# Simulink Design Verifier vs. SPIN – a Comparative Case Study

Florian Leitner and Stefan Leue

Department of Computer and Information Science
University of Konstanz, Germany
{Florian.Leitner,Stefan.Leue}@uni-konstanz.de

**Abstract.** An increasing number of industrial strength software design tools come along with verification tools that offer some property checking capabilities. On the other hand, there is a large number of general purpose model checking tools available. The question whether users of the industrial strength design tool preferably use the built-in state space exploration tool or a general purpose model checking tool arises quite naturally. Using the case study of an AUTOSAR compliant memory management module we compare the Simulink Design Verifier and the SPIN model checking tool in terms of their suitability to verify important correctness properties of this module. The comparison is both functional in that it analyzes the suitability to verify a set of basic system properties, and quantitative in comparing the computational efficiency of both tools.

## 1 Introduction

It is increasingly common for industrial strength software design tools for embedded systems to provide verification capabilities. This capabilities aid in checking software design models for important system properties. Examples include the SDL tool Telelogic TAU[1], the Statemate ModelChecker[2] and the Simulink Design Verifier (SLDV)[3] which is an optional component of the Matlab/Simulink tool set. These verification tools are tightly integrated with the design tool that they belong to. The design tools are typically domain specific. The Telelogic TAU tool, for instance, enjoys widespread popularity in telecommunications while Matlab/Simulink is a de-facto standard for software design in automotive system engineering [1].

On the other hand, there is a large number of model checking tools available that are not tightly integrated with some software design tool, but instead can be used as stand-alone tools and possess a tool-specific input language. Examples of such tools include SMV [2], NuSMV [3] and SPIN [4]. There is a tendency in the literature to view symbolic model checking tools such as SMV and NuSMV to be more suitably applicable in the hardware domain, since they seem to deal particularly well with regularly structured systems. Symbolic model checking also

[1] http://www.telelogic.com/products/tau/index.cfm
[2] http://modeling.telelogic.com/products/statemate/add-ons/testing-and-validation.cfm
[3] http://www.mathworks.com/products/sldesignverifier/

seems to more adequately deal with data oriented design models. Explicit state model checking on the other hand, as implemented in SPIN, is generally viewed to cope more easily with irregularly structured concurrent software systems [4].

Designers of safety-critical software applications are faced with the challenge to decide whether they should rely on the use of the built-in verification engines, or whether they should invest effort and resources into either a manual translation of their design models into the input language of a general purpose model checker, or even into building a translation tool that automatically translates their design into a general purpose model checking tool. It is the objective of this paper to discuss the basis for this decision making based on the evaluation of the Simulink Design Verifier versus SPIN documented in much more detail in [5]. The evaluation was performed in the setting of a software system from the automotive domain. For the purpose of the comparison, we considered the SLDV, since it is integrated into a domain-specific design tool, and the model checker SPIN, since it seems an obvious choice for the analysis of the type of concurrent software that we are interested in.

## 2 Related Work

The verification engine underlying SLDV is based on SAT solving technology [6]. There is a substantial body of literature available that discusses formal semantics and analysis approaches to Matlab/Simulink and the Stateflow language. An operational semantics of Stateflow is defined in [1]. The authors observe that in spite of the relative complexity of the Stateflow language, the portion that is generally recommended to be used in practice has a surprisingly easy semantics. In [7] the formal semantics of a fragment of Stateflow is described based on a modular representation called communicating pushdown automata. Various papers describe translations of the Stateflow language into various model checking tools. A translation into the symbolic model checker SMV is presented in [8] while [9] discusses a translation into the NuSMV symbolic model checker, a successor to SMV. The paper [10] suggests an analysis of Simulink models using the SCADE design verifier in the particular setting of system safety analysis. Another translation from Stateflow to Lustre is described in [11]. Both share the interpretation of Simulink and Stateflow as a synchronous language, as mandated by the use of Lustre as the target language for the translations. Invariance checking of Simulink/Stateflow automotive system designs using an symbolic model checker is proposed in [12]. A first attempt to validate Stateflow designs using SPIN is presented in [13]. However, the translation tool used is not publicly available and it is therefore difficult to compare with this approach. In conclusion, we are not aware of any study that relates the SLDV tool to the SPIN model checker, which is the focus of this paper.

## 3 The Tools

*SLDV* is an extension of the Mathworks Matlab / Simulink tool set. It performs exhaustive formal analysis of Simulink models to confirm the correctness of Simulink models with respect to given properties. The user can specify these properties directly in Simulink and Stateflow as assertions. The assertions are specified in the

form of prove objectives. To prove these objectives, SLDV searches for all possible values for a Simulink or Stateflow function to find a simulation that satisfies an objective. SLDV does not expect a model to be closed. It will compute all possible input signals automatically. As written in [14] SLDV uses the Prover Plug-In [6] to prove properties. The Prover Plug-In is developed and maintained by Prover Technology [4]. The key idea of the Prover Plug-In is to model problems as propositional logic formulas and to apply proof methods to decide whether the formulas are tautologies, that is the the formula evaluates to true for all assignments, or not. This problem is solved by the Prover Plug-In using a reductio ad absurdum argument, where a combination of different analyses is used to check whether the negation of the formula is unsatisfiable. As was also observed in [1], Simulink does not support an interpretation of concurrency in terms of a nondeterministic interleaving of concurrent events. Even if Stateflow charts could be executed concurrently, Simulink will interpret their execution in a predefined, deterministic order based on the visual layout of the chart. SLDV does not provide any means to check a model for any type of concurrency related properties, such as race conditions. Likewise, deadlocks could only be detected if the one deterministic execution that Simulink permits happens to end in a deadlocked state.

*SPIN* is an explicit state model checking tool. The Promela modeling language, which is interpreted by SPIN, allows for describing concurrent system models. The concurrent processes in Promela synchronize via asynchronous or synchronous message passing, and via shared variables. SPIN implements a simple Depth-First Search algorithm for model checking safety properties, and a nested Depth-First Search for checking properties specified in Linear Temporal Logic or as a variant of Büchi automata. To deal with large state spaces SPIN offers bit state hashing as well as automated partial order reduction. SPIN has been applied in the analysis of many concurrent software systems. A very comprehensive overview of SPIN is given in [4].

## 4 The NVRAM Case Study

For the assessment of the alternatives we used the design of a *Non Volatile Random Access Memory Manager* (NVRAM) component from the AUTOSAR[5] automotive open system architecture. AUTOSAR is being defined by many representatives of the automotive industry and its main goal is to increase the hardware independence of single software modules and high reuse. AUTOSAR promotes the usage of commercial and well tested off-the-shelf software and will also help to reduce the development costs of automotive systems. The NVRAM Manager [15] provides the central memory management of AUTOSAR control units. The application has only access to the random access memory (RAM). Upon request from the application the NVRAM Manager copies data from read only memory (ROM) to RAM, RAM to non volatile memory (NVM) or vice versa. The NVRAM Manager provides services to ensure the data storage and maintenance of non volatile data according to

[4] http://www.prover.com/
[5] http://www.autosar.org

the individual requirements in an automotive environment. An example for such a requirement is to handle concurrent access to the memory. The main reason why we choose the NVRAM Manager was its independence from the application. Each automotive application has to control memory, that is why the NVRAM Manager will be found in almost every AUTOSAR control unit regardless whether this is an airbag control unit or an electronic stability program control unit.

The study that we present compares the two verification approaches both qualitatively and quantitatively. We proposed a set of central requirements on the NVRAM component and discussed whether and how they can be verified using either of the approaches. For those requirements that can be verified using both approaches we compared the performance of both. We also comment on the effort necessary to convert the Stateflow model to Promela, the input language of the SPIN tool. For the time being this translation is manual. We decided not to translate Stateflow directly into Promela, but to use the visual modeling tool VIP [16] as a target for the manual translation. VIP is a visual modeling interface that allows for concurrent system modeling using hierarchical, communicating state machines. VIP compiles efficient Promela code that encodes states using a state-label and goto mechanism. Notice that our analysis does not hinge upon the use of VIP, even though we perceive it as supporting the manual translation well.

## 5 Simulink Design Verifier vs. SPIN

### 5.1 Evaluation Approach

For the purpose of the case study the NVRAM Manager was modeled in Simulink. The NVRAM Manager consists of two parts, one for the job scheduling and one for the data handling. The central function of the NVRAM Manger is the job scheduling, the data management part just stores the configuration of the memory blocks. Therefore we decided to focus the case study on the job scheduling part. Some of the coding features in the model would impede state space explosion since they lead to very large state spaces. Hence, we manually created and abstracted Simulink model.

This Simulink model was manually translated in a visual specification using the VIP tool. VIP was designed as a visual input language for the SPIN model checker. For details on VIP we refer the reader to [16]. In contrast to the SLDV, SPIN can only verify closed models and hence we also modeled the environment behavior in the VIP model. VIP creates a concurrent proctype for each capsule which results in a concurrent system model. However, the two state machines in the Simulink model are executed in a sequential order. The concurrent execution of these capsules in our VIP model corresponds to the reality of the implementation, in which the corresponding tasks will be executed concurrently. Simulink does not allow for the modeling of concurrent behavior. To model the non-concurrent execution in VIP/Promela, we introduced a global variable lock that controls the execution of the state machines. We included both the sequential and the concurrent model in the experimental comparison.

In order to compare the verification capabilities of both tools we selected 6 central properties required to hold of the NVRAM manager. These properties were

derived from the requirements on the NVRAM manager specified in [15]. Some of these properties establish temporal contexts and hence require some form of temporal logic capture and corresponding model checking, while others are merely assertional.

To perform a quantitative comparison of the performance of SLDV and SPIN we measured the runtime of the verifications runs. Runtime here means the time that is needed to analyze the model. One would prefer to have other measurements than runtime to make a quantitative comparison, but unfortunately SLDV does not make any quantitative characteristics of a verification run available. To reduce the impact of other processes running on the same machine that may slow down the verification, we performed each run 10 times and produce here the average runtime. All tests have been executed on the same machine with 2GB of main memory and a 2 GHz dual core CPU. In addition to the verification runs, we scaled the models by adding queues to see how the runtime relates to the size of the model.

### 5.2 Quantitative Comparison

In the following, we summarize the results of the quantitative comparison. For a more comprehensive overview on the results we refer the reader to [5].

**Assertion Checking** The verification of the properties that could be represented as assertions, SLDV needed an average runtime of 3.04 seconds, while SPIN needed 0.0018 seconds for verifying the sequential version and 0.0239 seconds for the concurrent version. According to verification runs with SLDV and the sequential version of the SPIN model, all properties were satisfied. Whereas some concurrency related bugs were found in the concurrent version of the SPIN model.

**Deadlock Checking** A deadlock occurs when concurrent processes are waiting for each other or a shared resource in a cyclic fashion. We verified one model variant in which writing a new job to a full queue will result in a blocking of the system. SPIN detects the resulting deadlock and produces an error trail leading into the deadlock state.

SLDV does not provide for deadlock detection. If one forces the Simulink simulation environment into the corresponding deadlock state it will be stuck in that state and wait for input.

**Temporal Property Checking** Three of the six properties we selected can not be modeled as assertions since they assert properties of temporal contexts. The verification of temporal properties that cannot be encoded in assertions is beyond the capabilities of SLDV. We were hence restricted to SPIN to verify these properties. The properties were modeled using linear temporal logic (LTL).

The average runtime for verifying temporal properties was 0,009 seconds for the sequential SPIN model and 0,015 seconds for the concurrent SPIN model.

The verification of the concurrent version of the model revealed that two properties are violated because of an unexpected interleaving of environment and system

actions. The common cause is that the environment starts a new command before the processing of the previous command has been completed.

**Scaling of the Model** To see how the runtime relates to the size of the model, we added a third and a fourth queue to the model. The verification with SLDV takes noticeably longer as the number of queues is increased (runtime in seconds: 2 queues: 3.623; 3 queues: 19.065; 4 queues: 26.789), while in SPIN the runtime for the sequential model remains the same (runtime in seconds: 2 queues: 0.0022; 3 queues: 0.002; 4 queues: 0.0016) and the runtime for the concurrent model doubles from 3 queues to 4 queues (runtime in seconds: 2 queues: 0.0385; 3 queues: 0.0201; 4 queues: 0.0836).

### 5.3 Lessons Learned

The following appear to be the lessons learned from our case study on verifying system properties using SPIN and SLDV:

- We observed a significant discrepancy between the concurrent software architecture that we were modeling and the deterministic execution semantics that Simulink imposes. In applications like the NVRAM manager the capabilities of SPIN to deal with concurrency and to reveal concurrency related bugs appear to be far superior to the capabilities of SLDV.
- SLDV and SPIN allows for the checking of assertions. However, SLDV does not allow for expressing properties that go beyond assertion checking and which require a formalization using temporal logic formulae or equivalent mechanisms. Notice that according to [17] temporal properties following the response pattern, as we used to capture our temporal properties belong to the most frequently used properties in practice.
- In terms of runtime SPIN appears to be superior for the problem instances that we considered.
- SLDV automatically closes the system against its environment, while in SPIN it is necessary to incorporate the environment into the model, which requires extra modeling effort, but gives the user the chance to control the environment behavior.
- The translation from the limited subset of Simulink constructs used in the NVRAM example to VIP was fairly straightforward. The preservation of the semantics of Stateflow, however, requires special attention.
- Both SLDV and SPIN provide counterexamples if a property is violated.
- The effort for translating the NVRAM Manager to VIP has paid off. Not only because SPIN was superior in terms of runtime, but but also because it was the only way to verify all properties we had selected for verification.
- The first author of this paper, who performed the majority of the modeling and verification, was very familiar with AUTOSAR and the NVRAM component, had a moderate amount of using Matlab/Simulink and SPIN, but was new to LTL model checking and the use of the SLDV. He estimated the effort for the different modeling and verification tasks as follows (given in Person Days (PD)):

- Editing of the original Simulink NVRAM model: 10 PDs.
- Editing of the abstracted Simulink NVRAM model used in the verification: 2 PDs.
- Familiarization with VIP and editing of the VIP models: 2 PDs.
- Property formalization and performing SLDV verification runs for assertional properties: 2 PDs.
- Property formalization and performing SPIN verification for all 6 properties: 3 PDs.

It can be concluded that the use of SPIN certainly requires additional effort, which on the other hand gives access to verification capabilities that SLDV does not possess.

### 5.4 Threats to Validity

This case study cannot claim any form of empirical validity, since just one problem instance has been investigated. We also limited our considerations to the relatively simple Simulink language constructs that were useful in our case study. While other researchers also consider a limited scope of the Simulink language, our conclusions regarding the good correspondence between the Stateflow and the VIP/Promela model may be invalid in case a different fragment of Simulink is considered. However, we are confident that a large portion of Simulink can easily be interpreted, not at least due to the results in [13]. In terms of the superiority of SPIN in terms of verification runtime for assertional checking we have not investigated how this advantage develops as the model grows bigger, and possibly semantically more complex.

## 6 Conclusion

We have presented a case study in which we compared the use of the SLDV and the SPIN model checker in the verification of important properties of the AUTOSAR NVRAM component. The quantitative comparison of both approaches is hampered by the unavailability of statistics regarding the verification run of SLDV. In terms of run time SPIN appears to be superior for the problem instance that we considered. From a qualitative perspective, SPIN appears to be more suitable in supporting the concurrent nature of the NVRAM software than SLDV. SPIN is is also able to verify both assertional and temporal properties, while SLDV can only check assertions. Unlike SPIN, SLDV does not require a system model to be closed, which saves modeling effort compared to SPIN.

Our case study addressed a component that can be classified as belonging to general system infrastructure that the AUTOSAR architecture defines. In further research we will investigate whether for other such components an equally straightforward and well scaling translation from a limited fragment of the Simulink language into VIP and/or SPIN is possible. We will then consider an integrated tool environment, in which models and verification results can be exchanged amongst the components Simulink, VIP and SPIN.

## References


1. Hamon, G., Rushby, J.: An operational semantics for stateflow. Int J Softw Tools Technol Transfer **9** (2007) 447–456
2. McMillan, K.: Symbolic Model Checking. Kluwer Academic (1993)
3. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: Nusmv 2: An opensource tool for symbolic model checking. In: Proc. CAV 2002. (2002)
4. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addision–Wesley (2003)
5. Leitner, F.: Evaluation of the matlab simulink design verifier versus the model checker spin. Technical report, University of Konstanz (2008)
6. Andersson, G., Bjesse, P., Cook, B., Hanna, Z.: A proof engine approach to solving combinational design automation problems. In: Proc. 39th Design Automation Conference (DAC'02), IEEE Computer Society Press (2002) 725
7. Tiwari, A.: Formal semantics and analysis methods for Simulink Stateflow models. Technical report, SRI International (2002) http://www.csl.sri.com/∼tiwari/-stateflow.html.
8. Banphawatthanarak, C., Krogh, B., Butts, K.: Symbolic verification of executable control specifications. In: Proc. 1999 IEEE Int. Symposium on Computer Aided Control System Design, IEEE (1999)
9. B., M., Bhatnagar, A., Roy, S.: Tool for translating simulink models into input language of a model checker. In: Proc. ICFEM 2006. Number 4260 in Lecture Notes in Computer Science, Springer Verlag (2006) 606–620
10. Joshi, A., Heimdahl, M.: Model-based safety analysis of simulink models using scade design verifier. In: Proc. SAFECOMP 2005. Number 3688 in Lecture Notes in Computer Science, Springer Verlag (2005) 122–135
11. Scaife, N., Sofronis, C., Caspi, P., Tripakis, S., Maraninchi, F.: Defining and translating a 'safe' subset of simulink/stateflow into lustre. In: Proc. EMSOFT'04, ACM (2004)
12. Sims, S., Cleaveland, R., Butts, K., Ranville, S.: Automated validation of software models. In: Proc. ASE 2001, IEEE Computer Society Press (2001)
13. Pingree, P., Mikk, E., Holzmann, G., Smith, M., Dams, D.: Validation of mission critical software design and implementation using model checking. In: Proc. Digital Avionics Systems Conference, IEEE (2002)
14. The MathWorks Inc.: Simulink design verifier 1 user guide. Available from URL http://www.mathworks.com/access/helpdesk/help/pdf_doc/sldv/sldv_ug.pdf (2008)
15. AUTOSAR GbR: Requirements on memory services v. 2.2.1. Available from URL http://autosar.org/download/AUTOSAR_SRS_MemoryServices.pdf. (2008)
16. Kamel, M., Leue, S.: VIP: A visual editor and compiler for v-promela. In: Proc. TACAS 2000. Number 1785 in Lecture Notes in Computer Science, Springer Verlag (2000) 471–486
17. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: 21st Conference on Software Engineering (ICSE), IEEE Computer Society (1999)