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. For example, 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. We demonstrate the approach via a smaller example taken from the Automotive area.

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.

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. Details of the SPINRunner tool are described in the appendix which also outlines a tool demonstration. 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`

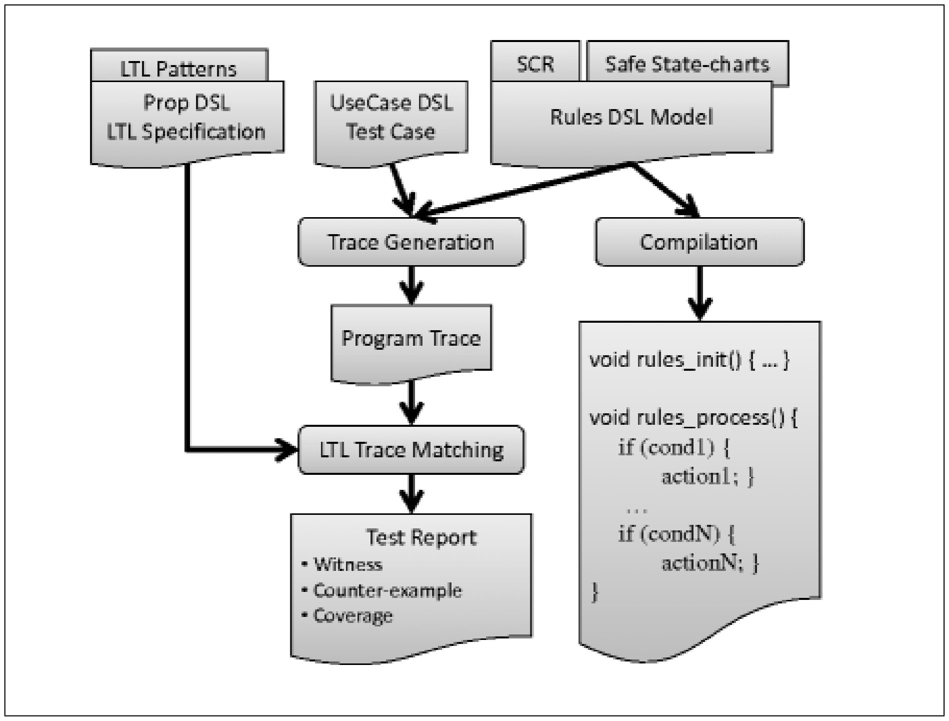

Fig. 1: DSL Tool-Chain Overview – Implementation and Testing

2 DSL Tool-Chain Overview

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

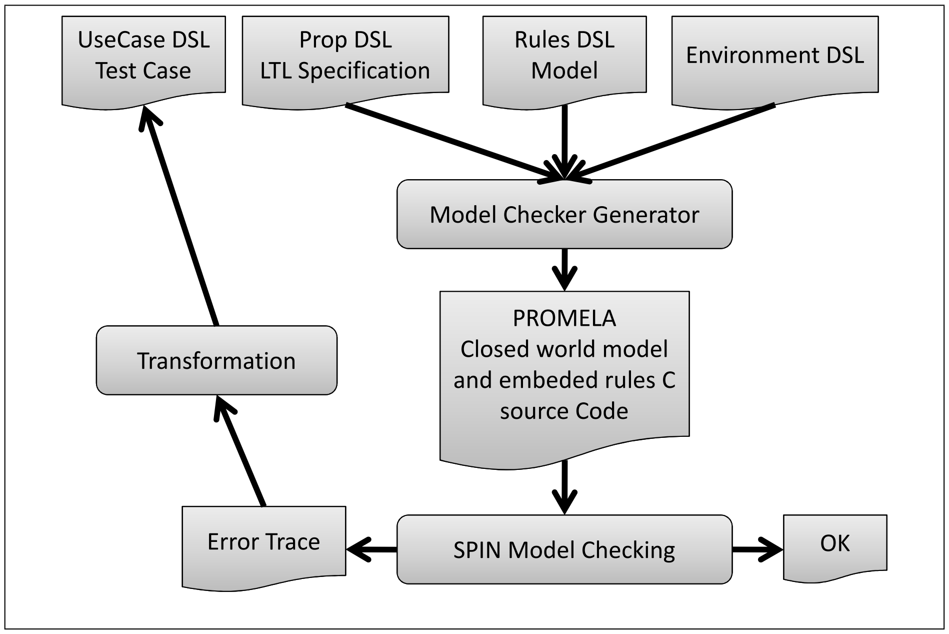

Fig. 2: SPIN-DSL Integration for Verification

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. 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.

3.1 Highlights of Model Checker Generator

```
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

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. 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.

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*, 2012. To appear.

A Tool Demonstration Outline

We demonstrate the SPINRunner with an example taken from the Automotive area. Specifically, we consider a motor start-stop automatic (MSA) application. The purpose of a MSA 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). Appendix B provides an overview of the MSA application. The SPINRunner is discussed in Appendix C.

B Motor Start-Stop (MSA) Example

Figure 4 shows the concrete interface of the MSA example encoded in the Rules DSL which is embedded in Haskell. DSL combinators `inp` and `outp` allow us to declare input and output variables. The first argument is a unique identifier whereas the second argument is the type to be used. Types in program text are not first-class in Haskell. Hence, we simply create a dummy program text, here `undefined`, which is assigned the desired type.

The translation to C is straightforward. For example,

```
speed           <- inp "SPEED" (undefined :: Speed)
```

translates to

```
int16_t IN_GENSPEED;
```

where the prefix 'IN_GEN' is used for internal naming reasons.

Figure 5 shows some snippets of the MSA applications where we make 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.

The mode table keeps track if the pressure is either normal, above a certain limit or some alarm mode applies. The alarm mode shall apply if the pressure is above the limit for a certain time period for which we use a counter variable. The counter's value is defined by the output value where for example we increment the current value if we are in the `PressureAboveLimit` mode.

Figure 6 formulates some of the requirements imposed on the MSA application as Prop DSL statement. The `embedGLTL` embeds a further domain-specific extension in the general LTL language. For example, we make use of 'higher-level' combinators such as

```
-- p .=|=|=>. (n,m) $ q
-- if p holds for min n or max m steps,
-- then thereafter q holds
```

The combinator `invariantTest` indicates that the LTL property shall be checked against all traces of this test suite. Requirement "MSA_Becomes_Active" states that the MSA becomes active if there's a rising key value edge and the MSA has been inactive earlier on. Requirements "Engine_Off" states that the engine shall be switched off if

```
-- type definitions
type Speed          = Int16 -- km/h
type Temperature    = Int16 -- Celcius
type Pressure       = Int16 -- 1/10 N/m^2
type Voltage        = Int16 -- 1/10 V
type SteeringAngle  = Int16 -- 0..360 Degree
data Gear = P | R | N | D deriving (Eq, Show, Bounded, Enum)
data OnOff = Off | On deriving (Eq, Show, Bounded, Enum)
data OpenClose = Open | Closed
                 deriving (Eq, Show, Bounded, Enum)
data PressedNotPressed = Pressed | NotPressed
                         deriving (Eq, Show, Bounded, Enum)
data OpMode =  MSA_Active | MSA_Inactive
               deriving (Eq, Show, Bounded, Enum)

appMSA = node "main" $ do
   -- inputs
   speed          <- inp "SPEED" (undefined :: Speed)
   temperature    <- inp "TEMPERATURE" (undefined :: Temperature)
   voltage        <- inp "VOLTAGE" (undefined :: Voltage)
   pressure       <- inp "PRESSURE" (undefined :: Pressure)
   steeringAngle  <- inp "STEERING_ANGLE" (undefined :: SteeringAngle)
   driversBuckle  <- inp "DRIVERS_BUCKLE" (undefined :: OpenClose)
   gear           <- inp "GEAR" (undefined :: Gear)
   frontLid       <- inp "FRONT_LID" (undefined :: OpenClose)
   driversDoor    <- inp "DRIVERS_DOOR" (undefined :: OpenClose)
   engineStatus   <- inp "ENGINE_STATUS" (undefined :: OnOff)
   msaKey         <- inp "MSA_KEY" (undefined :: PressedNotPressed)
   -- ouputs
   msaStatus      <- outp "MSA_STATUS" (undefined :: OpMode)
   engine         <- outp "ENGINE" (undefined :: OnOff)
   msaLed         <- outp "MSA_LED" (undefined :: OnOff)
   -- implementation is following here
```

Fig. 4: MSA Interface

- for 9 cycles, the (brake) pressure is above 15, and
- in the 10th cycle we have that
  - the pressure is above 15, and
  - the MSA is active, and
  - the speed is below 2, and
  - the temperature is above 3, and
  - the (battery) voltage is above 11.

In summary, the MSA application consists of a module `AppMSA` which represents the actual MSA implementation and a module `ReqMSA` which represents the LTL requirements imposed on the MSA application. See Figure 7. These modules and the corresponding 'main' functions `msaApp` and `reqs` are DSL inputs for the SPINRunnter tool.

```
-- mode table: pressure normal, above limit, alarm
mode .=:==.
   modeMachine "pressure" PressureNormal
        [ (PressureNormal .---->. PressureAboveLimit)
                     (pressure .>. limitPressure)
        , (PressureAboveLimit .---->. PressureNormal)
                     (limitPressure .>. pressure
                      .||. pressure .==. limitPressure)
        , (PressureAboveLimit .---->. PressureAlarm)
                     (pressure .>. limitPressure
                      .&&. (valueState counter) .==. constE 10)
        , (PressureAlarm .---->. PressureNormal)
                     (limitPressure .>. pressure)
        ]


 -- output table for a counter: increment if pressure above limit
 ifEq $
   counter =:=?=
       (valueVar mode .==. constE PressureNormal)
       .==:>.
       (constE (1::Word32))

       `elseIf`
       (valueVar mode .==. constE PressureAboveLimit)
       .==:>.
       ((constE (1::Word32)) .+. (valueState counter))

       `defaultCase`
       (valueState counter)
```

Fig. 5: MSA Implementation

C SPINRunner

The SPINRunner takes the DSL formulation of the application and its requirements and executes all steps from generation of the model checker (MCG) to SPIN model checking to replaying of error traces.

C.1 Overview of Execution Steps

Step 1 Read input models. SPINRunner reads and interprets the system model as Rules DSL model, specification as Prop LTL DSL Spec.

Step 2 Generate the C code. Then the system model described in Rules DSL is compiled to C code to be included by the SPIN Wrapper. Our code generator simply produces the end product C code as would be included in the final embedded system.

```
-- | condition about the state of the car ignoring the engine status
msa_on_car_cond = (valueIn frontLid .==. constE Closed)
             ./\. (valueIn driversDoor .==. constE Closed)
             ./\. (valueIn driversBuckle .==. constE Closed)
             ./\. (valueIn gear .==. constE D)
-- | car status including the engine status
msa_on_cond = msa_on_car_cond
         ./\. (valueIn engineStatus .==. constE On)

msaReqs = [
 invariantTest "MSA_Becomes_Active"
   (embedSeqLTL $
     always $
             (valueIn msaKey .==. constE NotPressed) ./\.
             (valueOut msaStatus .==. constE MSA_Inactive)
       .==>. 1 $
          (      ((valueIn msaKey .==. constE Pressed) ./\.
                  msa_on_cond)
            .=>. (valueOut msaStatus .==. constE MSA_Active)
          )
   )
 ,
    invariantTest "Engine_Off"
    (embedSeqLTL $
     always $
        (valueIn pressure .>. constE 15)
        .=|=|=>. (9,9) $
            (valueIn pressure .>. constE 15) ./\.
            (valueOut msaStatus .==. constE MSA_Active) ./\.
            (constE 2 .>. valueIn speed)  ./\.
            (valueIn temperature .>. constE 3) ./\.
            (valueIn voltage .>. constE 11)
           .=>.
              (valueOut engine .==. constE Off)
    )
 ]
```

Fig. 6: MSA Requirements

Step 3 Collect environment model. Often industrial-size model can only be rationally handled if integer type inputs are constrained to interesting scenarios. If supplied, SPINRunner collects information from the supplied environment model. Optionally, the tool can automatically infer restrictions from the LTL-specifications similar to equivalence class testing.

Step 4 Build SPIN wrappers. For each LTL formula specified, the Model Checker Generator (MCG) produces an individual wrapper and finally model checker.

```
module AppMSA where

msaApp = node "main" ...

module ReqMSA where

reqs = ...
```

Fig. 7: MSA DSL Input for SPINRunner

Step 4.1 Extract the model interface. MCG extracts all information from the system model to generate the wrapper: input, output, and state variables, user supplied types and enumerations. Integer types are mapped to corresponding SPIN types. Enumerations types are represented using the optimum bit-size.

Step 4.2 Generating state variables. All state variables including outputs and unconstrained integer type inputs are represented as Promela variables to include them into the state vector. All other inputs are combined to a single variable for compact representation of input configurations. To exchange information between the model C code, C functions *push* and *pop* (see Fig. 9) are generated surrounding the cyclically called system model.

Step 4.3 Input stimuli. For model-checking with SPIN, MCG generates code to stimulate the inputs of the system model. For exhaustive analysis, an input configuration is chosen non-deterministically in each cycle. As previously stated, input configurations for all enumeration-type and constrained variables are combined to a single combination variable which is assigned non-deterministically. MCG generates a list of all input combinations and a C function *setup_inputs* to assign the inputs from that list addressed by the combination number (see Fig. 10).

Step 4.4 Logging function. In order to allow further processing of a possible error trail, MCG produces the C function *dumpTrail* which dumps the variable contents during simulation runs. Variable configurations are dumped in a format to allow further processing in our tool chain.

Step 4.5 Model wrapper process. The model wrapper process cyclically calls the system model C code and generation of input stimuli. During simulation runs such as replay of an error trace the logging function *dumpTrail* is called. Fig. 8 depicts the output generated by this step.

Step 5 Compiling and executing the model checker. SPINRunner invokes SPIN and a C compiler (e.g. gcc) to generate the model checker executable *pan*. Compiler options for special SPIN optimizations or model checking variants can be supplied. As default breadth first search is activated which is restricted to safety properties but yields a good trade-off for performance. Finally, the pan model checker is called.

Step 6 Explaining an error trace. In case model checking yields an error trace, SPINRunner invokes pan again to dump the trace of input configurations which lead to the violation of the specified property.

```
active proctype ModelWrapper(){
  atomic{
    c_code{
      demonstratorInit();
      push_state();
    };
    select(_combiVar: 0..20735);

    c_code{
      pop_state();
      setup_input(now._combiVar);
      demonstratorRules();
      if (readtrail) dumpTrail();
      push_state();
    }
  }
  do
  ::atomic{
    select(_combiVar: 0..20735);

    c_code{
      pop_state();
      setup_input(now._combiVar);
      demonstratorRules();
      if (readtrail) dumpTrail();
      push_state();
    }
  }
  od;
}
```

Fig. 8: Promela model from the MSA case study

C.2 Approximation Reduction

A predominant challenge for model checking of industrial-size problems is to cope with the complexity of a model's inputs. Equivalence reduction is an approximative testing technique. SPINRunner implements this *pseudo-exhaustive* technique and can automatically derive equivalence classes from the Prop DSL Specification.

```
c_code{
\#include "setup_input.h"
\#include "dumpTrail.h"
  void push_state(){
    now.OUT_GENENGINE=OUT_GENENGINE;
...
    now.s_STATE_GENloc_engine54=s_STATE_GENloc_engine54;
    now._combiVar=0;
    now.IN_GENVOLTAGE=IN_GENVOLTAGE;
    now.IN_GENTEMPERATURE=IN_GENTEMPERATURE;
    now.IN_GENSPEED=IN_GENSPEED;
    now.IN_GENPRESSURE=IN_GENPRESSURE;
  }
  void pop_state(){
    OUT_GENENGINE=now.OUT_GENENGINE;
...
    s_STATE_GENloc_engine54=now.s_STATE_GENloc_engine54;
  }
}
```

Fig. 9: Exchange information between C code and SPIN model checker

```
typedef struct{
  int IN_GENSPEED;
...
  unsigned IN_GENMSA_KEY:1;
} tInputStimulus;
tInputStimulus cInputStimulus[]={
  {1,2,10,14,-31,Open,P,Open,Open,Off,Pressed},
...
};
void setup_input(int _combi){
  IN_GENSPEED=cInputStimulus[_combi].IN_GENSPEED;
...
  IN_GENMSA_KEY=cInputStimulus[_combi].IN_GENMSA_KEY;
}
```

Fig. 10: Stimulate the inputs adressed by an input combination

In Prop DSL propositions over integer variables x are of the form $x < c$ or $x = c$ where c is an integer constant. By traversing all provided LTL-expressions, the algorithm collects all propositions over a variable x. The propositions are then sorted and representatives chosen for each of the equivalence classes formed by the relation expressions.

```
useCase "emitEngine_OffFailureDerivedFromSPINErrorTrace" (
 -- Cycle #1
  (emitStep $
        speed           .=. constE (1::Int16)
    <+> temperature     .=. constE (2::Int16)
    <+> voltage         .=. constE (10::Int16)
    <+> pressure        .=. constE (16::Int16)
    <+> steeringAngle   .=. constE (-32::Int16)
    <+> driversBuckle   .=. constE Open
    <+> gear            .=. constE P
    <+> frontLid        .=. constE Open
    <+> driversDoor     .=. constE Open
    <+> engineStatus    .=. constE Off
    <+> msaKey          .=. constE NotPressed
  )
 <!> delay 7
 <!>
 -- Cycle #9
  (emitStep $
        temperature     .=. constE (3::Int16)
    <+> voltage         .=. constE (11::Int16)
    <+> driversBuckle   .=. constE Closed
    <+> gear            .=. constE D
    <+> frontLid        .=. constE Closed
    <+> driversDoor     .=. constE Closed
    <+> engineStatus    .=. constE On
    <+> msaKey          .=. constE Pressed
  )
 <!>
 -- Cycle #10
  (emitStep $
        temperature     .=. constE (2::Int16)
    <+> voltage         .=. constE (10::Int16)
    <+> pressure        .=. constE (14::Int16)
    <+> driversBuckle   .=. constE Open
    <+> gear            .=. constE P
    <+> frontLid        .=. constE Open
    <+> driversDoor     .=. constE Open
    <+> engineStatus    .=. constE Off
  )
 <!> delay 2
```

Fig. 11: DSL test case obtained from SPIN error trace

C.3 MSA Application

Applying the SPINRunner on our MSA example yields an error trace in case of the LTL requirement `Engine_Off` from Figure 6. The SPIN error trace is systematically turned into UseCase DSL test case. See Figure 11.

This test case is then used to stimulate the MSA application to obtain a (finite) program trace. Matching this program trace against the the LTL property `Engine_Off` fails. Based our constructive LTL matching method, we obtain the information that all pre-conditions

```
(valueIn pressure .>. constE 15) ./\.
(valueOut msaStatus .==. constE MSA_Active) ./\.
(constE 2 .>. valueIn speed)  ./\.
(valueIn temperature .>. constE 3) ./\.
(valueIn voltage .>. constE 11)
```

but the following does not hold

```
(valueOut engine .==. constE Off)
```

This suggests that there's a problem in our implementation of the pressure control logic. Indeed, the transition guard from Figure 5

```
(PressureAboveLimit .---->. PressureAlarm)
    (pressure .>. limitPressure
     .&&. (valueState counter) .==. constE 10)
```

is incorrect. Instead of the 10 cycles we should only wait 9 cycles. That is, we need to replace `constE 10` by `constE 9`. After this fix. All nine LTL requirements of the MSA application can be verified by SPIN.

C.4 Model-Checking Benchmarks

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]	[byte]	time [s]	[byte]	time [s]	[byte]	time [s]	[byte]
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_Becomes_Active	43.200	1,690,117	22.100	345,616	30.500	345,616	41.300	688,939
MSA_Becomes_Inactive1	42.900	1,690,117	13.000	345,518	10.800	345,518	25.200	688,841
MSA_Becomes_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

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

C.5 Summary: Invoking the SPIN Runner command line tool

Example usage for MSA case study. `$WP` is a shell variable pointing to the path where the output files should be placed.

```
SPINRunner -r $WP -w $WP -c gcc-4 -f ReqMSA.hs -t reqs
   -m AppMSA.hs -s msaApp
```

```
  -w FilePath             --workingPath=FilePath
```

Path to working folder

```
  -r FilePath             --spinModelPath=FilePath
```

Path to regression test folder

```
  -c Command              --compiler=Command
```

The C compiler you want to use (defaults to g++)

```
  -d Compiler Flags       --compileFlags=Compiler Flags
```

The compile flags you want to use (defaults to "-DSFH -DBFS -DSAFETY -DNOFAIR -DNOBOUNDCHECK -DXUSAFE")

```
  -p Modelchecker Flags  --modelcheckingFlags=Modelchecker Flags
```

The model checking flags you want to use (defaults to -m10000000)

```
  -f FilePath             --testFile=FilePath
```

The test file you want to use (defaults to supply req file), e.g. `ReqMSA`

```
  -t Name                 --testCases=Name
```

The test cases you want to use (defaults to supply test cases), e.g. `reqs`

```
  -E Name                 --testCases=Name
```

Environment model of possible values for a variable you want to use (defaults to Nothing)

```
  -m FilePath             --modelFile=FilePath
```

The model file you want to use (defaults to supply model file), e.g. `AppMSA`

```
  -s Name                 --modelName=Name
```

The model name you want to use (defaults to supply model name), e.g. `msa`

```
  -e True                 --useEqClsRed=True
```

Use equivalence class reduction from ltl formulas (defaults to True)