Executing Model Checking Counterexamples in Simulink

Jiří Barnat, Luboš Brim
Faculty of Informatics, Masaryk University, Botanická 68a, 60200 Brno, Czech Republic

Jan Beran, Tomáš Kratochvíla, Ítalo R. Oliveira
Honeywell International, Aerospace, Advanced Technology Europe, Brno, Czech Republic

***Abstract*—Verification of embedded systems has become increasingly important in many industrial domains. Safety-critical embedded systems, such as those developed in aerospace industry, are regularly subject to automated formal verification process. In this paper we extend our tool integration chain of parallel, explicit-state LTL model checker DIVINE and Matlab Simulink tool suit with an improved support of counterexample simulation. In particular, we show how to provide the verification engineer with a direct connection between the error discovered by the model checker and the simulation in Matlab Simulink. This work has been conducted within the Artemis project industrial Framework for Embedded Systems Tools (iFEST).**

***Keywords*-Model checking, DIVINE, Simulink, Counterexample**

I. INTRODUCTION

Verification of developed embedded systems is one of the major consumers in the overall development process in terms of time and cost. The safety-critical nature of embedded systems in avionics industry makes the situation even worse, as it calls for thorough process of validation, verification, and certification. Automated formal verification methods, such as model checking [1], are used to significantly alleviate the burden of the cost of verification and validation process in this case.

An important lesson learned from the effort spent on verification of avionics systems [2] shows that the major bottleneck preventing successful application of model checking to verification of avionics systems is the scalability of verification tools. While symbolic model checking tools have been successfully applied in model-based development based on Mathworks Simulink [3], their scalability is unfortunately quite limited. One of the factors limiting scalability is the absence of distributed tools: none of the symbolic model checkers can distribute the work among multiple computation nodes [4]. On the other hand, explicit-state model checking has been repeatedly reported to scale well in a distributed-memory environment [5], [6].

Consequently, we have created an integrated toolchain [7] composing tools used for the development of embedded systems in avionics. In particular, we have interconnected Simulink and the explicit-state model checker DIVINE [8]. This connection allows a design engineer to verify systems under development against properties specified in Linear Temporal Logic (LTL) [9]. LTL is the standard formalism for expressing important behavioural properties; see e.g. Process Specification Language [10]. The choice of DIVINE model checker stemmed from the simple fact that to our best knowledge, DIVINE is the only LTL model checker that can fight the scalability limits by means of parallel and distributed-memory processing.

The primary goal of our toolchain as described in [7] was to enable the verification of Simulink designs with an explicit-state model checker. While this was achieved in principle, minor attention was paid to help the design engineer to understand outputs of the verification process. The model checking verification consumes a model of the system under verification and a specification the model should meet. For those two inputs it performs an algorithmic decision about the validity of the system with respect to the specification. If the system meets the specification, the model checking procedure simply returns a message informing the user about the fact. However, should any behaviour of the system under verification violate the specification, the negative answer to the verification is supported with the so called counterexample, i.e. behaviour of the system witnessing the invalidity of the specification. The complete process flow can be seen in Figure 1.

In this paper we focus at the process of interpretation of the counterexample as returned by the model checker back to the Simulink environment. By doing that we help the design engineer to directly see the evidence of the erroneous behaviour. This is surprisingly as important for the successful application of the tool-chain as the feasibility of the process as such. Moreover, besides accelerated process of detection and correction of errors in Simulink models, the visual interpretation of counterexamples often leads to a discovery of misinterpreted requirements.

II. PRELIMINARIES & RELATED WORK

Model checking is an automated formal verification procedure that takes a model of a formal (most commonly computer) system and decides whether the model satisfies a

This work has been partially supported by the Czech Grant Agency grant No. GAP202/11/0312, Artemis-IA iFEST project grant No.100203.

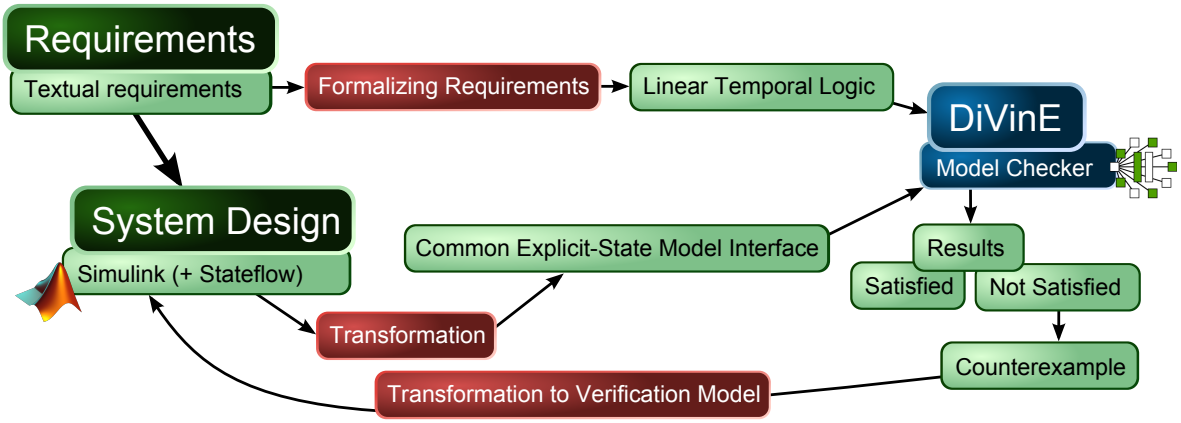

Figure 1. The complete formal verification process flow.

given property. Due to the exhaustive state-space search that is behind the model checking procedure, model checkers can formally prove properties such as absence of a deadlock in a concurrent system, or (un)reachability of a particular system state. However, the real advantage of the technique lies in the ability to check for temporal properties of systems. An example of temporal property that is most often checked with a model checker is the response property, i.e., *System will always react by performing action R to every stimulus action S*. Linear Temporal Logic (LTL) [9] is often used in combination with explicit-state model checkers to formalise the requirements. Formulae of LTL are built from the so-called *atomic propositions*, boolean-typed expressions evaluated on individual system states, and boolean and temporal operators.

Numerous approaches to automatic formal verification exist. Explicit-state model checkers such as DiVinE, enumerate and store individual states of the system. Implicit state (symbolic) model checkers such as NuSMV, store sets of the states using a compact symbolic representation. Satisfiability modulo theories (SMT) tools such as SAL and Prover use a form of induction to reason about models containing real numbers and unbounded arrays. Numerous attempts exist to combine different verification approaches into hybrid verification methods. For an example of combination of model checking, testing, and simulation approach based on SystemC see [11].

Simulink is a model-based design tool that allows design, simulation and code generation for dynamic and embedded systems [3]. A Simulink model is a design of the system that is built from a set of interconnected blocks. In this paper we only consider Simulink models made of discrete-time blocks. This is because model checking tools in general (and DiVinE in particular) can only analyse discrete systems. Discrete blocks produce an output at specific points in time, governed by a global discrete clock.

Extensive work has been spent on translating Simulink/Stateflow models to modelling languages of specific model checkers. Among the most relevant, it is the translation of Simulink models into the native language of the symbolic model checker NuSMV [12]. Another approach [13] suggests to analyse Simulink models with the SCADE design verifier. Translation from Stateflow to Lustre has been described as well [14]. Invariance checking of Simulink/Stateflow automotive system designs using a symbolic model checker is also proposed in [15]. Note that all mentioned translations interpret Simulink and Stateflow charts by synchronous language. An attempt to validate Stateflow designs with explicit-state SPIN has been described [16], however, the extension to SPIN is not publicly available.

More recently a significant effort was spent at Rockwell Collins to built a set of tools [17]–[19] that translate Simulink models into languages of several formal analysis tools, allowing direct analysis of Simulink models using model checkers and theorem provers. Among the tools considered are symbolic model checkers, bounded model checkers, model checkers using satisfiability modulo theories and theorem provers.

While application of symbolic methods to verification of Simulink models is more intuitive and can be applied with little cost, interesting observation was made in [20], [21]. It has been shown that explicit-state model checkers can outperform symbolic model checkers if there is some effort spent on manual abstraction and modelling.

III. Counterexample to Simulink

Once the DiVinE model checker is provided with the system and the Linear Temporal Logic formula it returns a confirmation that the formula is satisfied, or alternatively, a counterexample which demonstrates the behaviour of the system that falsifies the formula. We now focus only on the latter case, when a counterexample is generated.

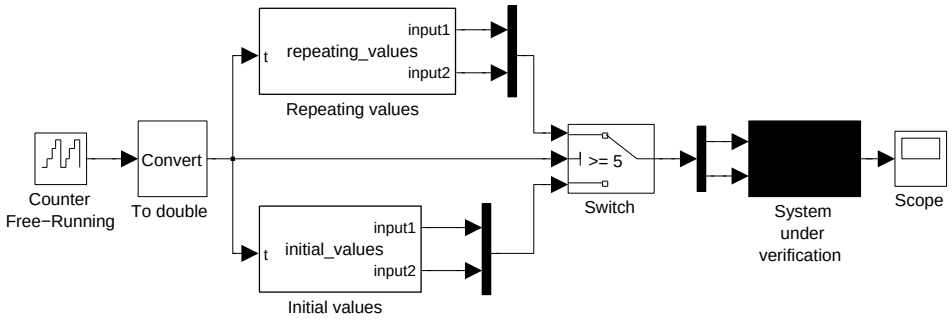

Figure 2. Verification model enabling system simulation guided by the counterexample.

```
===== Trace from initial =====

[LTL: 1] [1, 1]
[LTL: 1] [4, 0]
[LTL: 1] [3, 0]
[LTL: 1] [4, 0]
[LTL: 3] [3, 1]

===== The cycle =====

[LTL: 2] [7, 1]
[LTL: 2] [7, 0]
[LTL: 2] [0, 1]
```

Figure 3. When the formulae is not satisfied the counterexample is returned by a model checker.

Under the LTL model checking setting, the counterexample is always an infinite execution of the system. Since the systems under analyses are finite, an infinite counterexample can only be achieved by forming a lasso-shaped structure. As a result, the counterexample consists of two parts. The first part of the counterexample trace witnesses a path from the initial state of the system to a state on the cycle, while the second part is the sequence of states that is repeated infinitely many times (the cycle). An example of counterexample as returned by the DIVINE model checker is depicted in Figure 3. The values of all model inputs are shown in the second square brackets on every line of the counterexample. Each line corresponds to a single step of the execution of the systems and its duration corresponds to Simulink model sample time. Therefore, the model behaviour falsifying the property is fully defined by the counterexample. For the purpose of backporting the counterexample into Simulink, we ignore the contents of the first square brackets which shows the state of a Büchi automaton that is run synchronously with the system as an LTL property monitor.

The problem we address is that the counterexample in this form is extremely difficult to read. Therefore, engineers are often neither able to reproduce the wrong behaviour nor even able understand it. In the following we present the way how to simulate the counterexample within the Simulink environment in order to demonstrate the undesired system behaviour.

```
% ===== Trace from initial =====
function [input1, input2] = initial_values(t)
counterexample1 = [1 4 3 4 3];
counterexample2 = [1 0 0 0 1];
index = mod(t,5)+1;
input1 = counterexample1(index);
input2 = counterexample2(index);
```

Figure 4. Embedded Matlab Function providing initial values for each input in each time step.

```
% ===== The cycle =====
function [input1, input2] = repeating_values(t)
counterexample1 = [7 7 0];
counterexample2 = [1 0 1];
index = mod(t-4,3)+1;
input1 = counterexample1(index);
input2 = counterexample2(index);
```

Figure 5. Embedded Matlab Function providing repeating values for each input in each time step.

For the purpose of counterexample interpretation we have introduced a Simulink template model, shown in Figure 2, which can be used to simulate any Simulink system according to a counterexample. The initial non-repeating part of the counterexample is placed in the template as an Embedded Matlab Function block named Initial values. The listing of the Matlab Function for the initial part of the counterexample is given in Figure 4. This block is able to generate a series of vector outputs, one vector for each instant of time. This sequence of vectors is then used to drive the system from the initial state to some state on the cycle. Since there are five values (lines) in the first part of our counterexample as listed in Figure 3, the threshold of the Switch block in the verification template has been set to five.

Alter five time instants the verification template switches to the second Embedded Matlab Function block named Repeating values. See Figure 5 for the corresponding matlab code. This block generates a repetitive sequence of inputs for all subsequent time instants with a certain period (three in our case). Note that in our case, the system under verification has just two inputs that are repeated. However, the number may be arbitrary.

To simulate the counterexample, it remains to place the original system under verification as a subsystem in the verification template. Note that in our tool chain extension, we have automated the whole process including the mapping of system inputs to embedded function outputs.

IV. Conclusions and Future Work

In [7] we introduced a toolchain that allows explicit-state LTL model checking of Simulink systems improving thus the verification process for safety-critical embedded systems. In this paper we suggest a minor extension to the tool-chain that enables counterexample-guided simulation directly in the Simulink environment. This fills the gap of the tool-chain that is now ready to be used for verification of small to medium-sized stateful components. In the future, we would like to focus on tight control integration of the tools involved to directly invoke Simulink environment from the verification environment.

References

[1] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT press, 1999.

[2] D. Bhatt and K. Schloegel, "Effective Verification of Flight Critical Software Systems: Issues and Approaches," Presented at NSF/Microsoft Research Workshop on Usable Verification, November 2010.

[3] Mathworks. Simulink. [Online]. Available: http://www.mathworks.com/products/simulink/

[4] G. Ciardo, Y. Zhao, and X. Jin, "Parallel symbolic state-space exploration is difficult, but what is the alternative?" in *Parallel and Distributed Methods in Verification (PDMC)*, ser. EPTCS, vol. 14, 2009, pp. 1–17.

[5] K. Verstoep, H. Bal, J. Barnat, and L. Brim, "Efficient Large-Scale Model Checking," in *23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS 2009)*. IEEE, 2009.

[6] B. Bingham, J. Bingham, F. de Paula, J. Erickson, and M. Singh, G.and Reitblatt, "Industrial Strength Distributed Explicit State Model Checking," in *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC)*. IEEE, 2010, pp. 28–36.

[7] J. Barnat, L. Brim, J. Beran, and T. Kratochvíla, "Partial Tool Chain to Support Automated Formal Verification of Avionics Simulink Designs," in *Submitted to FMICS.*, 2012.

[8] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček, "DiVinE – A Tool for Distributed Verification (Tool Paper)," in *Computer Aided Verification*, ser. LNCS, vol. 4144. Springer-Verlag, 2006, pp. 278–281.

[9] Z. Manna and A. Pnueli, "The Modal Logic of Program," in *Proceedings of the 6th International Colloquium on Automata, Languages, and Programming (ICALP 1979)*, ser. LNCS, vol. 71. Springer-Verlag, 1979, pp. 385–409.

[10] C. Schlenoff, M. Gruninger, F. Tissot, J. Valois, T. C. Road, S. Inc, J. Lubell, and J. Lee, "The Process Specification Language (PSL) Overview and Version 1.0 Specification," 1999.

[11] J. Behrend, D. Lettnin, P. Heckeler, J. Ruf, T. Kropf, and W. Rosenstiel, "Scalable hybrid verification for embedded software," in *Design, Automation and Test in Europe (DATE 2011)*, 2011, pp. 179–184.

[12] M. B, A. Bhatnagar, and S. Roy, "Tool for translating simulink models into input language of a model checker," in *In ICFEM'06: Proceedings of the 8th International Conference on Formal Engineering Methods*. Springer, 2006, pp. 606–620.

[13] A. Joshi and M. P. E. Heimdahl, "Model-based safety analysis of simulink models using scade design verifier," in *SAFECOMP*, ser. Lecture Notes in Computer Science, vol. 3688. Springer, 2005, pp. 122–135.

[14] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi, "Defining and translating a "safe" subset of simulink/stateflow into lustre," in *EMSOFT*. ACM, 2004, pp. 259–268.

[15] S. Sims, R. Cleaveland, K. Butts, and S. Ranville, "Automated validation of software models," in *ASE*. IEEE Computer Society, 2001, pp. 91–102.

[16] P. Pingree, E. Mikk, G. Holzmann, M. Smith, and D. Dams, "Validation of mission critical software design and implementation using model checking," in *Proc. Digital Avionics Systems Conference*. IEEE Computer Society, 2002, p. 6A4–1 – 6A4–12.

[17] M. Whalen, D. Cofer, S. Miller, B. Krogh, and W. Storm, "Integration of Formal Analysis into a Model-Based Software Development Process," in *Formal Methods for Industrial Critical Systems*, ser. Lecture Notes in Computer Science. Springer, 2008, vol. 4916, pp. 68–84.

[18] D. Cofer, "Model Checking: Cleared for Take Off," in *Model Checking Software*, ser. Lecture Notes in Computer Science. Springer, 2010, vol. 6349, pp. 76–87.

[19] S. Miller, "Bridging the Gap Between Model-Based Development and Model Checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science. Springer, 2009, vol. 5505, pp. 443–453.

[20] M. Kim, Y. Choi, Y. Kim, and H. Kim, "Formal Verification of a Flash Memory Device Driver – An Experience Report," in *Model Checking Software*, ser. Lecture Notes in Computer Science. Springer, 2008, vol. 5156, pp. 144–159.

[21] Y. Choi, "From NuSMV to SPIN: Experiences with model checking flight guidance systems," *Formal Methods in System Design*, vol. 30, pp. 199–216, 2007.