Model Checking DSL-Generated C Source Code

Martin Sulzmann and Axel Zechner

Informatik Consulting Systems AG, Germany
{martin.sulzmann,axel.zechner}@ics-ag.de

Abstract. We report on the application of SPIN for model-checking C source code which is generated out of a textual domain-specific language (DSL). We have built a tool which automatically generates the necessary SPIN wrapper code using (meta-)information available at the DSL level. The approach is part of a larger tool-chain for developing mission critical applications. The main purpose of SPIN is for bug-finding where error traces resulting from SPIN can be automatically replayed at the DSL level and yield concise explanations in terms of a temporal specification DSL. The tool-chain is applied in some large scale industrial applications.

1 Introduction

The SPIN model checker [4] supports the embedding of native C source code for verification purposes. This has the advantage that there is no need to re-model the application in the PROMELA modeling language and the potentially error-prone model-to-model transformation step from a source model to PROMELA can be avoided entirely. As discussed in [5], the method is the easiest to apply in the verification of single-threaded code, with well-defined input and output streams. Our interest here is in the verification of synchronously executed C source code which perfectly matches these criteria.

The C source code we intend to model check is generated out of a textual domain-specific language (DSL) which is part of a DSL-based tool-chain for software development of mission critical systems. Figure 1 provides a summary. Our focus so far was on implementation and testing. What has been missing is static verification. To close this gap, we have integrated SPIN in our tool chain to guarantee a smooth integration between SPIN for model-checking and our DSL-based software development approach. See Figure 2.

The important point is that all testing and verification steps are performed at the C source code level where the C source code is generated out of the DSL. Hence, there is no need to validate the transformation step from DSL to C source code. The purpose of the DSL is to support 'higher-level' application/domain abstractions which in our experience has significant advantages compared to 'low-level' programming at the C source code level.

In this paper, we provide an overview of the purpose of the DSLs and their integration with SPIN. The particular contribution is the SPINRunner tool which effectively represents the SPIN-DSL integration described in Figure 2. Further details of the SPINRunner tool, e.g. the implementation (DSLs and tools) as well as some example from the Automotive area, are freely available via

`http://ww2.cs.mu.oz.au/~sulzmann/spin-dsls.html`

2 DSL Tool-Chain Overview

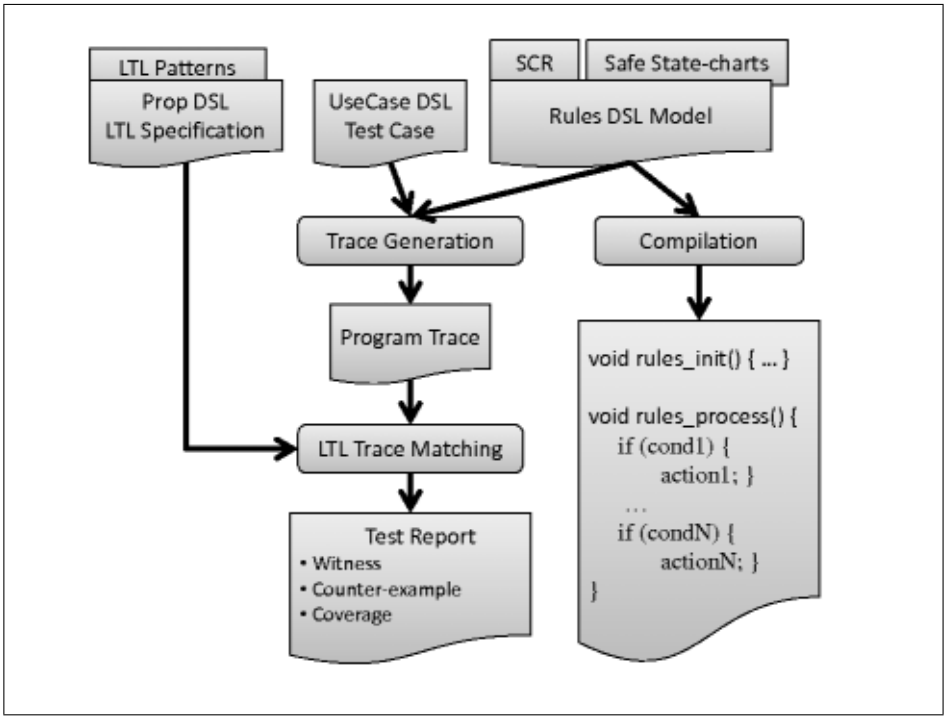

Fig. 1: DSL Tool-Chain Overview – Implementation and Testing

First, we review the various DSLs and their interaction in terms of implementation and testing. See Figure 1. For implementation we use a rule-based DSL in spirit of the Atom DSL [3] where in each cycle every rule is tried sequentially. Our DSL code generator translates each rule into straight-forward C code, in essence, simple if-then statements. Similar to SCADE [8], we generate a function `rules_init()` to initialize state variables and a periodically executed function `rules_process()`. All DSL variable declarations, e.g. input, output, state and local, are declared as global C variable declarations. Thus, we can ensure predictable memory consumption.

For testing, we use a DSL to specify use cases to stimulate the application. The stimulation is weaved together with the C code of the application and yields a test executable. Running the test executable yields a finite program trace which is then matched against a property DSL which describes linear temporal logic (LTL) [7] specifications. LTL trace matching yields a detailed test report based on the method described in [9].

The DSLs have been applied with success in some large scale industrial applications in the Aerospace& Defense area. An important feature is the ability to customize the DSLs to specific application needs. For example, we have built

numerous extensions such as Software Cost Reduction (SCR) [2] style mode and output tables, safe state-charts a la SCADE and new forms of LTL pattern abstractions [1] etc. Such extensions can be fairly quickly integrated in our approach thanks to our use of *internal* DSLs. The advantage of an internal DSL is that we can make use of the host language, in our case Haskell, to specify new constructs as 'library' extensions. That is, at the Haskell level, new constructs are mapped to existing constructs without having to implement new parsers, code generators etc.

3 SPIN-DSL Integration

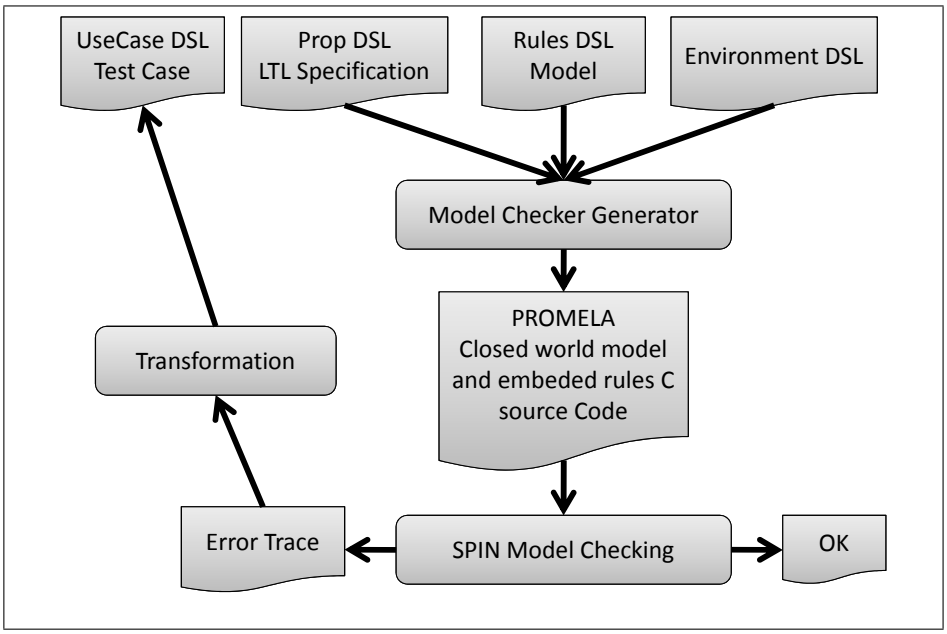

Fig. 2: SPIN-DSL Integration for Verification

Figure 2 gives an overview of the SPIN-DSL integration. The C source code generated out of the rule-based DSL is literally embedded into a PROMELA model which includes the LTL specification and a closed world model of the environment specified at the DSL level. The environment is represented by the input variables of the DSL model. To obtain a closed world model, we must set these inputs. For example, we can define equivalence classes among the set of input variables to reduce the state space. We also provide for a number of optional automatic optimizations by for example reducing the set of input variables to those used in the currently checked LTL property. The entire process of generating the PROMELA model out of the DSLs and performing the model checking is done automatically by the SPINRunner tool. That is, the user is freed from any low-level model checker interaction and can focus on the high-level DSL modeling part.

In our experience, this approach works fairly well for medium-sized examples. For example, we can fully model check a realistic example taken from the Automotive domain. For many real-world applications, the state space is simply too big such that model checking would yield an answer within some reasonable amount of time. For us this is not a serious issue. Our main use of SPIN is for bug-finding rather than fully static verification. In our experience, model checking often reveals many simple but tedious to spot coding errors by producing error traces.

The error trace can then be used to reach the point in the application where the violation occurs. Additionally, we can provide explanations in terms of the LTL specification which has been violated. Our tool automatically transforms the SPIN error trace into a test case of the UseCase DSL. Thus, we stimulate the application to obtain a program trace which is then matched against the LTL specification. Our constructive LTL matching algorithm provides a detailed test report which includes explanations which parts of the LTL specification have been violated.

```
active process ModelWrapper(){
  atomic{
    c_code { rules_init(); push_state(); }
  }
  do
  ::atomic{
    stimulate_inputs();
    c_code { pop_state(); rules_process(); push_state(); log_state(); }
  }
  od
}
```

Fig. 3: DSL-Generated SPIN Wrapper Process

Highlights of Model Checker Generator. *Model Checker Generator* (MCG) is the central component of the SPIN integration into our tool-chain. The MCG tool takes the DSL description of the model, specification, and some environment constraints to automatically generate the input for SPIN model checking:

1. LTL DSL statements in SPIN expression format.
2. Bit-optimal representation of DSL variables.
3. A wrapper process to execute the rule-based DSL model.

For brevity, we ignore the first two points which are fairly straightforward. For example, the bit-optimal representation of DSL variables reduces memory consumption and thus may allow 'longer' model-checker runs.

Figure 3 shows the central components of the SPIN wrapper code to execute the DSL model (i.e. its C code representation). We first atomically initialize the DSL model by executing `rules_init()` followed by repeated non-deterministic atomic execution of `rules_process()`.

Functions `push_state()` and `pop_state()` exchange state information between SPIN and the C interface of our DSL model. We only need to keep track of DSL variables which represent state and global input and output. Any locally declared DSL variable can be ignored because such variables are functionally defined by the surrounding context.

Function `stimulate_inputs()` represents the environment model for closed-loop verification. We non-deterministically select global input values. To reduce the set of input combinations, and thus the state-space of model-checking, we build equivalence classes of input values (as discussed in [6]). The definition of equivalence classes can be specified at the DSL level and has the consequence that model-checking is potentially incomplete. That is, depending on the representative of the equivalence class, an actual violation of an LTL specification might remain undetected. As already mentioned, our main motivation for the integration of DSL is for bug-finding and the ability to replay error traces at the DSL level. Hence, the incompleteness issue is not of major concern for us.

The trace logger function `log_state()` is activated during simulation of SPIN error trails. This function transforms the internal representation of valuations of input, output and state variables to a format readable for the subsequent steps of our tool chain.

4 Industrial Case Study: Motor-Start Stop (MSA)

MSA is application is an example from the Automotive area whose purpose is depending on the state of the vehicle to either switch on or switch off the engine (e.g. to save fuel, if the vehicle is not moving).

MSA consists of nine input variables, e.g. measuring the speed, brake pressure etc. There are three output variables to indicate the status of the MSA (on/off), engine recommendation (on/off) and MSA LED status (on/off).

The entire application is formalized by nine LTL statements. Here is a fairly simple statement which states that the LED shall be set if the MSA is active.

```
always $
  (valueOut msaStatus .==. constE MSA_Active)
  .=>.
    (valueOut msaLed .==. constE On)
```

Our DSLs are implemented as *internal* DSLs which come come with a bit of syntactic overhead because we re-use the syntax of the host language. Extra combinators such as `valueOut` and `constE` are required to embed the DSL into Haskell. The significant advantage of internal DSLs is that we quickly build new DSL extensions as libraries.

The implementation of the MSA application makes use of SCR [2] style mode and output tables. These extensions are mapped to the simple `rule`'s construct which provides the basis of the Rules DSL. The entire MSA application boils down to about 67 primitive rules.

Table 1 shows some benchmark results for our MSA example. The Tables show model-checking time / memory usage for different LTL properties. We have applied the optimizations mentioned in the previous sections. The results are obtained on a Dell Latitude E5510, Intel Core i5 CPU 2.67 GHz, 4GB Main

List of SPIN options used for benchmark:

Option 1: depth first search `-DSFH -DBFS -DSAFETY`
Option 2: safety with optimizations `-DSAFETY -DSFH`
Option 3: safety `-DSAFETY`
Option 4: acceptance

	Option 1		Option 2		Option 3		Option 4	
	time [s]	[kbytes]	time [s]	[kbytes]	time [s]	[kbytes]	time [s]	[kbytes]
Engine_Off	34.900	1,690,020	24.400	345,713	27.200	345,713	60.300	689,036
Engine_On1	41.000	1,690,117	-	-	-	-	-	-
Engine_On1b	7.030	402,840	1.830	345,518	1.780	345,518	1.960	688,841
Engine_On2	3.370	116,062	0.340	345,518	0.420	345,518	1.400	688,841
LED_Off	1.940	116,062	0.340	345,518	0.617	345,518	0.895	688,841
LED_On	2.400	116,062	0.605	345,518	0.315	345,518	1.620	688,841
MSA_Active	43.200	1,690,117	22.100	345,616	30.500	345,616	41.300	688,939
MSA_Inactive1	42.900	1,690,117	13.000	345,518	10.800	345,518	25.200	688,841
MSA_Inactive2	1.900	116,062	0.245	345,518	0.240	345,518	0.615	688,841

Table 1. Time / memory usage for different properties

Memory running with Windows 7 32-Bit. To get comparable results SPIN was run in single CPU mode only.

Acknowledgments

We thank the reviewers for their comments.

References

1. Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.
2. Stuart R. Faulk and Constance L. Heitmeyer. The SCR approach to requirements specification and analysis. In *Proc. of Requirements Engineering (RE'97)*, page 263. IEEE Computer Society, 1997.
3. Tom Hawkins. Atom DSL. http://hackage.haskell.org/package/atom/.
4. Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.
5. Gerard J. Holzmann and Rajeev Joshi. Model-driven software verification. In *Proc. of SPIN'04*, volume 2989 of *LNCS*, pages 76–91. Springer, 2004.
6. D. Richard Kuhn and Vadim Okun. Pseudo-exhaustive testing for software. In *Proc. of 30th Annual IEEE / NASA Software Engineering Workshop (SEW-30 2006)*, pages 153–158. IEEE, 2006.
7. Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.
8. Scade suite. http://www.esterel-technologies.com/products/scade-suite/.
9. Martin Sulzmann and Axel Zechner. Constructive finite trace analysis with linear temporal logic. In *Proc. of TAP'12*, volume 7305 of *LNCS*, pages 132–148. Springer, 2012.