, J. Friedman, P. Menter, S. Raman, N. Sivashankar, P. Smith, and S. Toeppe. Automotive powertrain controller development using CACSD. In Tariq Samad, editor, *Perspectives in Control Engineering: Technologies, Applications, and New Directions*. IEEE Press, 2001.
- [Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.
- [BS00] R. Bharadwaj and S. Sims. Salsa: Combining constraint solvers with BDDs for automatic invariant checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '00)*, Berlin, March 2000. Springer-Verlag.
- [CS96] R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In R. Alur and T. Henzinger, editors, *Computer Aided Verification (CAV '96)*, Lecture Notes in Computer Science, pages 394–397, New Brunswick, New Jersey, July 1996. Springer-Verlag.
- [HJL96] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.
- [HL96] M. Heimdahl and N. Leveson. Completeness and consistency in hierarchical state-based requirements, 1996.
- [Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Softw. Eng.*, 23(5):279–295, May 1997.
- [HP98] David Harel and Michal Politi. *Modeling Reactive Systems With Statecharts*. McGraw-Hill, 1998.
- [Mat] Home page of The MathWorks, Inc. <http://www.mathworks.com/>.
- [McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
- [SPS<sup>+</sup>00] P. Smith, S. Patel, W. Sun, R. Ramanan, H. Donald, S. Toeppe, S. Ranville, D. Bostic, and K. Butts. CACSD in production development : An engine control case study. In *Proceedings of the Global Powertrain Congress*, Detroit, MI, June 2000.
- [TR99] S. Toeppe and S. Ranville. An automated inspection tool for a graphical specification and programming language. In *Proceedings of the 12th International Software Quality Week Conference*, San Jose, CA, May 1999.